

중간고사 시험 범위는 6강 semaphore 전까지!

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

Chapter 6 Synchronization Tools

Heonchang Yu
Distributed and Cloud Computing Lab.

Contents

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Evaluation

Objectives

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

Background

two face locking protocol?

- Concurrent access to shared data may result in data inconsistency
시간적인 갭은 발생할 수 밖에 없지만, 룰에 따라 순서를 부여하는 것은 불만을 제기할 수 없음
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
동기화를 진행하는 것의 목적 : 순서를 정해주는 것
- Suppose we want to modify the algorithm to remedy this deficiency (allowed at most BUFFER_SIZE – 1 items in the buffer at the same time).
 - One possibility is to add an integer variable, count, initialized to 0.
 - count is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Semaphore = Two face locking protocol 이 거의 동일한 개념이라 보면 된다

Producer process

producer, consumer를 전역 변수로 설정해서 사용

while (true)

```
    /* produce an item and put in next_produced */
    while (count == BUFFER_SIZE)           buffer가 꽉 차있을 경우
        ; /* do nothing */
    buffer[in] = next_produced;            insert될 비어있는 공간에 값을 할당하겠다
    in = (in + 1) % BUFFER_SIZE;
    count++;                            buffer에 데이터를 하나 추가했으므로
}

```

Circular Queue를 구현했던 것을 Chapter 3에서 했던 것을 기억하면 된다

Consumer process

```
while (true) {  
    while (count == 0)          buffer가 비어 있을 때는 아무것도 하지 않는다  
        ; /* do nothing */  
    next_consumed = buffer[out];  buffer에 있는 값을 하나 삭제하겠다  
    out = (out + 1) % BUFFER_SIZE;  
    count--;                  count 지우기  
    /* consume the item in next_consumed */  
}
```

Race Condition

race condition == 경쟁 관계에 있는 상황이다

공유하는 자원이 있어서, 서로 경쟁관계에 있을 때 어떻게 할 것인가?

- **count++** could be implemented as

register1 = **count**

register1 = register1 + 1

어셈블리어도 system friendly 언어이기 때문에 3개 이상의 state로 구성

count = register1

- **count--** could be implemented as

register2 = **count**

register2 = register2 - 1

count = register2

어셈블리어라 단순한 코드라도 좀 더 길게 구성됨

이것도 실제 하나하나 의미로 나눌때는 interleaved 문제가 발생할 수 있음

Counter 값이 섞인다

- Consider this execution interleaving with "counter = 5" initially:

T ₀ :	producer execute	register1 = count	{register1 = 5}
T ₁ :	producer execute	register1 = register1 + 1	{register1 = 6}
T ₂ :	consumer execute	register2 = count	{register2 = 5}
T ₃ :	consumer execute	register2 = register2 - 1	{register2 = 4}
T ₄ :	producer execute	count = register1	{ count = 6}
T ₅ :	consumer execute	count = register2	{ count = 4}

Count ++ / Count — 가 interleaved 되었을 때, 발생할 수 있는 심각한 문제는 무엇인가?

결국에는 order가 가장 중요한 요소다

Synchronization도 order고, DB도 order를 정하는 것이다

Critical-Section Problem

코드 영역을 의미한다. 상호간의 conflict 또는 synchronization이 발생할 수 있어 특정 툴을 이용해 접근을 통제한다

동시 접근을 허용하는 경우 : reading만 할 때
대부분의 경우에는 order를 정해줘야한다

- Each process has a segment of code, called a critical section, in which the process may be accessing – and updating – data that is shared with at least one other process.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section.
- General structure of a typical process
 - Entry section - Each process must request permission to enter its critical section.
 - Exit section

```
while (true) {
```

entry section

critical section

exit section

끝났다는 것을 표현을 꼭 해줘야한다

remainder section

}

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section,
상호 배제
then no other processes can be executing in their critical sections.
pi만 실행가능하다

2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely. 적절하게 smooth하게 실행되어야한다.

3. **Bounded Waiting** - There exists a bound, or limit, 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.

