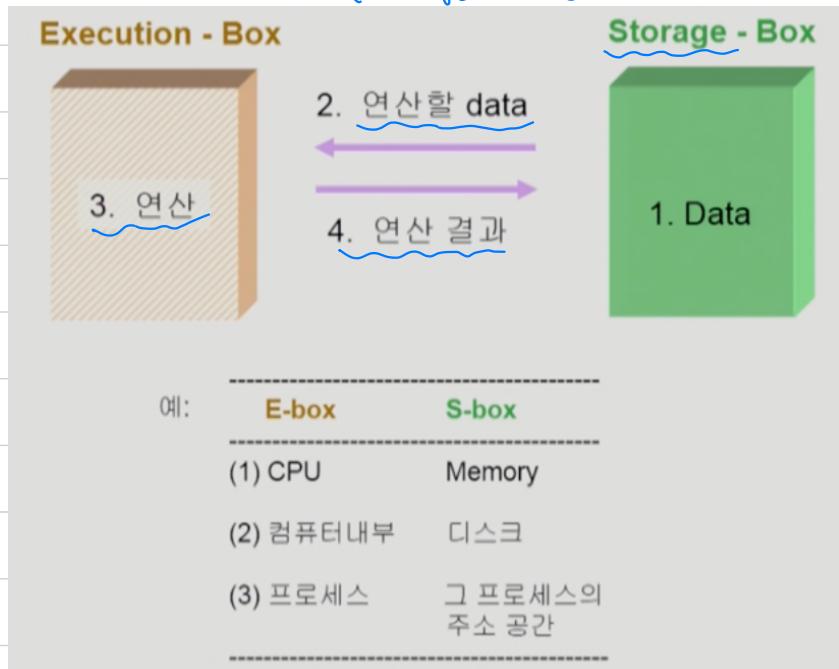


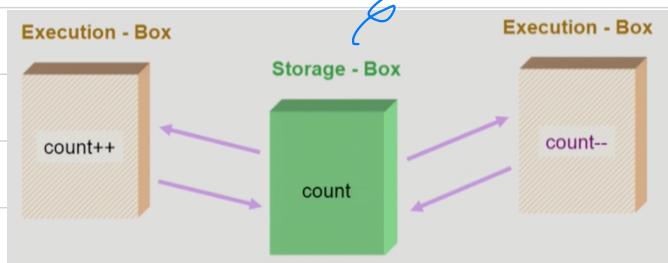
데이터의 접근

→ 컴퓨터 시스템 안에서 데이터가 접근되는 패턴



Race Condition

→ 데이터를 동시에 접근하여 할 때 ⇒ race condition



S-box를 공유하는 E-Box가 여러 있는 경우 Race Condition의 가능성이 있음.

Memory Address Space
CPU Space → Multiprocessor System
→ 같은 메모리를 사용하는 프로세스를

거울 대보 처리기를 갖춘다는 특성을 갖기 때문에
(예: 거울모드 수행 중 인터럽트로 커널드 다른 루틴 수행 시)

프로세스가 적절히 처리하지 못하는 부분이 있어 시스템을 봐야 커널의 코드가 실행 중인데
CPU를 뺏어 다른 프로세스로 넘기 있는데 이 프로세스가 또 시스템을 통해 커널의 데이터(?)
접근한다면 Race-Condition 발생 가능

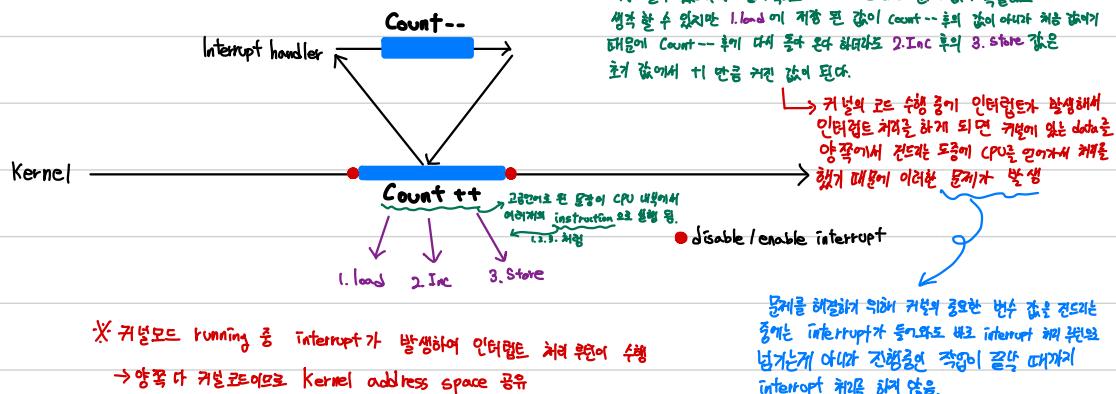
OS에서 race condition은 언제 발생하는가?

