

# Chapter 6: Process Synchronization

---

**2020.6**  
**Howon Kim**

- 정보보호 및 지능형 IoT연구실 - <http://infosec.pusan.ac.kr>
- 부산대 지능형융합보안대학원 - <http://aisec.pusan.ac.kr>

# Chapter 6: Process Synchronization

---

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

# Objectives

---

- To present the concept of **process synchronization**.
- To introduce the **critical-section problem**, whose solutions can be used **to ensure the consistency of shared data**
- To present both **software and hardware solutions of the critical-section** problem
- To examine **several classical process-synchronization problems**
- To explore **several tools that are used to solve process synchronization problems**

# Background

---

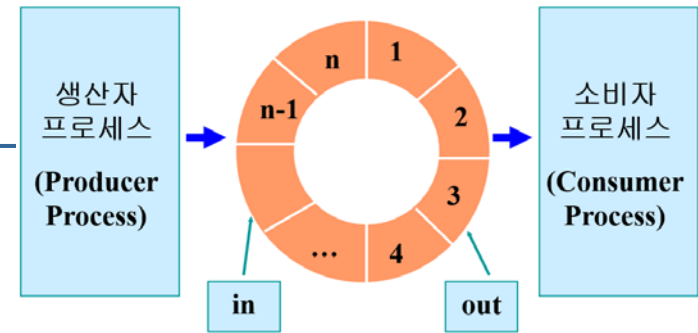
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in **data inconsistency**
- Maintaining data consistency **requires mechanisms to ensure the orderly execution of cooperating processes**

## Producer

```
while (true) {  
    /* produce an item in next  
    produced */
```

```
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;   
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

next 항목 생성 후, buffer에 write



### Illustration of the problem:

Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.

We can do so by having an integer **counter** that keeps track of the number of full buffers.

Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

## Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];   
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

# Race Condition 발생 !!

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter는 메모리상의 변수
- register는 CPU내부에 있음

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}

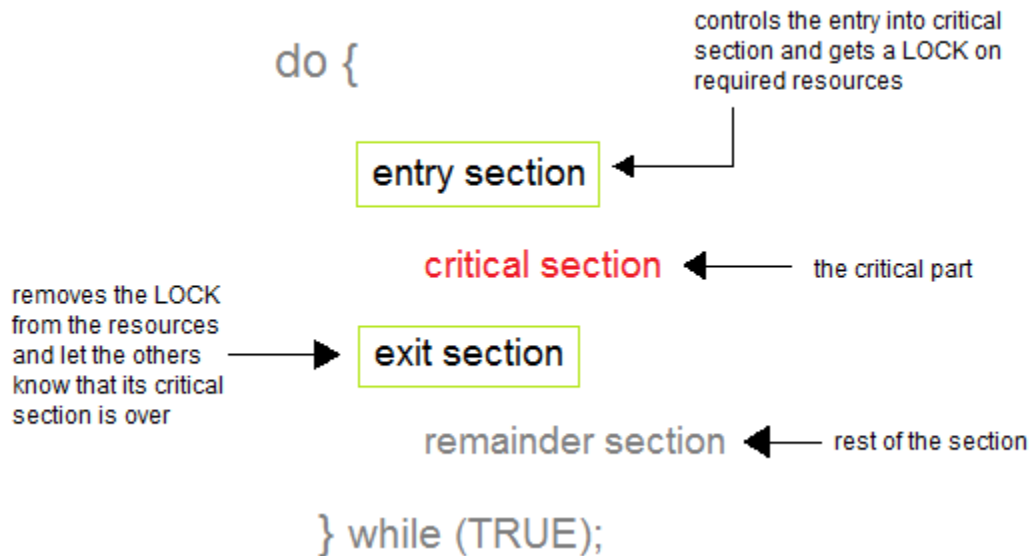
- 두 process가 동시에 counter 변수에 접근함 !
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- 최종 실행 결과는 프로세스나 thread의 실행 순서에 의해 달라져 버림. → 프로세스 동기화 필요함

# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
  - (임계구역 내에서,) Process may be changing shared variables, updating table, writing file, etc.
  - (임계구역의 중요한 특징) When one process in critical section, no other may be in its critical section
- **Critical section problem**
  - is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data
- Each process must **ask permission to enter critical section** in **entry section(진입구역)**, may follow critical section with **exit section(퇴출구역)**, then **remainder section(나머지구역)**

# Critical Section

- **A Critical Section** is a code segment that accesses shared variables and has to be executed as an atomic action.
  - It means that in a group of cooperating processes, at a given point of time, **only one process must be executing its critical section**.
  - If any other process also wants to execute its critical section, it must wait until the first one finishes
- General structure of process  $P_i$



- **진입 구역(entry section)**  
임계 구역에 진입하기 위해 허용을 요구하는 코드 영역
- **임계구역(critical section)**  
다른 프로세스와 공유하고 있는 변수를 접근 변경하거나 테이블을 갱신, 파일을 쓰는 등의 동작 수행함. atomic하게 동작해야 함
- **퇴출 구역(exit section)**  
임계 구역 다음에 나오는 영역으로써, 임계 구역을 벗어나기 위한 코드 영역
- **나머지 구역(remainder section)**  
프로그램의 나머지 코드 부분

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

# Algorithm for Process $P_i$

Shared variables:

`int turn;` initially `turn = 0`

`turn = i`  $\rightarrow$   $P_i$  can enter its critical section

Process  $P_i$

```
do {  
    while (turn != i) ; //it can be forever  
    critical section  
    turn = j;  
    remainder section  
} while (1);
```

자신의 차례가 됨(i)  $\rightarrow$  critical section

busy waiting  $\rightarrow$  while (turn != i) ;

Satisfies mutual exclusion, but not progress

# Solution to Critical-Section Problem

- 임계구역 문제에 대한 해결안은 다음과 같은 세가지 요구 조건을 만족해야 함
  1. **Mutual Exclusion (상호배제)** – If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
  2. **Progress (진행)** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
    - critical section에 프로세스가 없어서 진입 가능한 상황에서, 진입하고자하는 process들이 있다면, 이들은 무기한 진입을 위해 기다려서는 안됨
    - Deadlock avoidance 필요
  3. **Bounded Waiting (한계대기)** - A bound must exist on the number of times that other processes are allowed to enter their critical sections **after a process has made a request to enter its critical section and before that request is granted**
    - 하나의 프로세스가 임계 구역 진입을 요청한 후, 진입이 허용될 때까지 다른 프로세스가 임계 구역에 진입하는 횟수(시간)에 제한을 두어야 함
    - 만족시키지 못할 경우 starvation(기아상태) 유발
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the  $n$  processes

# Critical-Section Handling in OS

## ❑ Critical section 관련 해결책

- Software solutions
  - software algorithms who's correctness does not rely on any other assumptions
- Hardware solutions
  - rely on some special machine instructions
- OS solutions
  - OS provide some functions and data structures to the programmer

## ❑ Two approaches depending on if kernel is preemptive or non-preemptive

- ❑ Non-preemptive kernel – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - ▶ 특정 순간에 커널내에서 실행되는 프로세스는 하나밖에 없음 → kernel내 자료구조에 대한 race condition 발생하지 않음
- ❑ Preemptive kernel – allows preemption of process when running in kernel mode
  - ▶ race condition 발생하지 않도록 설계되어야 함
- ❑ Preemptive kernel mode에서는 race condition 발생 가능성이 있는데, 발생하지 않도록 설계 되어야 함

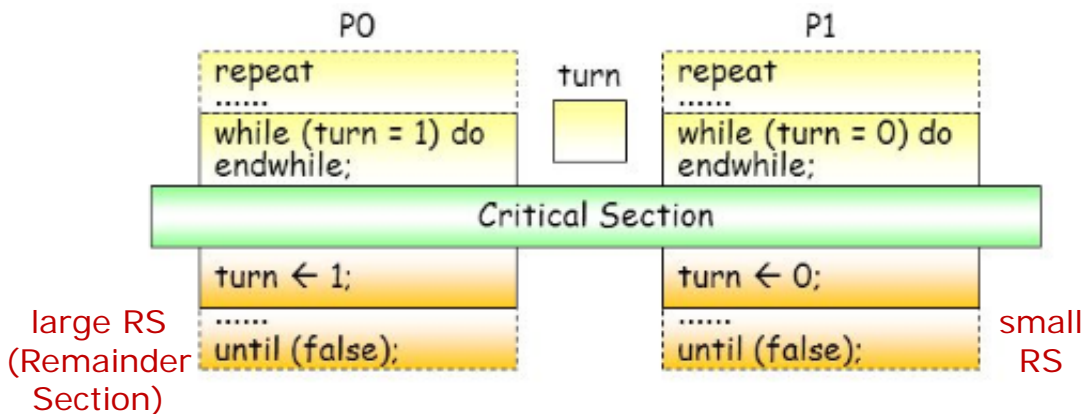
# SW 기반 방법

---

- process가 두 개가 있는 경우의 해결책
  - 두개의 processes : P0, P1
  - Algorithm 1, 2 → incorrect
  - Algorithm 3 → Correct (Peterson's algorithm)
  
- n개의 process를 위한 해결책
  - bakery algorithm

# SW 기반 방법 - Algorithm 1

- The shared variable **turn** is initialized (to 0 or 1) before executing any  $P_i$
- $P_i$ 's critical section is executed iff  $\text{turn} = i$
- $P_i$  is **busy waiting** if  $P_j$  is in CS: mutual exclusion is satisfied
- **Progress requirement is not satisfied** // “진행” 요구사항 만족 못시킴
- Ex:
  - $P_0$  has a large RS and  $P_1$  has a small RS. If  $\text{turn}=0$ ,  $P_0$  enter its CS and then its long RS ( $\text{turn}=1$ ).  $P_1$  enter its CS and then its RS ( $\text{turn}=0$ ) and tries again to enter its CS: request refused! He( $P_1$ ) has to wait that  $P_0$  leaves its RS.