무한한 대기 시간은 적절하지 못하니, n개의 프로세스가 있을 때 나라는 프로세스 p_i 는 $n-1$ 개 만큼 기다리면 내 차례가 올 것이라 판단함

Race condition when assigning a pid

- Two processes, P_0 and P_1 , are creating child processes using the fork() system call. fork() returns the process identifier of the newly created process to the parent process.
- There is a race condition on the variable kernel variable next_available_pid which represents the value of the next available process identifier.
- Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

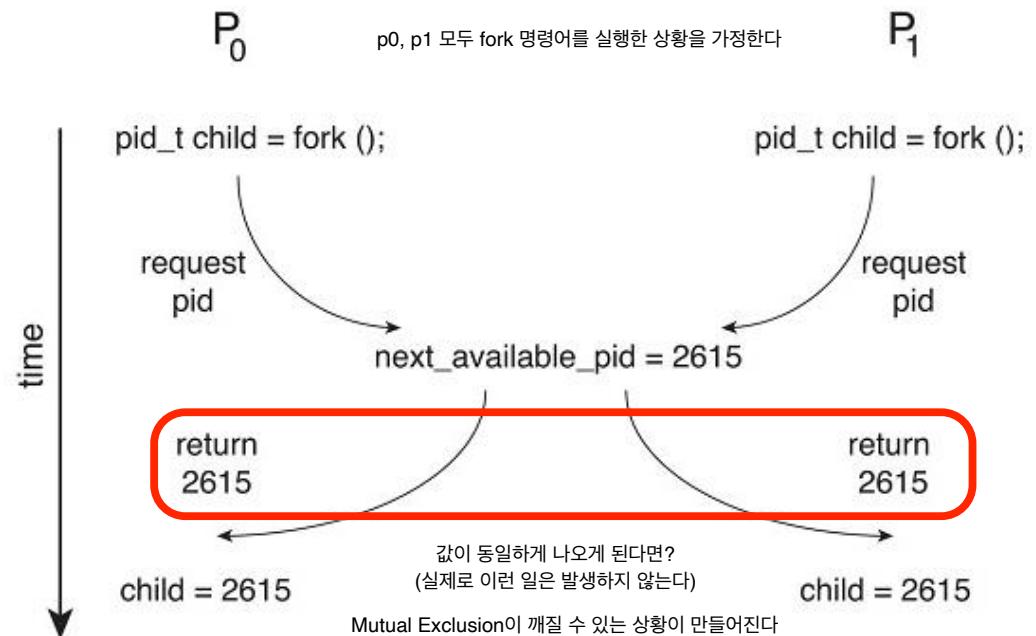


Figure 6.2 Race condition when assigning a pid.

Alternating Algorithm

11, 12페이지 알고리즘은 2개의 프로세스가 동작한다고 가정하고 작성된 내용임
두 프로세스 사이의 synchronization에 대한 얘기다
 P_i, P_j 2개의 프로세스

repeat turn 값은 정수 자료형으로 선언되어 있는 상태
 while turn $\neq i$ **do** no-op;

Mutual Exclusion만 만족하고
progress, bounded waiting은 만족하지 않음

turn이 i 면 크리티컬 섹션이 실행된다. *CRITICAL SECTION*

turn := j; exit section 부분이라 이해해도 무방하다

turn 값이 i 가 실행되면, i 가 되고, j 가 실행되면 j 가 되어
변경되는 상태임. 그래서 progress, bounded
waiting의 조건이 보장 안됨

REMAINDER SECTION

until false;

turn이 뭐냐에 따라 수행이 되느냐 안되느냐 결정되는 사항임

p_i, p_j 가 교차되는데, p_i 가 초기화 되지 않으면 P_j 는 무조건 기다리는 상황임

State Testing Algorithm

이 알고리즘은 현재 $p[i]$ 입장에서 작성된 상태임

repeat

$\text{flag}[i] := \text{true};$ $\text{flag}[i]$ 는 상태를 의미

while $\text{flag}[j]$ **do** no-op; j 의 프로세스를 체크

CRITICAL SECTION

$\text{flag}[i] := \text{false};$ i 의 초기 상태로 변경

REMAINDER SECTION

until false;

p_i, p_j 가 동시에 접근하는 상황에 progress, bounded waiting이 만족 안됨

deadlock (교착상태)가 발생할 수 있다
프로세스가 죽은 게 아니라 프로세스가 꼬여서 서로 기다리고 있는 상태

Peterson's Solution

- Two process solution 앞의 2가지 알고리즘 (Alternating Algorithms, Start Test Algorithm)
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two data items:
 - ✓ 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 its critical section. **flag[i] = true** implies that process P_i is ready to enter its critical section.