1. Kernel 수행 중 인터럽트 발생 시

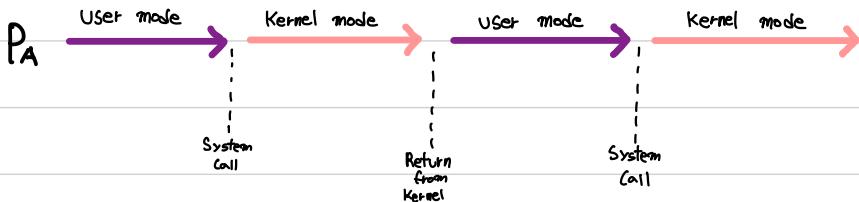
2 Process가 system call을 하여 Kernel mode로
수행 중인 Context switch가 일어나는 경우

3. Multiprocessor에서 Shared memory 내의 Kernel data

OS에서의 race condition (1/3) (interrupt handler vs Kernel)



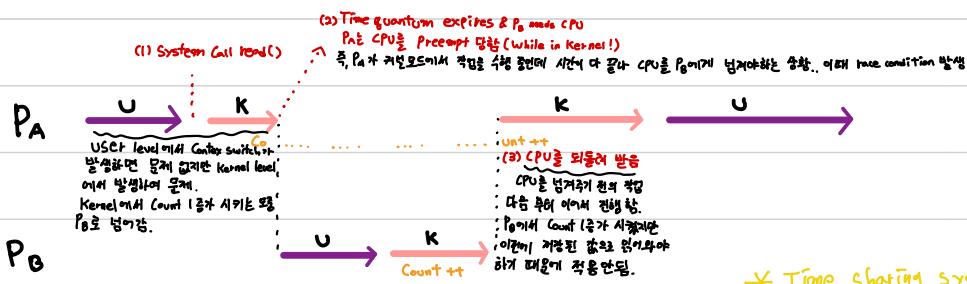
OS에서의 race condition (2/3) Preempt a process running in Kernel?



* 두 프로세스의 address space 간에서 data sharing이 있음

* * 그러나 System call을 하는 동안에는 Kernel address space의 data를 access하게 됨(share)

* * * 이 작업 중간에 CPU를 preempt 하자면 race condition 발생



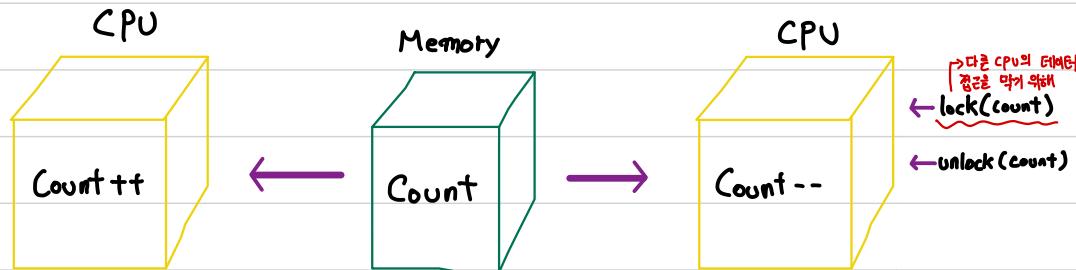
* * * 해결책: 커널 모드에서 실행 중일 때는 CPU를 preempt 하지 않는
 커널 모드에서 사용자 모드 돌아갈 때 preempt

* Time sharing system이라는 것은

Real time system이 아니다. 조금 시간이 더 걸리해서 시스템에 문제가 생기는 게 아니기 때문에 오히려 이런 문제를 쉽게 해결 가능.