```

Process  $P_i$ :
repeat
    while( $\text{turn} \neq i$ ) {};
    CS
     $\text{turn} := j$ ;
    RS
until FALSE
    
```

$P_0$ 가 CS 진입되어  $\text{turn} = 1$ 이 되었음. 그러면  $P_1$ 도 CS에 진입할 수 있게 됨.  
 $P_1$ 은 CS 수행후,  $\text{turn} = 0$ 으로 만든 후, 짧은 RS 수행후, 다시 repeat되어  $P_1$ 의 CS에 들어가려고 하는데, 당연히  $P_1$  진입안됨,  $\text{turn}$ 이 아직 0임.  
 $\text{turn}$ 이 1이 되기 위해서는  $P_0$ 가 CS 수행을 해야 하는데,  $P_0$ 는 이전 CS 진입후, 아직도 large RS를 수행하는데 머물러 있음. (이때는  $P_1$ 에 의해  $\text{turn}$ 은 0으로 변경된 시점임. 그래서  $P_0$ 가 CS 진입후,  $\text{turn}$ 을 1로 바꿔줘야,  $P_1$ 의 들어갈 수 있는데 그게 되지 안되는 상황임).

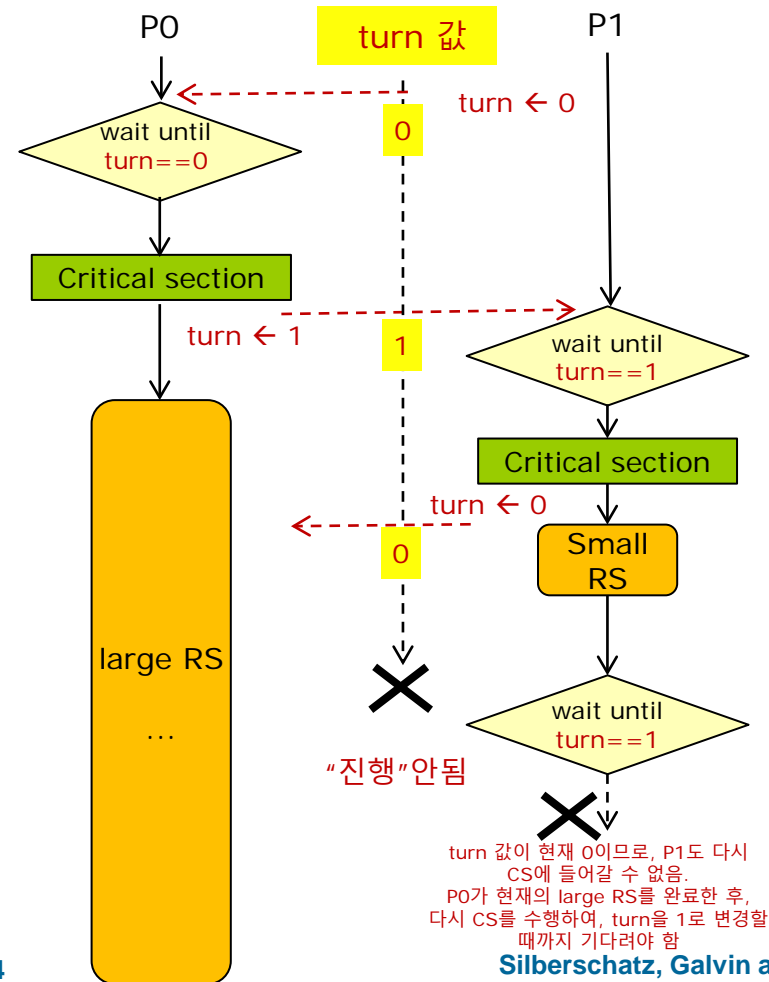
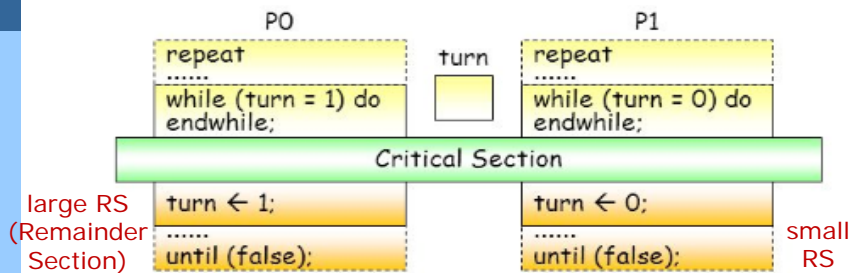
# SW 해결책 - Algorithm 1

□ **Progress requirement is not satisfied : “진행” 요구사항 만족 못시킴**

P0가 CS 진입되어 turn = 1이 되었음. 그러면 P1도 CS에 진입할 수 있게 됨.

P1은 CS 수행후, turn = 0으로 만든 후, 짧은 RS 수행후, 다시 repeat되어 P1의 CS에 들어가려고 하는데, 당연히 P1 진입안됨, turn이 아직 0임.

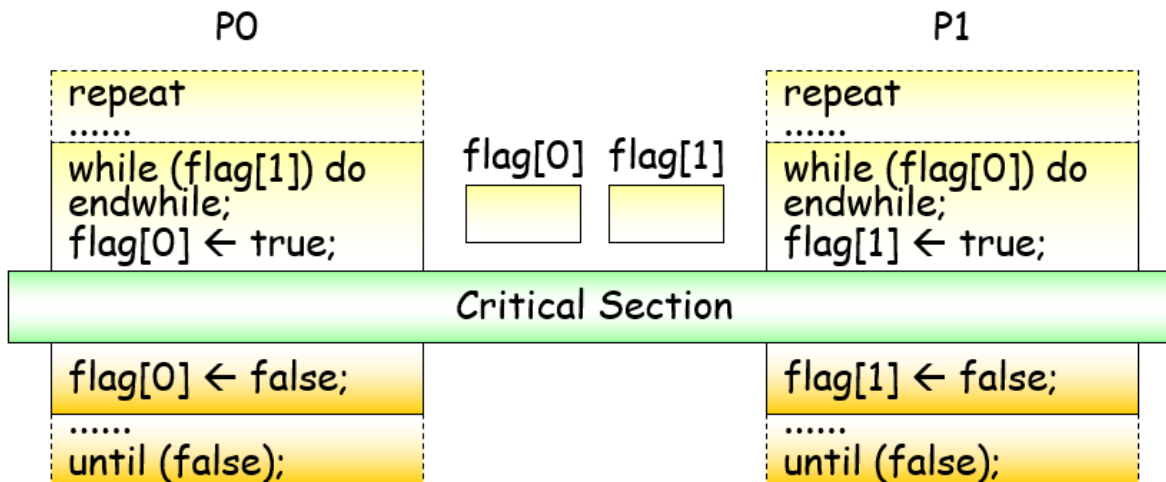
turn이 1이 되기 위해서는 P0가 RS 수행 완료하고 다시 CS로 진입해야 하는데, 현재 P0는 이전 CS 진입후, 아직도 large RS를 수행하는데 머물러 있음. (이때는 P1에 의해 turn은 0으로 변경된 시점임. 그래서 P0가 CS 진입후, turn을 1로 바꿔줘야, P1의 들어갈 수 있는데 그게 되지 안되는 상황임).



# SW 기반 방법 - Algorithm 2

- Keep one BOOL variable for each process: **flag[0]** and **flag[1]**
- $P_i$  signals that it is ready to enter it's CS by: **flag[i]:=true**
- Mutual Exclusion is satisfied **but not the progress requirement**
- If we have the sequence:
  - T0: **flag[0]:=true** → P0가 CS에 들어가고 싶다고 깃발드는 것임
  - T1: **flag[1]:=true**
- Both process will wait forever to enter their CS: we have a **deadlock**

flag[0]과 flag[1]이 모두 true 이면, 두 process는 모두 계속 기다리는 상황 발생 : deadlock !



```

Process Pi:
repeat
    flag[i]:=true;
    while(flag[j]){};
    CS
    flag[i]:=false;
    RS
until FALSE
    