Process P_i in Peterson's Solution

```
while (true) {  
  
    flag[i] = TRUE;  
    turn = j;           deadlock을 깨주는 상황이 만들어진다 (Tie Break)  
    while (flag[j] && turn == j);  상대방의 플래그 값만 체크하면 된다  
    상대방의 turn 값만 확인하면 된다. 현재 process pj가 exit 코드를 실행하고 나갔거나, 실행된 적이 없는 초기 상태일 가능성이 매우 높음  
    critical section  
  
    flag[i] = FALSE;  
    remainder section  
  
}
```

entry code 부분

exit code 부분

Atomic Operation : turn 값은 i or j로 반드시 유지가 된다
turn 값을 바꾸는 것을 atomic하다라는 표현으로 얘기한다

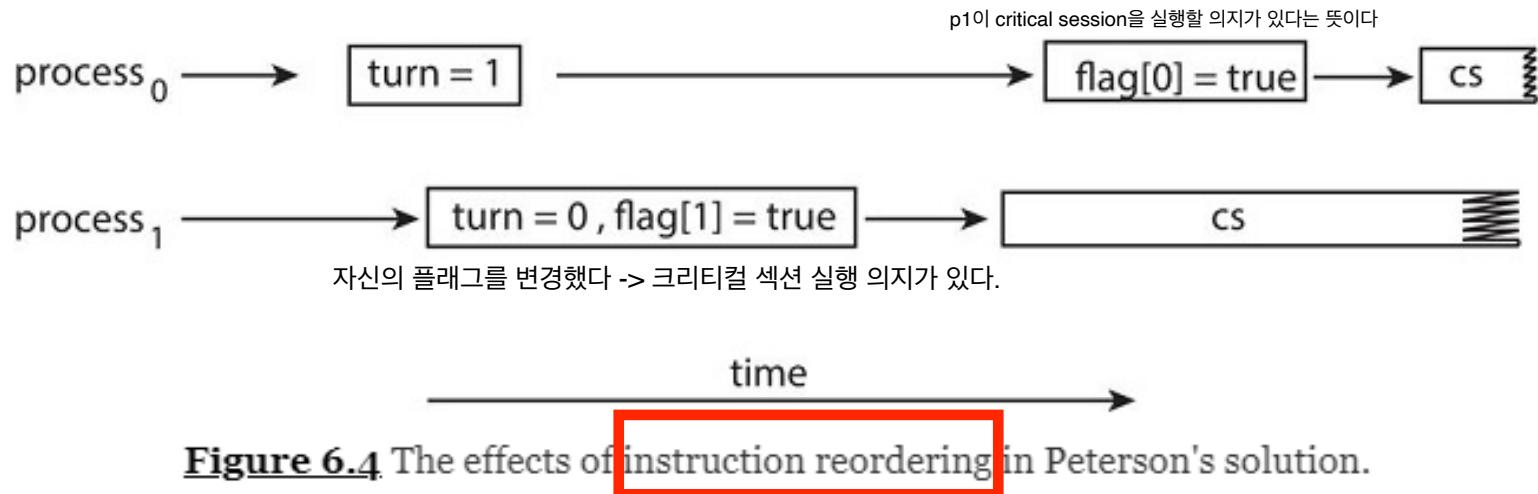
- Mutual exclusion is preserved.
- The progress requirements is satisfied.
- The bounded-waiting requirement is met.

Process P_i in Peterson's Solution

경우에 따라서 worst case를 가정하고, reordering이 발생하면 문제가 있을 수 있다

- Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson's solution are reordered.
- It is possible that both threads may be active in their critical sections at the same time, as shown in Figure 6.4.