OS에서의 race condition(3/3) multi processor

* 자주 등장하는 경우는 아래만 위의 두가 경우로는 해결이 안됨.



어떤 CPU가 마지막으로 count를 store 했는가? → race condition

multi processor의 경우 interrupt enable/disable로 해결되지 않음

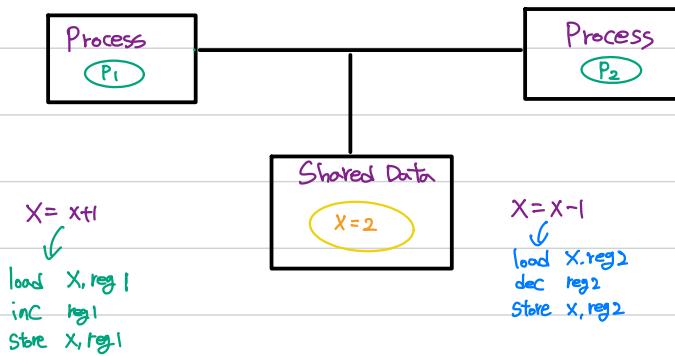
(방법1) 한 번에 하나의 CPU만이 커널에 들어갈 수 있게 하는 방법

(방법2) 커널 내부에 있는 각 공유 데이터에 접근할 때마다 그 데이터에 대한 lock/unlock을 하는 방법

Process Synchronization 문제

- 공유 데이터(shared data)의 동시 접근(concurrent access)은 데이터의 불일치(inconsistency)를 발생 시킬 수 있다.
- 일관성(consistency) 유지를 위해서는 협력 프로세스(cooperating process)간의 실행 순서(orderly execution)를 정해주는 메커니즘 필요.
- **Race Condition**
 - ↳ 여러 프로세스들이 동시에 공유 데이터를 접근하는 상황
 - ↳ 데이터의 최종 연산 결과는 마지막에 그 데이터를 다룬 프로세스에 따라 달라짐.
- race condition을 막기 위해서는 concurrent processes는 동기화(synchronize)되어야 한다.

Example of a Race Condition



※ 사용자 프로세스 P1 수행 중 timer interrupt가 발생해서 context switch가 일어나서 P2가 CPU를 잡으면?

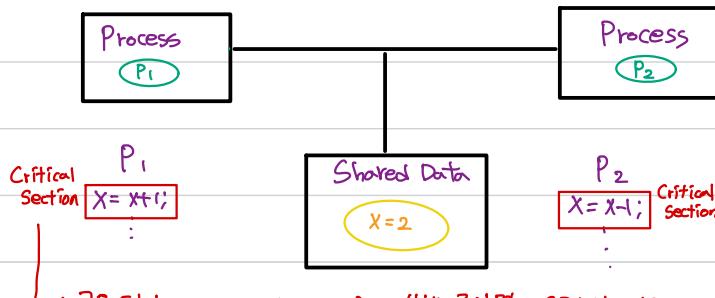
⇒ 보통은 문제가 생기지 않는다. 특별히 두 프로세스가 Shared memory를 쓰고 있다면 P1이 개별에 있고 데이터는 전부는 동안에 P2가 접근한다면 할 때 생기는 문제... 단순히 P1에서 P2로 넘어간다로 생각하는 문제는 아니다.

The Critical - Section Problem

— 각각의 프로세스가 공유 데이터를 동시에 사용하기를 원하는 경우

— 각 프로세스의 code segment에는 공유 데이터를 접근하는 코드인 critical section이 존재

- Problem
↳ 하나의 프로세스가 critical section에 있을 때 다른 모든 프로세스는 critical section에 들어갈 수 없어야 한다.



→ 공유 데이터에 접근하는 코드를 실행 중이면 CPU가 다른 프로세스에게 넘어 가더라도 공유 데이터를 접근하는 Critical Section에 들어가지 못하도록 함.

프로그램적 해결법의 충돌 조건

- Mutual Exclusion (상호 배제)

↳ 프로세스 P_i가 Critical Section 부분을 수행 중이면 다른 모든 프로세스들은 그들의 critical section이 들어가면 안된다.