```

# SW 기반 방법 - Algorithm 3, Peterson's Solution

---

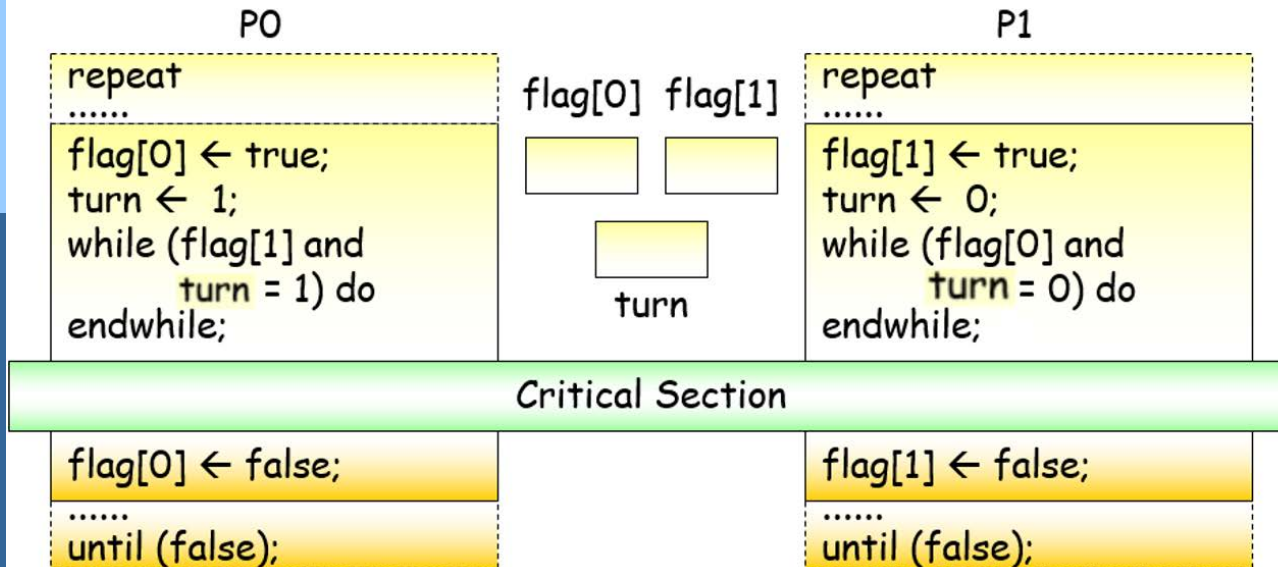
- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable **turn** indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process  $P_i$  is ready!

# Algorithm for Process $P_i$

- Initialization:  $\text{flag}[0] := \text{flag}[1] := \text{false}$ ,  $\text{turn} := 0$  or  $1$  ;
  - $\text{flag}$ 은 해당 Process가 CS 들어갈 준비되어 있음 표현,  $\text{turn}=0$ 은  $P_0$ 가  $\text{turn}=1$ 은  $P_1$ 이 CS에서 실행될 수 있음을 의미.
  - That is, Willingness to enter CS is specified by  $\text{flag}[i] := \text{true}$
- If both processes attempt to enter their CS simultaneously, only one  $\text{turn}$  value will last.
 

결국  $\text{turn}$  값은 0 혹은 1 둘 중 하나임

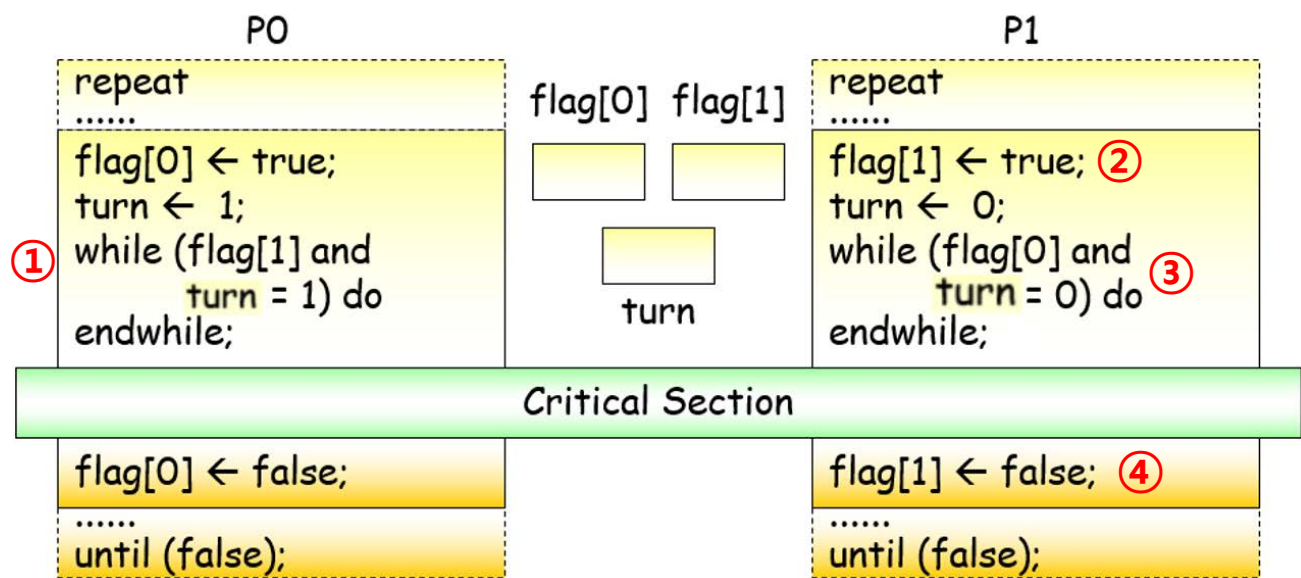
  - CS에 들어가기 위해  $P_0$ 는  $\text{flag}[0] = 1$ 로하고  $\text{turn} = 1$ 로 지정  $\rightarrow$  이는 process 0 자신도 CS에 들어가고 싶지만,  $\text{turn} \leftarrow 1$ 로함으로써, process 1에게 CS에 들어가고 싶으면 들어가라는 의미임
  - $P_1$ 이  $\text{while}()$ 에서 기다리다,  $\text{turn} = 1$ 로되면,  $P_1$ 은 CS로 들어감
- Exit section: specifies that  $P_i$  is unwilling to enter CS



```

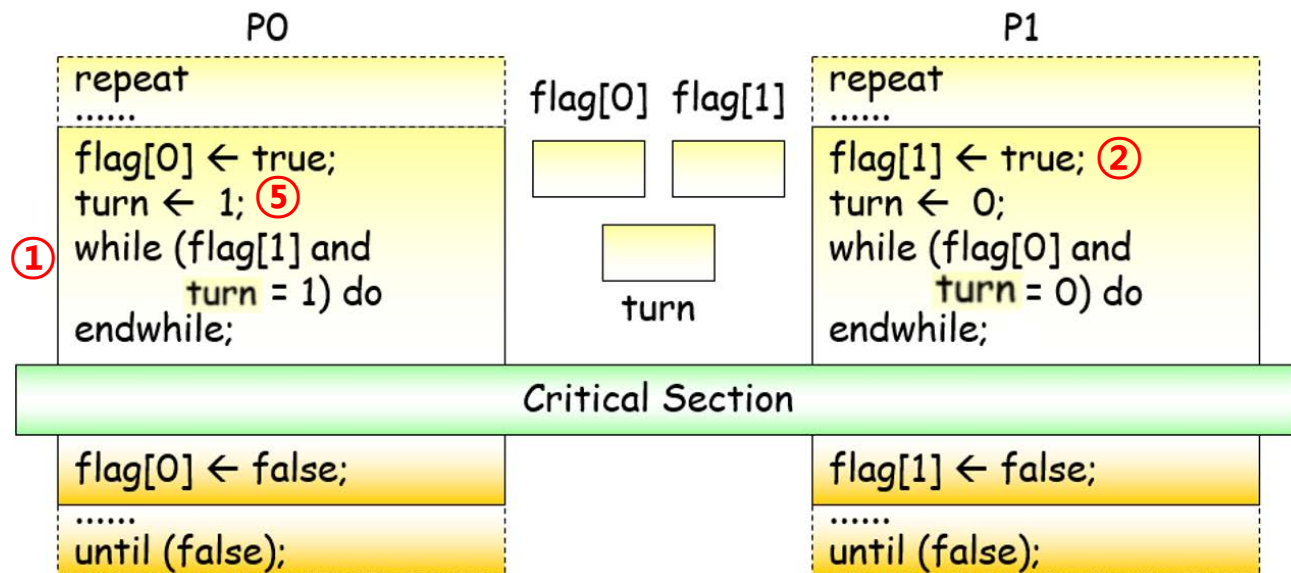
Process  $P_i$ :
repeat
     $\text{flag}[i] := \text{true};$ 
     $\text{turn} := j;$ 
    do {} while
        ( $\text{flag}[j]$  and  $\text{turn} = j$ );
        CS
     $\text{flag}[i] := \text{false};$ 
    RS
until FALSE
    
```

- Mutual exclusion is preserved since:
  - $\text{flag}[0] = \text{flag}[1] = \text{true}$ 는  $P_0$ 와  $P_1$ 이 CS에 들어가고자 하는 의도가 있음을 의미,  $\text{turn} = i$ 이면  $P_i$ 가 CS에 들어감  $\rightarrow$  이에 mutual exclusion 보장됨
- The progress and bounded waiting requirements are satisfied:
  - $P_i$ 의 CS영역 진입 제한은 **while (flag[j]==true && turn==j) 때문 ①**.
    - ▶  $P_i$ 가 CS영역에 진입할 준비가 안되었거나(flag[j]← false) 혹은 j가 들어갈 순서가 아니면(turn != j)  $\rightarrow$  while문의 wait을 벗어나서  $\rightarrow P_i$ 는 CS에 진입할 수 있음
  - $P_j$ 는 자신의 **flag[j]← true**로 하고 CS 진입하고자 기다림(while문) ②③
    - ▶ 이때 turn ==i 이라면  $P_i$ 의 CS 영역으로 진입하고①, turn==j이면,  $P_j$ 의 CS 영역에 진입③
    - ▶ 한편  $P_j$ 는 CS 영역 실행 후, flag[j]← false로 지정하여, 자신은 CS에서 나왔다는 것을 알림④. 이는 곧, 상대방  $P_i$ 가 CS에 진입하도록 만듦①



# Peterson's Solution (Cont.)

- The progress and bounded waiting requirements are satisfied: (계속)
  - $P_i$ 가 자신이 CS에 들어갈 의도가 있다면  $\text{flag}[i] \leftarrow \text{true}$ 로 지정하면 turn 값은  $i$ 로 지정해서 다른 entity  $i$ 에 양보함 ②.
  - $P_i$ 는  $i$  자신이 while 문에서 기다리는 동안 ①, 다른 entity에게 양보한 turn 값  $j$ 를 바꾸지 않으므로(즉 turn 값이  $j$ 로 지정된것 ⑤ 유지),  $P_j$ 는 CS 진입이 보장됨  $\rightarrow P_j$ 가 CS 들어갔다 나오면,  $j$  자신은 다시 들어가지 않는다고 함( $\text{flag}[j] \leftarrow \text{false}$ )
- $\rightarrow$  이에,  $P_i$ 도 한번은 CS 진입이 보장됨(progress 보장) ①.



# Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - **Protecting critical regions via locks**
- Uniprocessors – could disable interrupts
  - **Currently running code would execute without preemption**
  - **Generally too inefficient on multiprocessor systems**
    - ▶ **On a multiprocessor: mutual exclusion is not preserved**
    - ▶ **Generally not an acceptable solution**
    - ▶ **Operating systems using this not broadly scalable**
- Modern machines provide **special atomic hardware instructions**
  - ▶ **Atomic = non-interruptible**
  - **Atomic 명령어 사례**
    - ▶ **test memory word and set value**
    - ▶ **swap contents of two memory words**

# Solution to Critical-section Problem Using Locks

---