이 내용은 worst case 시나리오에 해당한다고 보면 된다



경우에 따라 더 빠른 실행을 위해 reordering을 할 수도 있다 (반드시 한다는 것은 아니다)

최근 버전에 추가된 내용이고, 시스템 환경 자체가 계속 변화되면서 새로운 내용들이 추가되기 때문이다.

이런 상황이 정말로 발생할 것이라 생각할 필요는 없고, 책을 집필한 사람이 고의로 만들어 낸 것이라 보면 된다

Hardware Support for Synchronization

피터슨 알고리즘에서 reordering 문제를 해결하기 위한 방법

- To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies.
- Example: data that are shared between two threads

boolean flag = false; 시간적인 gap은 여기서 플래그가 만든다

int x = 0; Thread와 Thread 사이에 이 2개의 변수를 전역 변수로 활용하는 것 같다

✓ Thread 1: while (!flag)

;

print x;

Thread 2: x = 100;

flag = true;

x=100이 할당되기 전에 0이 출력되는 상황이 있을 수 있다

❖ The expected behavior is that Thread 1 outputs the value 100 for x

❖ However, as there are no data dependencies between the variables flag and x, it is possible that a processor may reorder the instructions for Thread 2 so that flag is assigned true before assignment of x = 100.

❖ In this situation, it is possible that Thread 1 would output 0 for x.

Hardware Support for Synchronization

- Memory model
 - **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
 - **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.
- Computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors.
- Such instructions are known as **memory barriers** or **memory fences**.
- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Hardware Support for Synchronization

- Reordering of instructions could have resulted in the wrong output
- Use a memory barrier to ensure that we obtain the expected output.
- Thread 1:

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

✓ We guarantee that the value of `flag` is loaded before the value of `x`.
- Thread 2:

```
x = 100;
memory_barrier();
flag = true;
```

✓ We ensure that the assignment to `x` occurs before the assignment to `flag`.

Hardware Support for Synchronization

교수님이 지속적으로 언급하는 개념 -> ACID

- Modern computer systems provide special hardware instructions.
 - **Atomic = non-interruptable**

프로세서가 여러개인 상황
multi-core environment 상황이다

- Either to test and modify the content of a word – `test_and_set()`
- Or to swap the contents of two words – `compare_and_swap()`

- `test_and_set()` instruction

```
boolean test_and_set (boolean *target) {
```

```
    boolean rv = *target;
```

```
    *target = true; False 값을 True로 바꾼다
```

```
    return rv;
```

```
}
```

주로 연산이라하면 write를 진행하는 update 연산일 것이다

- If two `test_and_set()` instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Solution using test_and_set()

- Shared boolean variable lock, initialized to false.
- Solution:

order가 주어지지 않는 상황에서는 Progress, Bounded Waiting이 보장이 안되는 상황이 만들어질 것

mutualization도 중요하지만, order가 정확하게 부여되어야한다.

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

compare_and_swap() Instruction (CAS)

- Mechanism that is based on swapping the content of two words
- If two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.
- Definition:

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

Solution using compare_and_swap()

- A global variable (lock) is initialized to 0;
- When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.
- Solution:

```
do {          이 조건을 만족하면 크리티컬 섹션을 실행할 권한을 얻는 것이다
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

        /* critical section */

    lock = 0;  다시 초기값으로 변경하는 것이다
    /* remainder section */

} while (true);
```

swap() Instruction

동일한 instruction이라 교수님이 빠르게 넘어감

- Definition:

```
void swap (boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Solution using swap

- Shared Boolean variable lock initialized to FALSE.
 - ✓ Each process has a local Boolean variable key.
- Solution: 옛날 버전의 책에 담겨 있는 내용 (최근 버전에는 없는 내용이다)

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        swap (&lock, &key );  
        // critical section  
    lock = FALSE;  
    // remainder section  
} while ( TRUE);
```

