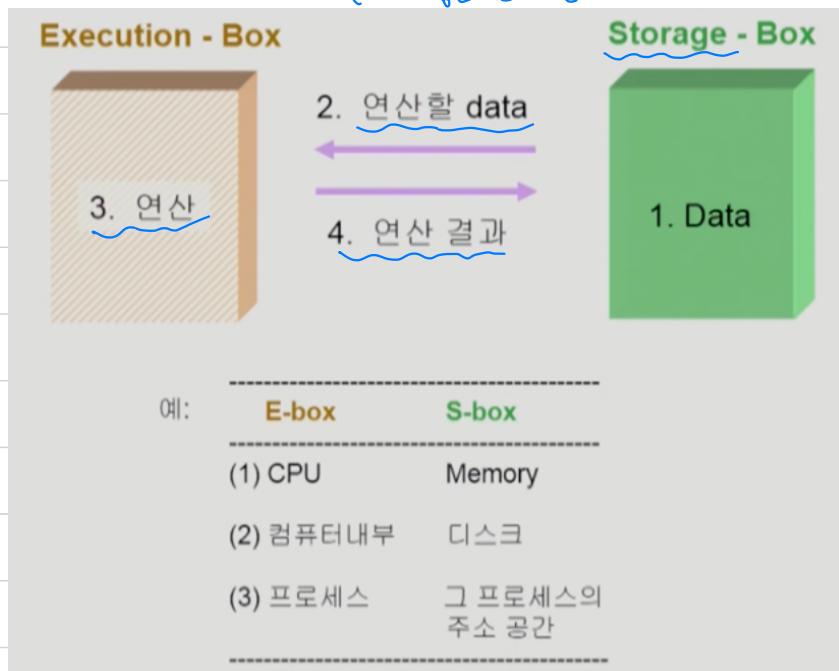


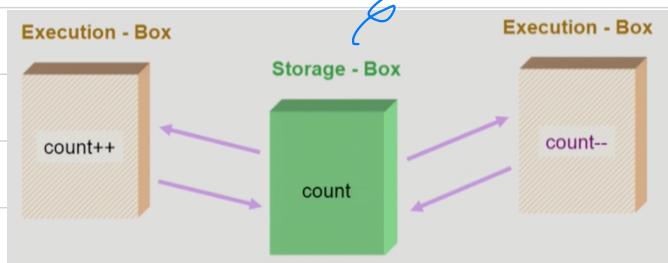
데이터의 접근

→ 컴퓨터 시스템 안에서 데이터가 접근되는 패턴



Race Condition

→ 데이터를 동시에 접근하려 할 때 → race condition



S-box를 공유하는 **E-Box**가 여러 있는 경우 **Race Condition**의 가능성이 있음.

↓
Memory Address Space
CPU Space → Multiprocessor System
→ 같은 메모리를 사용하는 프로세스들

커널 내부 타이밍을 정밀화하는 코드를 갖는다
(예: 커널 모드 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

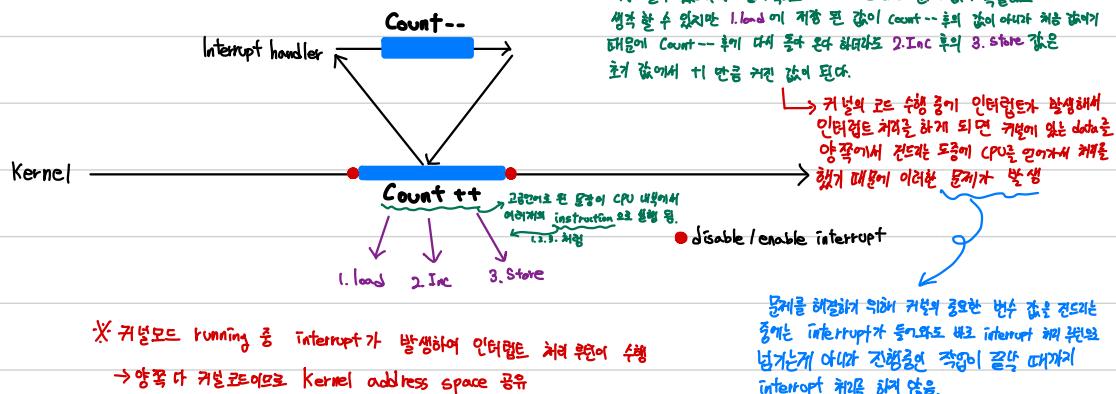
OS에서 race condition은 언제 발생하는가?