```
do {  
    acquire lock  
        critical section  
    release lock  
        remainder section  
} while (TRUE);
```

# test\_and\_set Instruction

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

## test\_and\_set:

- **test** : target 주소에 있는 값을 test 확인함
- **set** : target 주소에 해당하는 값을 true로 setting 함
- Atomic하게 동작함

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

□ Shared Boolean variable lock, initialized to FALSE

```
□ do {
    while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
} while (true);
```

lock은 process간 공유된 값이며, 초기에는 false임  
lock 값을 test하니 false이며, 이를 set하여 true로 만들 → CS에 진입

CS를 떠날때 다시 이를 false로 setting하여, 다른 process가 CS에 진입할 수 있도록 해줌

# compare\_and\_swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

## compare\_and\_swap:

- **compare** : 해당 주소의 값(\*value)이 expected 기대값과 같으면,
- **swap** : 새로운 값으로 바꿈
- Atomic하게 동작함

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new\_value” **but only if “value” == “expected”**. That is, the swap takes place only under this condition.

❑ Shared integer “lock” initialized to 0;

❑ Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```

- lock의 value값이 1이면, 계속 기다림.
- lock값이 expected 값(0)과 같으면, new\_value(1)로 바꿈

이들은 한정된 대기조건 만족못시킴!

# Bounded-waiting Mutual Exclusion with test\_and\_set

- boolean waiting[n];
- **boolean lock;**
- // 모두 false로 초기화됨

```
do {  
    waiting[i] = true;  
    key = true;  
    while (waiting[i] && key) key = test_and_set(&lock);  
    waiting[i] = false;  
    /* critical section */  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = false;  
    else  
        waiting[j] = false;  
    /* remainder section */  
} while (true);
```

- $P_i$ 가 waiting하고 있으며, key가 true인 경우, CS진입위해 test\_and\_set check함
- lock이 false 값이면(다른 process에 의해 lock되어 있지 않으면), key가 false가 되어 CS에 진입됨

- CS 영역 진입하면  $P_i$ 는 waiting하지 않는다고 알림

CS에서 나오면서 다른  $P_j$  상태를  $i+1, \dots, i-1$ 까지 순환하면서, check함. waiting하는  $j$ 가 있으면, 해당  $j$ 를  $waiting[j] \leftarrow false$ 로 하여, 해당  $P_j$ 가 CS에 진입가능하게 함.  
자신일 경우에는 lock을 해제함(CS 실행했기때문에, 다른  $P_j$ 가 CS 진입가능하도록)

한정된 대기조건 만족시킴

# Mutex Locks

---

- ❑ Previous solutions are **complicated** and generally **inaccessible to application programmers**
- ❑ OS designers build software tools to solve critical section problem
- ❑ **Simplest is mutex lock**
- ❑ Protect a critical section by first **acquire()** a lock then **release()** the lock
  - ❑ Boolean variable indicating if lock is available or not
- ❑ Calls to **acquire()** and **release()** must be atomic
  - ❑ Usually implemented via hardware atomic instructions
- ❑ But this solution requires **busy waiting**
  - ❑ This lock therefore called a **spinlock**

# acquire() and release()

```
□ acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}  
  
□ release() {  
    available = true;  
}  
  
□ do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

- acquire() : lock을 획득함
- release() : lock을 반환함
- mutex lock은 available 변수를 가짐 → available 변수: lock의 가용 여부를 표시함
- lock을 사용할 수 있으면, acquire() 호출은 성공하고 lock은 곧 사용불가 상태로 바뀜
- 단점 : busy waiting, consumes CPU power !

# Semaphore

- sema : 그리스어로 sign을 의미
- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

- **wait()** and **signal()**

- ▶ Originally called **P()** and **V()**

\* 네델란드어  
P: Proberen(검사하다)  
V: Verhogen(증가하다)

- Definition of the **wait()** operation

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

atomic operation  
(interrupt 없음)

- Definition of the **signal()** operation

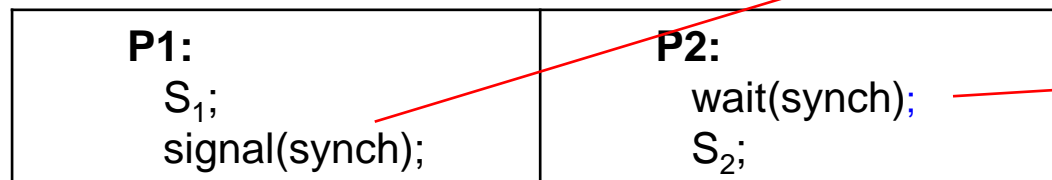
```
signal(S) {  
    S++;  
}
```

atomic operation

# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**, 즉, 상호배제 목적으로만 사용가능
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$
- 가정:
  - P1은 S1 명령문을 P2는 S2 명령문을 수행하는데 P1과 P2는 병행 수행됨
  - 하지만, S1다음에 S2 실행되어야 함
  - P1과 P2는 semaphore “synch”를 공유함

Create a semaphore “**synch**” initialized to 0



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

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

- Process P1은 S1을 무조건 실행할 것임
- S1 실행 후에 signal(synch) 실행되면, synch 값은 증가됨

- Process P2는 wait(synch)에 도달함. → P1의 S1 실행 후, signal(synch) 실행 전까지는 wait( )를 벗어날 수 없음.
- (synch 값이 0이므로)
- S1 다음에 S2가 실행되는 것이 보장됨

- Can implement a counting semaphore **S** as a binary semaphore

# Semaphore Implementation

---

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
  - 결국 **busy waiting이 발생함 !!**
    - ▶ But implementation code is short
    - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

# Semaphore Implementation with no Busy waiting

- 기존 busy waiting하는 semaphore 문제를 해결하기 위해,
  - wait() 연산 실행 후, semaphore 값이 양수가 아니면, 프로세스를 waiting queue에 넣고 해당 프로세스는 waiting state으로 감 !! → scheduler는 다른 프로세스를 선택 실행.
  - Waiting queue에 대기하는 프로세스는 향후, signal()의 wakeup()에 의해 waiting state을 ready state으로 변경 → Ready queue에서 scheduling됨
- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- ```
typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```

  - list → semaphore를 기다리는 process들

# Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
```

```
    S->value--;
```

```
    if (S->value < 0) {
```

```
        add this process to S->list;
```

```
        block(); // 자신을 호출한 프로세스를 중지시킴
```

```
    }
```

```
}
```

- 결국 value 값의 절대치는 semaphore를 대기하는 프로세스의 수

- Process가 semaphore 를 기다려야 한다면, 이 프로세스는 semaphore의 process list에 추가됨
- 그리고 해당 프로세스는 waiting state로 감

```
signal(semaphore *S) {
```

```
    S->value++;
```

```
    if (S->value <= 0) {
```

```
        remove a process P from S->list;
```

```
        wakeup(P); // 봉쇄된 프로세스 P의 실행을 재개함
```

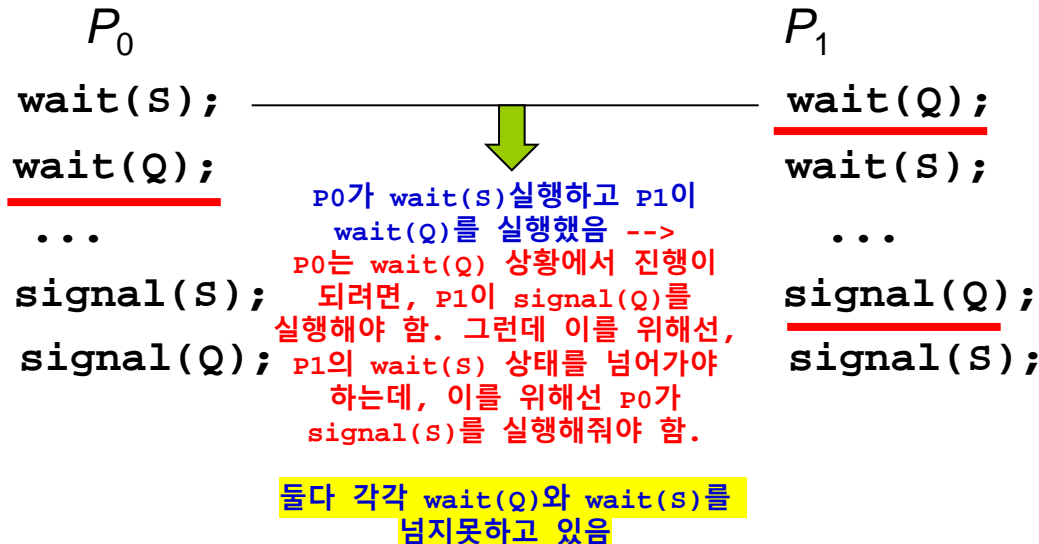
```
    }
```

```
}
```

- List에는 semaphore 를 기다리는 프로세스들이 있음
- Signal은 process list에서 하나의 프로세스를 꺼내서, 이를 깨움.

# Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $S$  and  $Q$  be two semaphores initialized to 1



- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion(우선순위 역전문제)** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol (우선순위 상속 프로토콜)**

# 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  $n$

# Semaphores

1

Mutex

0

Full

n

Empty

- **mutex**: buffer pool 접근위한 상호 배제 기능 제공 (초기 1)
- **Empty**: 비어있는 각 buffer의 수 (초기 n개의 buffer가 empty)
- **Full**: 채워진 각 buffer의 수 (초기 0개의 buffer가 full)

```
while (true) {  
    // produce an item  
  
    wait (empty);  
    wait (mutex);  
  