여기 while문 조건은 True 값을 기준으로 움직이기 때문에,
나머지 프로세스들은 대기하게 될 것임
lock & key 값을 sorting하는 형태임

lock이 False가 되는 시점부터 다른 프로세스 실행

- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

Bounded-waiting mutual exclusion with CAS

- Algorithm using the compare_and_swap() instruction that satisfies all the critical-section requirements
 - ✓ boolean waiting[n] /* initialized to false
 - ✓ Int lock /* initialized to 0
- Requirements
 - ✓ mutual-exclusion
 - ❖ Process P_i can enter its critical section only if either $\text{waiting}[i] == \text{false}$ or $\text{key} == 0$
 - ✓ progress
 - ❖ A process exiting the critical section either sets lock to 0 or sets $\text{waiting}[j]$ to false.
 - ✓ bounded-waiting
 - ❖ It scans the array waiting in the cyclic ordering $(i+1, i+2, \dots, n-1, 0, \dots, i-1)$ and designates the first process in this ordering that is in the entry section ($\text{waiting}[j] == \text{true}$).

Bounded-waiting mutual exclusion with CAS

```
while (true) {  
    waiting[i] = true;  
    key = 1;  
    while (waiting[i] && key == 1)  
        key = compare_and_swap(&lock, 0, 1);  
  
    waiting[i] = false; // 크리티컬 섹션 실행 전에 Waiting 값을 False로 바꾼 것에 가능한 시나리오는 2가지  
    /* critical section */  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i) // 전체를 다 체크했는데, (한 바퀴를 다 돌고서)  
        lock = 0; // 프로세스 중에 크리티컬 섹션 실행 의지가 있는 것이 하나도 없다는 뜻  
  
    else // pi가 값을 변경해주는 것이다.  
        waiting[j] = false; // 내가 n-1만큼 기다리면 내 프로세스 실행이 온다 (하염없이 기다리지 않는다는 뜻)  
    /* remainder section */  
}
```

28쪽의 알고리즘 내용과 동일한 내용이다

waiting 값이 True라면,
지금 크리티컬 섹션 진입을 위한 준비 중이라는 것이다.
(모든 프로세스는 이곳 while문에서 대기 중)

j가 i와 같지 않은 동안 실행되는 것이다(즉, 다시 돌아서 자신의 순서로 왔을 때)
-> 프로세스가 실행되면서 i, i+1, ..., n-1 실행되고 나서 다시 자신의 프로세스 순서에 오는 것

Bounded-waiting mutual exclusion with test_and_set()

- Algorithm using the test_and_set() instruction that satisfies all the critical-section requirements
- Mutual-exclusion
 - ✓ P_i can enter its critical-section only if either $\text{waiting}[i] == \text{false}$ or $\text{key} == \text{false}$.
- Progress
 - ✓ A process exiting the critical section either sets lock to false or sets $\text{waiting}[j]$ to false.
- Bounded-waiting
 - ✓ When a process leaves its critical section, it scans the array waiting in the cyclic ordering $(i+1, i+2, \dots, n-1, 0, \dots, i-1)$.

Bounded-waiting mutual exclusion with test_and_set()

```
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);
```

여기까지가 시험범위다
(중간고사)

Mutex Locks

공유하는 자원에 대해서는 서로 다른 유저 or 애플리케이션에게 순서를 어떻게 부여할 것인가?

- The hardware-based solutions to the C/S problem are complicated as well as generally inaccessible to application programmers.
- OS designers build software tools to solve critical-section problem.
- The simplest of these tools is the **mutex lock**.
 - ✓ Use the mutex lock to protect critical sections and thus prevent race conditions
 - ✓ A process must **acquire** the lock before entering a critical section; it **releases** the lock when it exits the critical section.
 - ✓ Boolean variable (`available`) indicating if lock is available or not
 권한을 받는 것이 *acquire*, 권한을 해제하는 것이 *release*
- Calls to **acquire()** and **release()** must be performed atomically.
 - ✓ Usually implemented via hardware atomic instructions
 CS 수행을 위해 권한을 받는 과정
- But this solution requires **busy waiting**
 - ✓ This lock is called a **spinlock** *lock*이 해제되기를 *waiting*
 - ✓ Because the process "spins" while waiting for the lock to become available.