- Progress (진행) ↳ 아무도 Critical Section에 없으면 끌어갈 수 있게 해줘야 한다는 조건

↳ 아무도 critical section에 있지 않은 상태에서 critical section에 들어가고자 하는 프로세스가 있으면 critical section에 들어가기 해줘야 한다.

- Bounded Waiting (유한한 대기)

↳ 특정 프로세스 입장에서 Critical Section이 못 들어가고 지나가게 오래 기다리는 Starvation의 생활이 일어나지 않는 풍경

↳ 프로세스가 critical section에 들어가려고 요청한 후부터 그 요청이 허용될 때까지 다른 프로세스들이 critical section에 들어가는 횟수에 한계가 있어야 한다.

* 가정

↳ 모든 프로세스의 수행 속도는 이보다 크다.

↳ 프로세스를 간의 상대적인 수행 속도는 고정하지 않는다.

↳ 언제 발생하나: Critical Section이 들어가고자 하는 프로세스 세ট이 있는데 두개의 프로세스만 멈춰가면서 들어가고 나면 다시 뛰는 못 들어가고 아는 기다려야 하는 상황

Initial Attempts to Solve Problem

- 두 개의 프로세스가 있다고 가정 P₁, P₂

- 프로세스들의 일반적인 구조

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (1);
```

- 프로세스들은 수행의 동기화 (Synchronize)를 위해 몇몇 변수를 공유할 수 있다. → Synchronize variable

※ 어떻게 소프트웨어적으로 LOCK을 풀 수 있는지 알고리즘을 알아본다.

Algorithm 1

- Synchronization Variable

```
int turn;
initially turn=0; ⇒ P_i can enter its critical section if (turn==i)
```

- Process P_i

```
do {
    // 첫째로 풀 때까지
    while (turn != i); /* My turn? */
    critical section → 턴이 i이면 while문이 끝나고
    Critical section은 끝난다.
    turn = 1; /* Critical section에 진입한 후에는 턴은 1이 된다.
    턴을 설정하는 내용 */
    remainder section
} while (1);
```

- Process P_j

```
do {
    while (turn != j); /* My turn? */
    critical section
    turn = 0; /* Now it's your turn */
    remainder section
} while (1);
```

Satisfies mutual exclusion, but not progress

즉, 과정설명: 반드시 한 번씩 고대로 들어가야만 함 (swap-turn)
 그자 turn을 대신하고 빠져 나와야만 내가 들어갈 수 있음
 특별 프로세스가 더 빨리 critical section에 들어가게 한다면?

상호 배제는 만족 하지만 Progress를 만족하지 못함
 아무것도 Critical Section에 진입하지 않는데도 불구하고 Critical Section이
 들어가기 못하는 문제 발생

* 위 코드는 Critical Section이 반드시 고대로 들어가도록 조정 되어 있다.

즉, P_i가 한번 들어갔다 나와야지만 P_j의 차례가 되고, P_j가 들어갔다 나와야지만 P_i의 차례가 된다.
 극단적인 예로, P₀은 빠르게 들어가고 하고 P₁은 천천히 Critical Section에 들어가면 끝나고 하면 상대방이 Turn을
 바꿔주지 않기 때문에 P₀도 영원히 Critical Section에 들어가지 못하게 된다. ⇒ 차례로 동작하는 알고리즘 X

Algorithm 2

- Synchronization variables

v boolean flag[2]; → 본인의 Critical Section에 들어 가고자
 하는 의지를 표시
 initially flag[0]=false; /* no one is in CS */
 v P_i ready to enter its critical section if (flag[i]==true)

- Process P_i

```
do {
    // 들어가겠다는 의지를 표시
    flag[i] = true; /* Pretend I am in */
    while (flag[i]); /* Is he also in? then wait */
    critical section → 상대방이 Critical Section 사용하겠다는 표시를 했으면
    풀 때 까지 while문 안에서 기다림
    flag [i] = false; /* I am out now */
    remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement

- 동시에 실행되는 두 프로세스가 같은 Critical Section에 진입하려면

두 프로세스 모두 While문에 진입하기 전에
 flag[i]=true가 된다면 둘다 들어가게 됨
 계속 While문 안에서 상대방의 flag[i]=false를
 바꿔가기 만큼 가능할 것임

Algorithm 3 (Peterson's Algorithm)

- Combined synchronization variables of algorithm 1 and 2.