    // buffer pool에 item 쓰기  
  
    signal (mutex);  
    signal (full);  
}
```

Producer

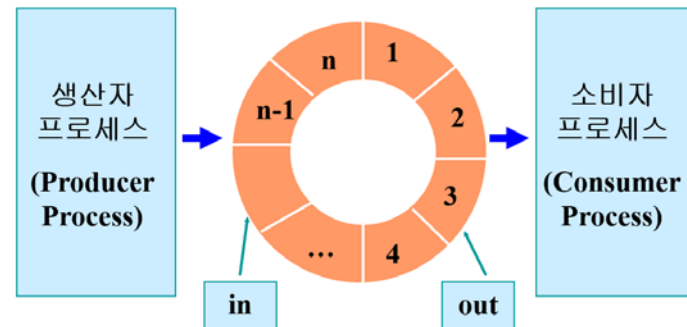
```
while (true) {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
}
```

Consumer

- **Consumer 동작**: 채워진 buffer가 있음(읽을 item 있음) → 상호배제(mutex) 상태에서 공유buffer 배타적 읽음 → 읽은 후, empty signal 보냄

- **producer 동작**: 써야할 item 있음 → empty한 buffer slot 있을 때까지 기다림( wait(empty)) & 쓸 때는 상호배제 필요. Buffer pool에 item 씬

- n개의 버퍼로 구성됨
- 버퍼 하나씩 채워지거나(full), 비워짐(empty)



# 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 : if a writer and some other process (either a reader or a writer) access the database simultaneously, **chaos may occurs**
  - **Solution:** writers **should have exclusive access to the shared database** while writing to the database. (Writer가 쓰기작업할 동안에는 공유 데이터에 대한 배타적 접근권한 가지게 할 필요 있음)

□

# Readers-Writers Problem

- Several variations of how readers and writers are considered
  - (1) first readers-writers problem:
    - ▶ requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting
    - ▶ Writer가 공유자원을 이미 사용하고 있지 않는 한, reader들은 계속 wait할 필요없이 자원을 잡을 수 있음
    - ▶ First readers 의미 → Reader가 자원 접근할 수 있도록 함
    - ▶ 이 기법의 문제는 Writer가 기아에 빠질 수 있음
  - (2) Second readers-writes problem:
    - ▶ once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
    - ▶ Writer가 우선임 !!
    - ▶ 이 기법의 문제는 reader가 starvation할 수 있음

# Readers-Writers Problem (Cont.)

- 1<sup>st</sup> Readers-Writers Problem (Writer가 기아에 빠질 수 있는 문제)에 대한 해결책 제시
  - Writer가 공유자원을 잡지못했다면, 기다리는 Reader에게 계속 자원을 잡을 수 있도록 함
  - Semaphore **rw\_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read\_count** initialized to 0
    - **rw\_mutex**: reader와 writer가 공유하는 semaphore
    - **mutex**: read\_count 갱신시 상호 배제 보장용 semaphore
      - **read\_count**: 현재 몇 개의 프로세서들이 이 객체를 읽는지 알려줌

- **rw\_mutex**: reader와 writer가 공유 (즉, 이는 reader와 write간 경쟁을 의미함)
- **Mutex**: **read\_count** 갱신시 상호 배제 보장용
  - **Read\_count**: 현재 몇 개의 프로세스들이 객체 읽는지 알려줌

## □ writer process

```
do {
    wait(rw_mutex);

    ...
    /* writing is performed */
    ...

    signal(rw_mutex);
} while (true);
```

## □ reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);

    signal(mutex);

    ...
    /* reading is performed */
    ...

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

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

- W1과 N개의 Reader가 있는 상황에서, 예를 들어, W1 → R1 → R2 → R3 → R1, R3, R2 (예) ... 가 발생할 수도 있어, W1이 starvation 되는 상황 방지해야 함
- 참고: "Concurrent control with Readers and Writers by Courtois, Pamas, Communications of the ACM, 1971."
- 이를 위해, writer 와 하나의 첫번째 진입 Reader가 **rw\_mutex semaphore**를 공유. 진입 Reader 가 **rw\_mutex** 사용시, 이때부터 **rw\_mutex** 값은 1 → 0으로 됨(**wait(rw\_mutex)**).
- 즉, 나머지 N-1개의 Readers는 W1과 경쟁하지 못하게 하여(즉, **rw\_mutex**을 기다리지 않도록 하여), **W1이 starvation 되는 상황 없앴**
- N-1개의 Readers들 중에서의 Reading 순서 선택은 scheduler에 의함
- 이를 통해, 결국, Readers 들이 모두 reading 하면, **rw\_muxtex** 값을 1로 만들어 (**signal(rw\_mutex)**) 다시 W1에 Writing 순서 넘겨주는 셈

# Readers-Writers Problem Variations

- Starvation problem is solved on some systems by providing **reader-writer locks**
- The readers–writers problem and its solutions have been generalized to provide **reader–writer locks** on some systems.
- Acquiring a reader–writer lock requires specifying the mode of the lock: **either read or write access**.
  - When a process wishes only to **read** shared data, it requests the **lock in read mode**.
  - A process wishing to **modify** the shared data must request the **lock in write mode**.
- Multiple processes are permitted to concurrently acquire **a lock in read mode**, but only one process may acquire **the lock for writing**, as exclusive access is required for writers.
- **Reader–writer locks are most useful** in the following situations:
  - shared data를 읽기만 하는 프로세스와 쓰기만 하는 프로세스를 식별하기 쉬운 응용
  - Write보다 reader 개수가 많은 응용 (← This is because reader– writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock. )

# Dining-Philosophers Problem



- ❑ Philosophers spend their lives alternating thinking and eating
- ❑ When a philosopher thinks, she does not interact with her colleagues.
- ❑ From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors)
- ❑ A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.
- ❑ In the case of 5 philosophers
  - ❑ Shared data
    - ▶ Bowl of rice (data set)
    - ▶ 5 Semaphores : chopstick [5] initialized to 1

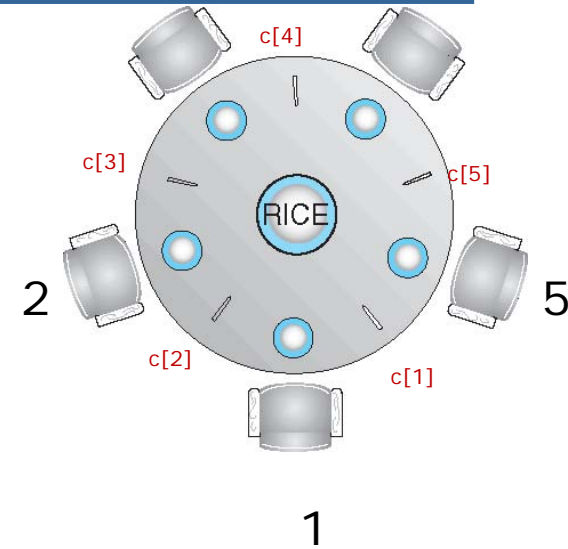
# Dining-Philosophers Problem Algorithm

□ The structure of Philosopher  $i$ :

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

□ What is the problem with this algorithm?

- 인접한 두 철학자가 동시에 식사하지 않음을 보장하지만, deadlock 발생
- 5명의 philosopher가 동시에 배가 고픈 경우:
  - 자신의 left chopstick을 먼저 잡으면, 해당 chopstick semaphore는 모두 0이 되어, 각 philosopher는 오른쪽 chopstick을 잡기 위해 영원히 기다려야 함 (deadlock)



# Dining-Philosophers Problem Algorithm (Cont.)

---

## □ Deadlock handling

- Allow at most 4 philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up chopsticks only if both are available (picking must be done in a critical section).
- Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

# Problems with Semaphores

- Semaphore는 상호배제 보장과 프로세스간 협력을 위해 사용되는 강력하고 유연한 기법임
- 하지만, semaphore의 사용은 쉽지 않음.
  - Semaphore의 연산인 wait과 signal이 프로그램 전체에 산재 되어 있어, 이런 연산들이 전체적으로 수행에 어떤 영향을 미치는지 파악 어려움 → (다른 기법 제안됨: Monitor 기법)
- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) ... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

# Monitors

- Monitor는 프로그래밍 언어 수준에서 제공되는 구성체 (construct) 기술임
- Semaphore와 동일한 기능을 제공하며, 보다 사용하기 쉬움
  - Monitor는 Concurrent Pascal, Java 등, 다양한 프로그램 언어에 구현되어 있음
  - 특정 언어 수준이 아닌, library 수준으로 구현도 가능함
- 프로그래머는 자신이 보호하려는 어떠한 객체도 monitor lock을 설정할 수 있음
  - 예를 들어, 리스트 구조의 경우, 프로그래머는 리스트 구조 전체에 하나의 monitor lock을 설정할 수 있음.
  - 혹은, 각 리스트마다 monitor lock 설정 가능
  - 각 리스트에 존재하는 각 원소마다 monitor lock 설정도 가능

# Monitors - 상호배제기능

- Monitor는 Signal 기반이 Monitor 기법
- Monitor는 지역변수, 초기화 루틴, 프로시저로 구성됨
- **Monitor의 주요 특징**
  1. 지역변수는 monitor의 프로시저를 통해 접근 가능함. 즉, 외부에서 변수에 대한 직접 접근 허용 안됨
  2. 프로세서는 monitor 의 프로시저 중에 하나를 호출함으로써 monitor로 들어감
  3. 한 순간에 오직 하나의 process 만이 monitor 내에 존재함. 즉, monitor가 이미 사용 중일 경우, 다른 process 들은 모니터가 이용가능해질 때까지 대기함
- 1과 2는 객체 개념과 유사함
- 3번 : 즉, 한 순간에 하나의 process 만이 수행한다는 원칙을 지킴으로써, **monitor는 상호배제 기능을 제공하게 됨**
- 즉, Monitor 내부의 변수는 한 순간에 하나의 process 만 접근할 수 있음
  - → 다수의 process들에 의해 공유되는 data를 monitor 내부에 위치시키면?  
**상호 배제 보장 받을 수 있음**

# Monitors – 동기화 방법

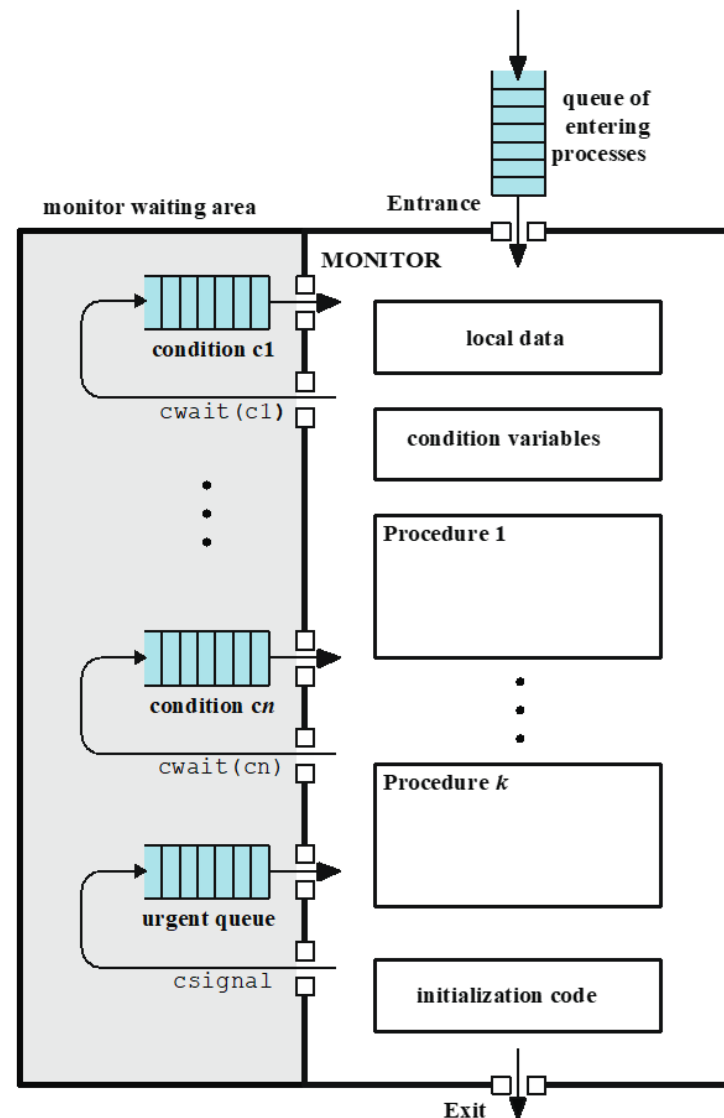
- Monitor가 멀티프로세스의 병행처리에 유용하게 사용되려면, “**동기화 방법**”도 제공해야 함
  - 예를 들어, 한 process가 monitor에 진입하여 수행하는 도중에, 특정 조건이 만족될 때까지 blocked 될 필요가 생겼다고 가정하자.
  - 이때, 이 process는 block되는 것 뿐만 아니라, monitor에 대한 독점권도 반납하여, 다른 process가 monitor에 진입할 수 있도록 허용해야 함
  - 그런 다음, 기다리던 조건이 만족되고 monitor도 사용 가능한 상태가 되면, block된 process는 다시 깨어나 monitor 내부로 진입하여, 자신의 중지되었던 위치에서부터 다시 수행을 재개할 수 있어야 함
- Monitor는 동기화를 위해 조건변수(condition variable)을 제공함
  - 조건변수는 monitor 내부에 포함되며, monitor 내부에서만 접근가능
  - 조건변수는 monitor에 사용되는 특수한 유형의 데이터타입이며, cwait(c), csignal(c) 인터페이스에 의해 접근됨 (c: 조건)

# Monitors – cwait, csignal

- Monitor이 조건 변수를 접근하는 interface
- cwait ( c )
  - 호출한 process를 조건 c에서 일시 중지 시킴
  - Monitor는 다른 process에 의해 사용될 수 있음
- csignal( c )
  - cwait ( c )에 의해 중지되었던 process의 수행을 다시 재기 시킴
  - 만일 중지된 process가 여러 개일 경우? → 하나만 선택
  - 만일 중지된 process가 없다면? → 아무것도 하지 않음 ☺
- Monitor의 wait, signal은 semaphore의 wait, signal과 다른 특성 가짐:  
monitor에서는 signal이 발생했을 때, 이를 기다리는 process 가 없으면,  
해당 signal을 잃어버림

# Monitors – 하나의 entry point를 갖는 monitor

- Monitor에는 어떤 시점에 하나의 process만 진입할 수 있음 → **하나의 entry point 갖고 있음**
- Monitor가 이미 사용 중이면, monitor를 사용하려는 다른 process 들은 가용해질 때까지 진입하려는 monitor 들의 큐에서 블록됨
- 한편, monitor 내부에서 수행중인 process 가 조건 x에서 잠시 대기할 필요가 있으면, cwait (x)를 호출함
  - 이때, 해당 process 는 조건 x와 연관된 대기큐에서 블록됨
  - 그런 다음, 조건이 변하면 블록된 process 는 깨어나 monitor 에 재진입하게 되고 cwait (x) 이후의 위치에서 수행을 재개함
  - Monitor 내부에서 수행중인 process가 조건 x의 변화를 발견하면, csignal (x) 를 호출함
  - 이 호출이 바로 조건 x와 연관된 대기 큐에서 블록된 process를 깨우는 셈이 됨



# Monitors

- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
}
```

# Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure



Process enters monitor by invoking one of its procedures



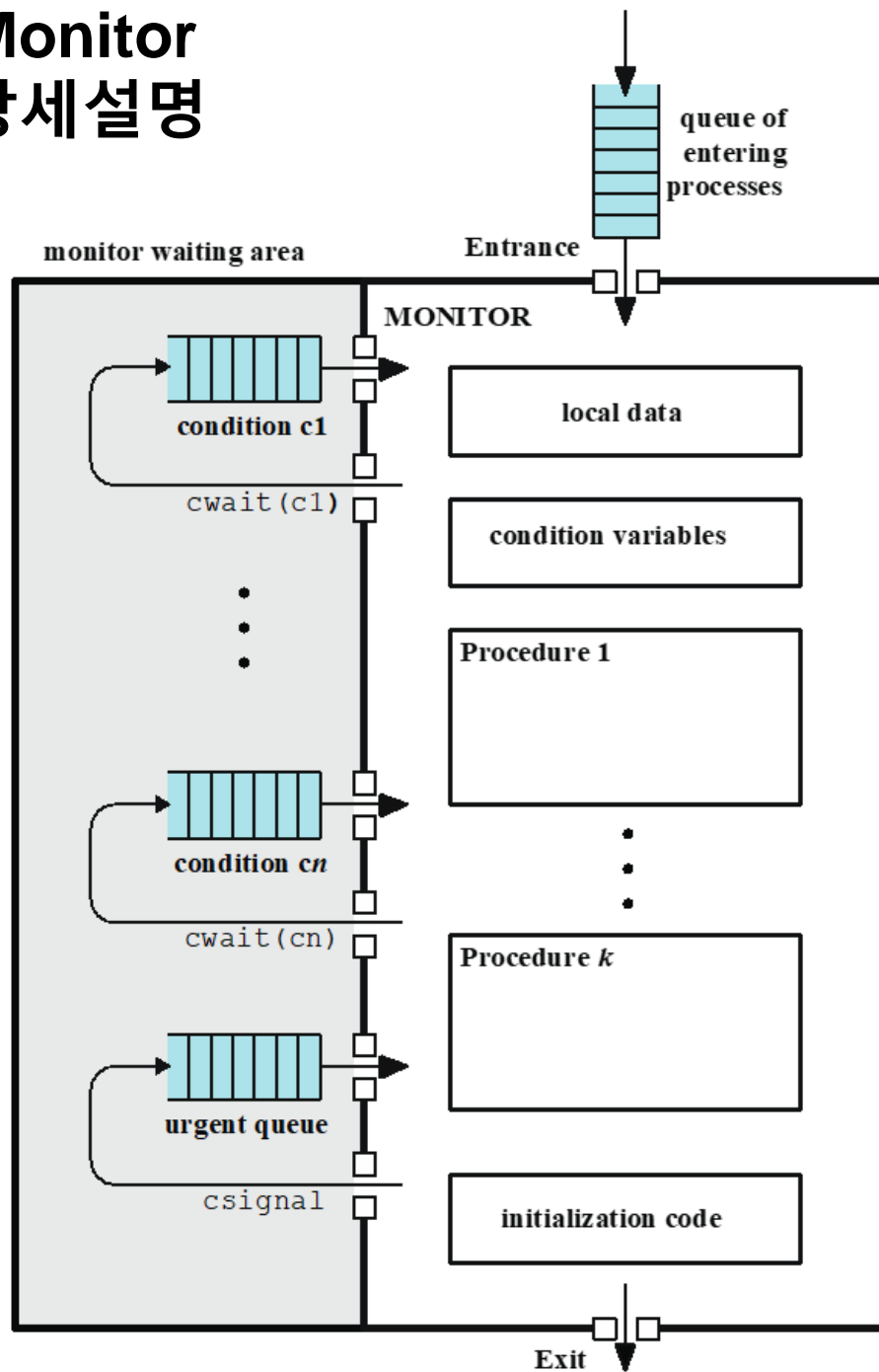
Only one process may be executing in the monitor at a time

# Monitor Synchronization with cwait, csignal

---

- Achieved by the use of **condition variables** that are **contained within the monitor** and accessible only within the monitor
  - Condition variables are operated on by two functions:
    - ▶ **cwait(c)**: suspend execution of the calling process on condition c
    - ▶ **csignal(c)**: resume execution of some process blocked after a cwait on the same condition