Mutex Locks – acquire() and release()

```
release() {  
    처음에 true기 때문에 acquire 부분의 반복문 부분은 초기에 False임  
    available = true;  
}  
}
```

```
do {           2-phase locking protocol 과 동일
    acquire lock -> CS 진입을 위해 획득하는 과정
    acquire lock
    critical section
    release lock      unlocking하는 것과 똑같다
    remainder section
} while (true);
```

Semaphore

옛날부터 있던 개념이기는하다.

- Semaphore S – integer variable
 $\text{2개의 } operation : \text{wait}() \& \text{signal}()$ 순서도 항상 맞춰짐
- Two standard atomic operations : **wait()** and **signal()**

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

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

$s=1$ 로 초기화 되었다고 가정하자 (별다른 설명이 없으면)

✓ **wait (S) { entry**

 처음 수행하는 프로세스는 $S=1$ 이기 때문에 무조건 CS 실행 가능

 while $S \leq 0$

 ; // busy wait

 두, 세 번째 프로세스는 *busy wait*

$S--$; $s=0$ 이 될 것

P_i 는 CS 수행

}

 atomic 연산 : 특정 연산이 수행될 때,
 그 연산이 다른 작업에 의해 방해받지 않는다는 것

 하드웨어적인 문제로 중단이 될 수는 있어도,
 타 작업이 중간에 중단시킬 수는 없음

✓ **signal (S) { exit**

$S++$; $0 \rightarrow 1$

}

wait & signal은 동시에 실행될 수 없음
 atomic operation의 원칙을 깰 수 있다.

Semaphore

- Counting semaphore 정수 값이 범위가 정해져 있지 않고, 세마포 값이 변경되면서 사용 가능 일반적으로 세마포라고 하면 *counting semaphore*라고 보면 됨✓ integer value can range over an unrestricted domain semaphore의 초기 값은 시작 시 항상 1이 아니다.
상황에 따라 다르게 초기화
 - Binary semaphore✓ Integer value can range only between 0 and 1✓ Behave similarly to mutex locks
 - Can implement a counting semaphore S as a binary semaphore
 - Suppose we require that S_2 be executed only after S_1 has completed.
 - ✓ Common semaphore $synch$, initialized to 0 반드시 1일 필요는 없다. 이 설명은 하나의 예시 s_1 이 끝나야 s_2 가 실행되는 구조를 현재 만들고 싶은 것임
- $P_1:$
- $S_1;$
 $signal(synch);$
- 여기서 $synch$ 값을 1로 변경
- $P_2:$
- $wait(synch);$ $s \leq 0$ 이면 busy wait
 $S_2;$
- $synch$ 값이 0으로 초기화 되어 있기 때문에 S_2 는 *waiting*하게 될 것

wait -> signal 순서지, *signal -> wait*은 불가능하다.

Semaphore Implementation with no Busy waiting

- Disadvantage of the semaphore definition – busy waiting 끊임없이 Looping 도는 것이 문제다.
 - ✓ While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. – spinlock
- Disadvantage of the semaphore definition – busy waiting
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations.
 - ✓ When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However the process can suspend itself.
- Two operations
 - ✓ sleep (suspend) – places a process into a waiting queue associated with the semaphore
 - ✓ wakeup – changes the process from the waiting state to the ready state

개념적으로 데이터베이스 *concurrency control*과 동일하다.
단지 사용하는 위치가 다를 뿐

Semaphore Implementation with no Busy waiting

- 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
- Define a semaphore as a "C" struct

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

C언어에서 *semaphore*를 구현한다면 *struct*를 사용해 구현이 될 것이다

세마포마다 *queue*가 존재한다고 보면 된다.

Semaphore Implementation with no Busy waiting

- Implementation of wait():

```
wait (semaphore *S) {  
    S->value--;  
    if (S->value < 0) { 이미 CS를 수행하는 프로세스가 존재한다 (s<0)  
        나머지 프로세스는 queue에서 대기  
        add this process to S->list;  
        sleep(); } 프로세스 대기 배열에 추가하고, sleep  
        sleep 목적 : busy waiting을 피해보자  
    }  
}
```

- Implementation of signal():

```
signal (semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P); } 어떤 프로세스를 깨내는지는 2번째 문제고,  
        대기 중인 프로세스 중에 하나를 깨내는 것  
    }  
}
```

Semaphore Implementation

- It is critical that semaphores be executed atomically.
- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
 - ✓ In a single-processor environment, we can solve it by inhibiting interrupts during the time `wait()` and `signal()` operations are executing.
 - ✓ In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may be interleaved in some arbitrary way.
 - ❖ Disabling interrupts on every core can be a difficult task and can seriously diminish performance.