- Process P_i

```
do {  
    flag [i] = true;      /* My intention is to enter ... */  
    turn = j;            /* Set to his turn */  
    while (flag [j] && turn == j); /* wait only if ... */  
    critical section  
    flag [i] = false;  
    remainder section  
} while (1);
```

상대방의 flag가 true이고 turn은
상대방의 turn을 때면 while문 안에서 기다리고
그렇지 않으면 critical section 실행

- Meets all three requirements; Solves the critical section problem for two processes

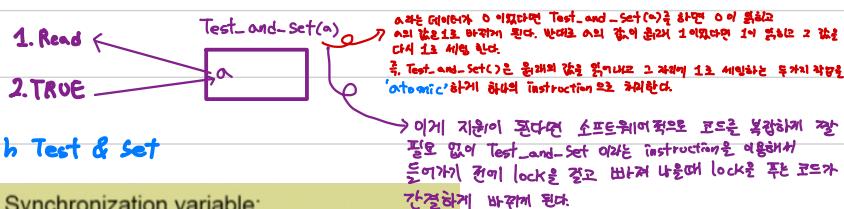
- Busy Waiting!! (계속 CPU와 memory를 쓰면서 wait)
 $(= \text{spin lock})$

• 본인의 CPU 활용 시간을 while문 안에서 다시 사용하는 문제 발생
• 왜냐하면 while문에서 뺏어 낼 수 있는 코드는 다른 프로세스가 CPU 활용 받아
• 그걸 바꾸어야 하기 때문에 while문에서 CPU와 memory 사용이 일어나는 비효율적!!

* 왜 간단하게 코드를 짜지 않았나? \Rightarrow 코드 중간 출전에 인해 CPU를 뺏길 수 있다는 가정 하에 코드를 짜야 보니까 소프트웨어적으로 복잡한 코드가 될

Synchronization Hardware

- 하드웨어적으로 Test & modify를 atomic하게 수행할 수 있도록 지원하는 경우 앞의 문제는 간단히 해결



- Mutual Exclusion with Test & Set

Synchronization variable:
boolean lock = false;
처음 lock이라는 값은 0

Process P_i

```
do {  
    lock이 점령되었지 확인하고 안정화 있다면  
    내가 lock을 걸고 critical section에 들어갈 때 lock을 Test_and_Set 하는  
    while (Test_and_Set(lock));  
    critical section  
    lock = false;  
    remainder section  
}
```

lock이 0이면 lock을 걸고 동시에 while문을 빠져나와
critical section으로 가는
lock이 1이면 대체 lock을 걸고 세팅하고 계속 while문을 걸고 있거나
* 하드웨어적으로 값을 얻는 작업과 세팅하는 작업을 atomic으로
실행 할 수 있다고 하면 lock을 걸고 푸는게 간단하게 해결 된다는 것.

Semaphores

- 앞의 방식들을 추상화 시킴

- Semaphores(S)

추상 자료형은 논리적으로 정의를 하는 거지 컴퓨터에서 구현되는 자료형과는
별개의 자료형

v integer variable.

v 아래의 두 가지 atomic 연산에 대해서만 접근 가능

변수가 정의
했었는데 이는 자원의 것수를
의미함

예) 봄 수가 5다
= 자원의 것수가 5개이다.

P(S):
Semaphore 값을 (음수 GI리Ex)
획득하는 연산

while ($S \leq 0$) do no-op;
 $S--;$

i.e. wait

If positive, decrement-&-enter.

Otherwise, wait until positive (busy-wait)

V(S):

다 써고 만족하는 연산

$S++;$

P, V 연산은 atomic하게 연산 된다고 가정.

자원 반납

* Semaphores를 왜 사용하나?