# Monitor 상세설명



- 모니터에는 특정 시점에 하나의 프로세스만 진입할 수 있음
- 모니터가 한 프로세스에 의해 사용 중이면, 다른 프로세스는 "queue of entering process"에 queue됨
- 모니터 내부에서 수행 중이던 프로세스가 조건 c1에서 잠시 대기할 필요있으면, **cwait(c1)**을 호출함 → c1 queue에서 block됨
- 조건이 변하면 blocked된 프로세스는 깨어나서 monitor에 재진입하고 **cwait(c1)** 코드 이후부터 수행
- 모니터 내부에서 수행 중이던 프로세스가 조건 c1의 변화를 발견하면, **csignal(c1)**을 호출함
- 이는 c1 queue에서 대기중이던 프로세스를 깨움
- urgent queue는 monitor에서 수행되던 프로세스가 블록될 경우, 새로 진입하는 프로세스(queue of entering processes)보다 더 높은 우선 순위를 갖고 다시 실행되기 위해 사용 가능

# A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```

/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                      /* space for N items */
int nextin, nextout;                  /* buffer pointers */
int count;                          /* number of items in buffer */
cond notfull, notempty;              /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull);    /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                 /* resume any waiting consumer */
}

void take (char x)
{
    if (count == 0) cwait(notempty);    /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* one fewer item in buffer */
    csignal(notfull);                 /* resume any waiting producer */
}

/* monitor body */
{
    nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}
    
```

```

void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void main()
{
    parbegin (producer, consumer);
}
    
```

- Buffer 접근 제어를 위해 두개의 조건 변수(cond data type)로 notfull과 notempty 사용
- **notfull**: buffer에 하나이상의 데이터를 저장할 공간 남아 있는 경우→true
- **notempty**: buffer에 읽을 데이터가 있는 경우→true

(1) Producer는 직접 buffer 공간 접근할 수 없으며, 오직 monitor의 append( )를 호출하여, 데이터를 buffer에 추가

- (2) notfull 조건을 우선 확인함. 진행. 만약 Full이면 이 조건변수에서 일시중지함. 즉, cwait함
- 이때 다른 프로세스는 monitor에 들어갈 수 있음
  - 그 후, consume가 buffer 에서 data를 꺼내어, 더 이상 full이 아닐 경우, producer는 일시중지 상태에서 깨어나, 중지된 위치에서부터 다시 수행을 재개함
  - 결국 producer는 buffer에 문자를 추가함. 그리고 nonempty 조건변수에 대한 signal 연산을 수행함

(3) consumer는 buffer가 notempty일때 읽음. 읽은 후에 waiting 중인 producer가 받을 수 있는 signal(notfull)을 발생함. waiting 중인 Producer가 있으면 buffer가 notfull 이라는 signal을 받았으므로 새로운 데이터를 쓸 수 있게 됨

```

void append (char x)
{
    while(count == N) cwait(notfull);    /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                             /* one more item in buffer */
    cnotify(notempty);                   /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                             /* one fewer item in buffer */
    cnotify(notfull);                   /* notify any waiting producer */
}

```

**Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor**

# 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**

# Monitor Solution to Dining Philosophers

---

## □ 기초

- **THINKING** – When philosopher doesn't want to gain access to either fork.
- **HUNGRY** – When philosopher wants to enter the critical section.
- **EATING** – When philosopher has got both the forks, i.e., he has entered the section.
  
- Philosopher  $i$  can set the variable  $state[i] = EATING$  only if her two neighbors are not eating  
( $state[(i+4) \% 5] \neq EATING$ ) and ( $state[(i+1) \% 5] \neq EATING$ ).

# Monitor Solution to Dining Philosophers

## // Dining-Philosophers Solution Using Monitors

### monitor DP

```
{
    status state[5];
    condition self[5];
    // Pickup chopsticks
    Pickup(int i)
    {
        // indicate that I'm hungry
        state[i] = hungry;
        // set state to eating in test()
        // only if my left and right neighbors
        // are not eating
        test(i);
        // if unable to eat, wait to be signaled
        if (state[i] != eating) self[i].wait;
    }
    // Put down chopsticks
    Putdown(int i)
    {
        // indicate that I'm thinking
        state[i] = thinking;
        // if right neighbor R=(i+1)%5 is hungry and
        // both of R's neighbors are not eating,
        // set R's state to eating and wake it up by
        // signaling R's CV
        test((i + 1) % 5);
        test((i + 4) % 5);
    }
}
```

### test(int i)

```
{
    if (state[(i + 1) % 5] != eating
        && state[(i + 4) % 5] != eating
        && state[i] == hungry) {
        // indicate that I'm eating
        state[i] = eating;
        // signal() has no effect during Pickup(),
        // but is important to wake up waiting
        // hungry philosophers during Putdown()
        self[i].signal();
    }
}
```

### init()

```
{
    // Execution of Pickup(), Putdown() and test()
    // are all mutually exclusive,
    // i.e. only one at a time can be executing

    for i = 0 to 4
        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
}
```

} // end of monitor

# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
```

```
{
    enum { THINKING; HUNGRY, EATING } state [5];
    condition self [5];

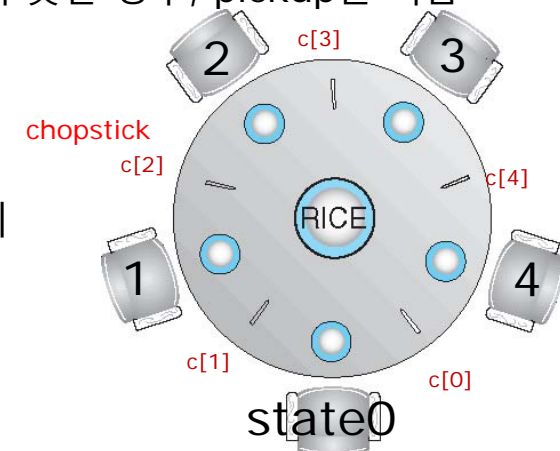
    void pickup (int i) {
        state[i] = HUNGRY;
        (1) test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        (4) // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```

```
void test (int i) {
    (2) 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;
}
```

- (1) Philosopher는 양쪽 chopstick을 모두 얻을 수 있을 때만 chopstick을 pickup하게 됨
- (2) philosopher  $i(i=0)$ 는 그의 양쪽 두 이웃이 EATING 이 아닐때만  $state[i] \leftarrow EATING$ 으로 설정 가능  
( $state[(i + 4) \% 5] \neq EATING$ ) && ( $state[(i + 1) \% 5] \neq EATING$ )
- (3) Philosopher  $i$ 는 배고프지만 자신이 원하는 chopstick을 pickup하지 못한 경우, pickup을 미룸
- (4) Philosopher ( $i=0$ )가 식사후 chopstick putdown시, 양쪽옆 philosopher(4,1)의 상황을 본 후 (2), EATING 할 수 있도록 함
  - Philosopher 4가 EATING이 되기위해선, 3과 0의 state이 EATING이 아니고 4가 HUNGRY여야 함.
  - Philosopher 1이 EATING위해선, 0과 2가 EATING이 아니어야 함



# Synchronization Examples

---

- Solaris
- Windows
- Linux
- Pthreads

# Solaris Synchronization

- ❑ Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- ❑ Uses **adaptive mutexes** for efficiency when protecting data from **short code segments**
  - ❑ Starts as a standard semaphore spin-lock
  - ❑ If the data locked(i.e., already in use) by a thread running on another CPU, the thread **spins** while waiting for the lock to be available.
  - ❑ If the lock is held by non-run-state thread, **block and sleep waiting**
- ❑ Uses **condition variables & semaphore** for long code segments
- ❑ Uses **readers-writers locks** are used **to protect data that are accessed frequently** but are usually accessed in a read-only manner
- ❑ Uses **turnstiles to order the list of threads** waiting to acquire either an adaptive mutex or reader-writer lock
  - ❑ turnstile은 blocked thread를 위한 queue 구조
  - ❑ Turnstiles are per-lock-holding-thread, not per-object

# 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
- **For thread synchronization outside the kernel (i.e, user land), Windows provides dispatcher objects**
  - Using a dispatcher object, threads synchronize according to **several different mechanisms, including mutex locks, semaphores, events, and timers**
  - **Events**
    - ▶ An event acts much like a condition variable
  - **Timers** notify one or more thread when time expired

# 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

# Alternative Approaches

---

- Transactional Memory
- OpenMP
- Functional Programming Languages

# Transactional Memory (TM)

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.
  - 한 Transaction의 모든 연산이 완료되면, memory transaction은 commit(확정됨)
  - 그렇지 않으면, 그 시점까지 수행된 모든 연산은 취소되고 transaction 시작 이전 상태로 roll-back함

```
void update ()
{
    acquire();

    /* modify shared data */

    release();
}
```

<mutex 혹은 semaphore로  
구현되는 공유데이터 update 함수>



```
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```

<트랜잭션 메모리의 장점을 활용할 수 있는  
코드 construct: atomic {S} >

- TM synchronizes memory accesses **so that each transaction seems to execute sequentially and in isolation**
- **Software Transaction Memory(STM)**: logical한 atomic 코드 블록
- **Hardware Transaction Memory(HTM)**: cache 계층구조와 coherency 유지 프로토콜 기반으로 transactional memory 구현

# OpenMP(Open Multi-Processing)

- OpenMP is a set of compiler directives and API that support **parallel programming**.