서로 다른 세마포에 대해서는 서로 의존성이 없기 때문에 (독립적이기 때문에) 같이 실행되도 문제 없지만,
같은 프로세스는 함께 실행될 수 없다.

코어가 여러개일 때는 동일한 세마포에 대해 동시에 접근할 수 있는 문제가
발생할 수 있기 때문에 (임의로 *interleaved*되는 상황) 피해야한다.
special hardware instruction 등을 통해

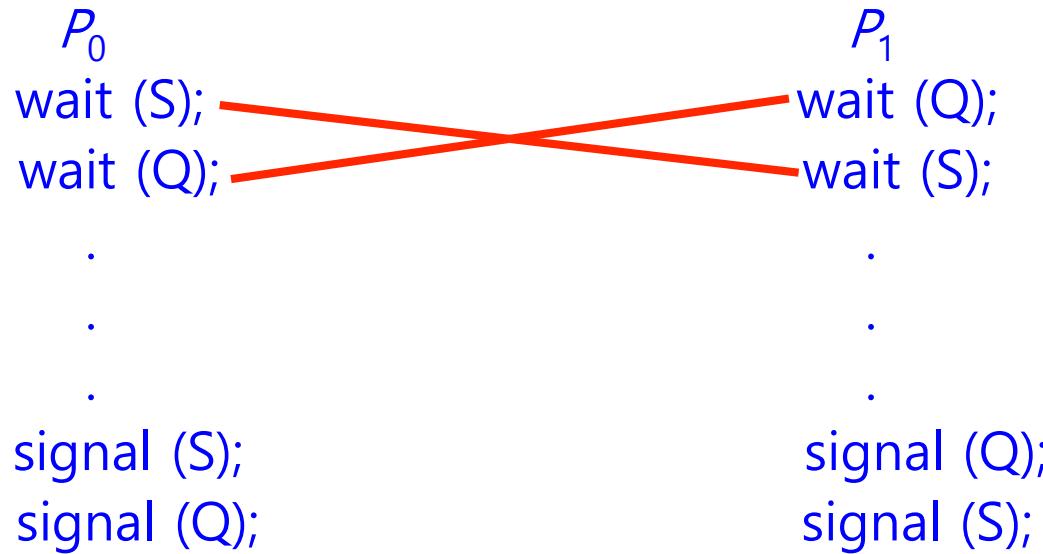
Semaphore Implementation

- We have not completely eliminated busy waiting with this definition of wait() and signal() operations.
 - ✓ We have moved busy waiting from the entry section to the critical sections of application programs.
 - ❖ These sections are short (if properly coded, they should be no more than about ten instructions)
 - ❖ The critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time.

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

세마포의 단점 중 하나로 언급되는 부분



- Deadlock (Starvation) – indefinite blocking
그냥 둘 중 하나의 프로세스를 *kill*해야함
 - ✓ Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. *cycle*을 끊어줘야한다
 - ✓ When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`.

Problems with Semaphores

- Using semaphores incorrectly can result in timing errors that are difficult to detect
 - ✓ An example of such errors in the use of counters in our solution to the producer-consumer problem
- Correct use of semaphore operations:
 - ✓ Interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed
 - ❖ Several processes may be executing in their critical sections simultaneously
 - Replaces signal(mutex) with wait(mutex)
 - ❖ A deadlock will occur
 - Omits the wait(mutex), or signal(mutex), or both

이 경우 모든 프로세스가 CS를 수행하면서 synchronization이 깨진다

signal(mutex) wait(mutex)

맨 처음 값만 CS 수행되나, 나머지는 모두 대기 (mutex 값이 계속 decrement)

wait(mutex) ... wait(mutex)

• Omits the wait(mutex), or signal(mutex), or both