→ 한 세울의 Lock을 걸고 풀고는 Semaphore를 이용해서
프로 그레미에게 간단하게 제공 가능
공유 자원을 획득하고 반납하는 걸은 Semaphore가 해 줍니다

Critical Section of n Processes

Synchronization variable

semaphore mutex; /* initially 1: 1개가 CS에 들어갈 수 있다 */

Process P:

do {

P(mutex); /* If positive, dec-&-enter, Otherwise, wait. */

critical section

V(mutex); /* Increment semaphore */

remainder section

} while (1);

CPU 계속 사용 중

busy-wait은 효율적이지 못함 (= spin lock)

Block & Wake up 방식의 구현 (= sleep lock)

Semaphore 가 자원 된다면

프로 그레미는 P연산, V연산만

하면 되는 것이고 P, V를 어떻게 구현 할지는

구현하는 시스템에서 생각 해야 할 목

Block / Wake up Implementation

- Semaphore를 다음과 같이 정의

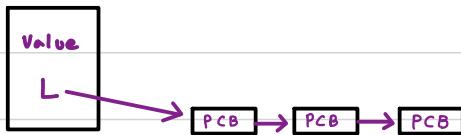
```
typedef struct
{
    int value;           /* semaphore */
    struct process *L;  /* process wait queue */
} semaphore;
```

- block과 wake up을 다음과 같이 가정

v block : 커널은 block를 호출한 프로세스를 suspend 시킴
이 프로세스의 PCB를 semaphore에 대한 wait queue에 넣음.

v Wake up(p) : block된 프로세스 P를 wake up 시킴
이 프로세스의 PCB를 ready queue로 옮김

Semaphore:



Implementation (block/wake up version of PC & VC)

- Semaphore 연산이 아래와 다음과 같이 정의됨

```
P(S):
    S.value--;
    /* prepare to enter */
    if (S.value < 0) /* Oops, negative, I cannot enter */
        {
            add this process to S.L;
            block();
        }

V(S):
    S.value++;
    if (S.value <= 0)
        {
            remove a process P from S.L;
            wakeup(P);
        }
```

자유가 있다면 바로 연고
없다면 잠들(blocked)

값을 내 놓았는데 값이 0 이하라는 것은
누군가가 지금 잠들어 있다는 뜻

remove a process P from S.L;
wakeup(P);

잠들기 있으면 깨워줌

* 여기서 S.value는 busy wait 이후의 값을 다룬다. busy wait 아닌 자유한방 가능 예제를 나타냈지만
S.value에서의 Value 값은 누군가가 잠들어 있는지를 알려줌.

Which is better?

- Busy wait vs Block/wake up

- Block/wake up overhead vs Critical Section 걸이

↳ Critical Section의 걸이가 긴 경우 Block/wakeup이 적음

↳ Critical Sections의 걸이가 매우 짧은 경우 Block/wake up 오히려 해보다 busy-wait 오버헤드보다 커질수 있음.

↳ 일반적으로 Block/wake up 방식이 더 좋음.

 ↳ 빌더없는 CPU를 갖고 있지 않고

 ↳ 자원을 누군가가 가지고 있다면 그냥 CPU를 분할하고 block/unlock을
 ↳ 돌아가는게 의미있는 CPU 사용률을 높힐 수 있다.

Two Types of Semaphores

- Counting semaphore

↳ 도메인이 0 이상인 임의의 정수값

↳ 주로 resource counting에 사용

- Binary Semaphore (= mutex)

↳ 0 또는 1 값만 가질 수 있는 semaphore

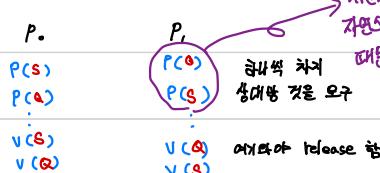
↳ 주로 mutual exclusion(lock/unlock에 사용)

Deadlock and Starvation

- Deadlock

↳ 두 이상의 프로세스가 서로 상대방에 의해 충돌 될 수 있는 event를 무한히 기다리는 현상

- S와 Q가 1로 초기화된 Semaphore 라 하자



↳ 자원을 갖는 순서를 P₁과 같은 순서로 바꿔주면
자연스럽게 deadlock 해결 Q를 획득 하려면 S 먼저 획득 해야하기
때문에 P₁에서 S를 받았을 때 P₂가 S를 획득하고 Q를 획득하게됨.

- Starvation

↳ indefinite blocking. 프로세스가 Suspend된 이유에 해제하는

 Semaphore 큐에서 빠져 나갈 수 없는 현상.