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

- The code contained within the `#pragma omp critical` directive is treated as a **critical section and performed atomically**.
  - The critical directive specifies a region of code that must be executed by only one thread at a time
  - If a thread is currently executing inside a critical region and another thread reaches that critical region and attempts to execute it, it will block until the first thread exits that critical region

# Functional Programming Languages

---

- Functional programming languages offer a different paradigm than procedural languages **in that they do not maintain state.**
- Variables are treated as **immutable (불변)** and **cannot change state once they have been assigned a value.**
  - Relies on immutable data structures and pure calculations to derive new data from existing data hence **eliminating the race condition.**
- There is increasing interest in functional languages such as Erlang and Scala **for their approach in handling data races.**

[https://www.reddit.com/r/functionalprogramming/comments/7z6lz2/why\\_does\\_functional\\_mean\\_no\\_race\\_conditions/](https://www.reddit.com/r/functionalprogramming/comments/7z6lz2/why_does_functional_mean_no_race_conditions/)



# End of Chapter 6

## Q&A

<http://pages.cs.wisc.edu/~remzi/OSTEP/>

참고 문헌:

<http://pages.cs.wisc.edu/~remzi/OSTEP/>

## Operating Systems: Three Easy Pieces

[Remzi H. Arpaci-Dusseau](#) and [Andrea C. Arpaci-Dusseau](#)

Blog: [Why Textbooks Should Be Free](#)

Quick: [Free Book Chapters](#) - [Hardcover](#) - [Softcover \(Lulu\)](#) - [Softcover \(Amazon\)](#) - [Buy PDF](#) - [EU \(Lulu\)](#) - [Buy in India](#) - [Buy T-shirt](#) - [Donate](#) - [For Teachers](#) - [Homework](#) - [Projects](#) - [News](#) - [Acknowledgements](#) - [Other Books](#)

COMING SOON: [Computer Systems: Three Easy Steps](#) --- ALSO COMING SOON: [Distributed Systems: Three Easy Steps](#)

Welcome to **Operating Systems: Three Easy Pieces** (now **version 1.00** -- see [book news](#) for details), a free online operating systems book! The book is centered around three conceptual pieces that are fundamental to operating systems: **virtualization**, **concurrency**, and **persistence**. In understanding the conceptual, you will also learn the practical, including how an operating system does things like schedule the CPU, manage memory, and store files persistently. Lots of fun stuff!

This book **is and will always be free** in PDF form, as seen below. For those of you wishing to **BUY** a copy, please consider the following:

- [Lulu Hardcover \(v1.00\)](#): this may be the best printed form of the book (it really looks pretty good), but it is also the most expensive way to obtain *the black book* of operating systems (a.k.a. *the comet book* or *the asteroid book* according to students). Now just: **\$38.00**
- [Lulu Softcover \(v1.00\)](#): this way is pretty great too, if you like to read printed material but want to save a few bucks. Now just: **\$22.00**
- [Amazon Softcover \(v1.00\)](#): Same book as softcover above, but printed through Amazon CreateSpace. Now just: **\$25.90** (but works with Prime shipping)
- [Downloadable PDF \(v1.00\)](#): this is a nice convenience and adds things like a hyperlinked table of contents, index of terms, lists of hints, tips, systems advice, and a few other things not seen in the free version, all in one massive DRM-free PDF. Once purchased, you will always be able to get the latest version. Just: **\$10.00**
- [Kindle](#): Really, just the PDF and does not include all the bells and whistles common in e-pub books.



# Terminology

|                         |                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>atomic operation</b> | A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes. |
| <b>critical section</b> | A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.                                                                                                                                                                                                                     |
| <b>deadlock</b>         | A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.                                                                                                                                                                                                                                                            |
| <b>livelock</b>         | A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.                                                                                                                                                                                                                                 |
| <b>mutual exclusion</b> | The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.                                                                                                                                                                                                  |
| <b>race condition</b>   | A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.                                                                                                                                                                                                                               |
| <b>starvation</b>       | A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.                                                                                                                                                                                                                                                   |

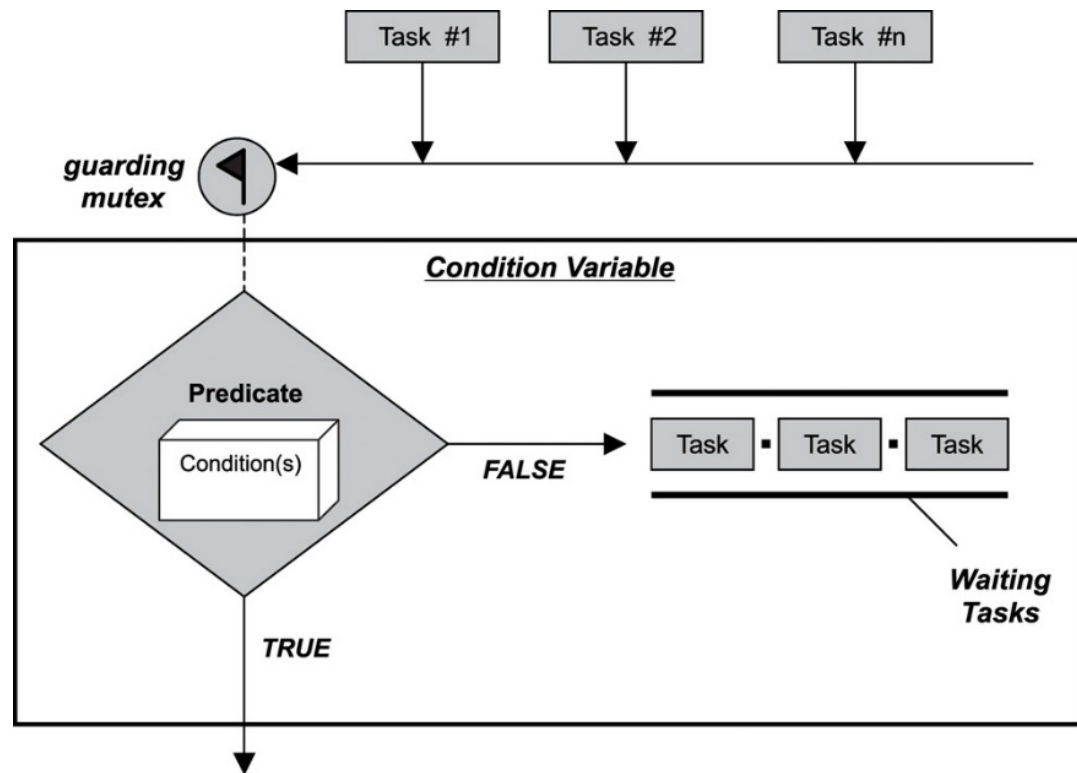
# Difficulties in using Semaphores

---

- Semaphores provide a very general mechanism for synchronization.
  - The power of semaphores derives from calls to P() and V() that are unmatched.
  - E.g., reader/writer and dining phil.
  - Unlike mutex (locks) where we lock() and unlock()
- This means that it's very tricky to get it right (e.g., no single fixed pattern of use and/or more than one semaphores).

# Condition Variables

- **Condition variables** is a synchronization primitive that helps us **model events** (rather than resources in the case of semaphores).
- A condition variable represents some condition that a thread can:
  - **Wait on**, until the condition occurs; or
  - **Notify other waiting threads that the condition has occur**
- It provides a place to wait (queue).



# Condition Variables

---

- ❑ Operations on condition variables:
  - ❑ wait() -- Block until another thread calls signal() or broadcast() on the CV
  - ❑ signal() -- Wake up one thread waiting on the CV
  - ❑ broadcast() -- Wake up all threads waiting on the CV
- ❑ Pthread
  - ❑ pthread\_cond\_wait(pthread\_cond\_t \*c, pthread\_mutex\_t \*m);
  - ❑ pthread\_cond\_signal(pthread\_cond\_t \*c);
  - ❑ pthread\_cond\_broadcast (pthread\_cond\_t \*c);
- ❑ Used with mutex

# Broadcast

---

- The `pthread_cond_broadcast()` function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task.
- Single producer/multiple consumer problem
  - The producer would notify all consumers that might be waiting; more throughput on a multi-processor.
- Read-write lock.
  - Wakes up all waiting readers when a writer releases its lock.

## Recommended reading

<http://pages.cs.wisc.edu/~remzi/OSFEP/threads-cv.pdf>

# Producer and consumer

```
cond_t empty, fill;

void produce(int item) {
    mutex_lock(&mutex);
    while (count == MAX)
        cond_wait(&empty, &mutex);
    put(item); // put an item in the
               // circular buffer
    cond_signal(&fill); // signal an
                       // item is filled
    mutex_unlock(&mutex);
}
```

```
int consumer() {
    mutex_lock(&mutex);
    while (count == 0)
        cond_wait(&fill, &mutex);
    int item = get();
    cond_signal(&empty);
    mutex_unlock(&mutex);
    return item;
}
```

- Important to use while statement to check the condition after waking up. It handles spurious wakeup.

- **rw\_mutex**: reader와 writer간 상호배제 semaphore 공유 (즉, 이는 read와 write간 경쟁을 의미함)
- **mutex**: read\_count 갱신시 상호 배제 보장용
- **read\_count**: 현재 몇 개의 프로세서들이 객체 읽는지 알려줌

#### □ The structure of a **writer process**

```
do {
    wait(rw_mutex);

    ...
    /* writing is performed */
    ...

    signal(rw_mutex);
} while (true);
```

- W1과 N개의 Reader가 있는 상황에서, 예를 들어,  $W1 \rightarrow R1 \rightarrow R2 \rightarrow R3 \rightarrow \dots$  가 발생하여, W1이 starvation이 됨. 이를 방지하는 코드.
- 이를 위해, writer에 rw\_mutex semaphore를 사용하고,
- 또한, Reader들 중에서, W1과 경쟁하는 Reader는 하나만(첫번째) reader( $\text{if}(\text{read\_count} == 1)$ )만 되도록 함.
- 즉, 나머지 N-1개의 Reader는 W1과 경쟁하지 못하게 하여(즉, rw\_mutex을 기다리지 않도록 하여), W1이 starvation 되는 상황 없앴
- 나머지 N-1개의 Reader들 중에서의 선택은 scheduler에 의해 선택됨.

#### □ The structure of a **reader process**

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);

    signal(mutex);

    ...
    /* reading is performed */
    ...

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

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

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

## < 사례 >

```
writer process
do {
    wait(rw_mutex);

    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);
```

```
reader process
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

read\_count

rc=1

R사이의 scheduling: FIFO로 가정

R1이 rw\_mutex 세마포를 잡은 이후에는 rw\_mutex 값은 0으로 바뀜.. R2, R3가 scheduling 되어 공유공간 Reading이 일어나고 빠져나가게 됨. 맨마지막 Reader인 R3가 빠져나갈때, rw\_mutex 값이 1로 되면서, 다시 W1은 공유공간 Writing 기회 잡음

즉, Readers 들을 한번씩만 수행되어, W1이 기아에 빠지는 것을 막음