1. Kernel 수행 중 인터럽트 발생 시

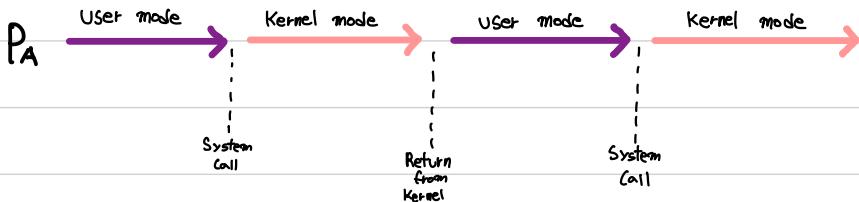
2 Process가 system call을 하여 Kernel mode로
수행 중인 Context switch가 일어나는 경우

3. Multiprocessor에서 Shared memory 내의 Kernel data

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



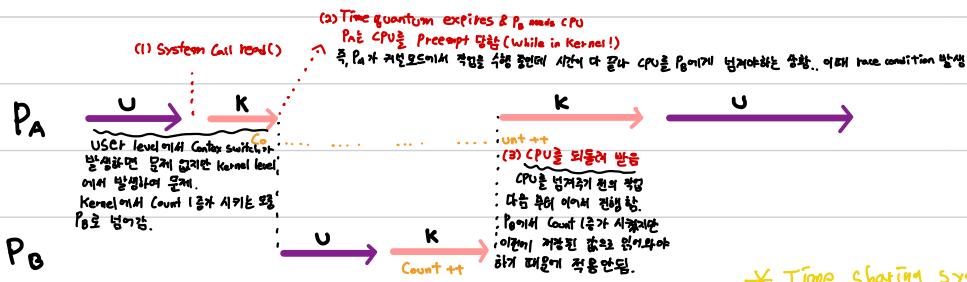
OS에서의 race condition (2/3) Preempt a process running in Kernel?



* 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



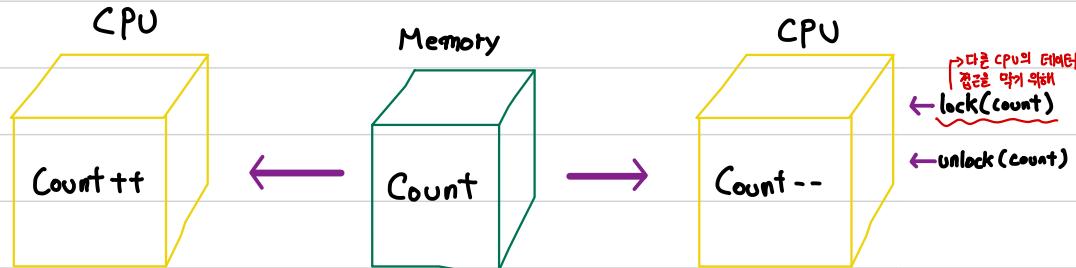
* * * 해결책: 커널 모드에서 실행 중일 때는 CPU를 preempt 하지 않는
 커널 모드에서 사용자 모드 돌아갈 때 preempt

* 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

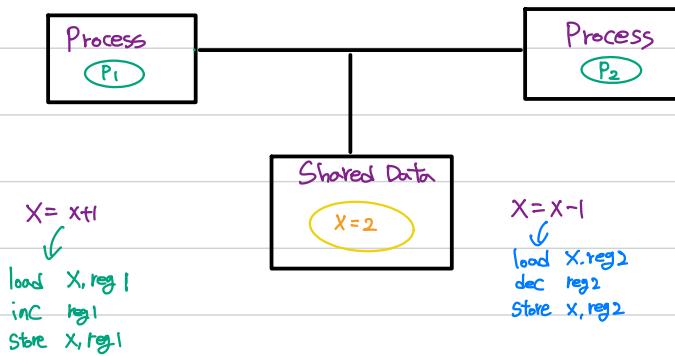
(방법1) 한 번에 하나의 CPU만이 커널에 들어갈 수 있게 하는 방법

(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?

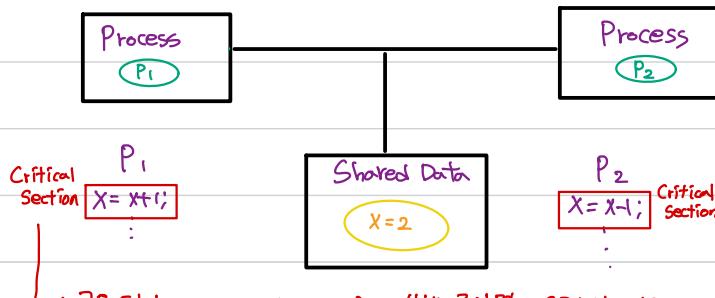
⇒ 보통은 문제가 생기지 않는다. 특별히 두 프로세스가 Shared memory를 쓰고 있다면 P1이 개별에 있고 데이터는 전부는 동안에 P2가 접근한다면 할 때 생기는 문제... 단순히 P1에서 P2로 넘어간다로 생각하는 문제는 아니다.

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 가정하기 않는다.

↳ 언제 발생하나: Critical Section이 들어가고자 하는 프로세스 세ট이 있는데 두개의 프로세스만 멈춰가면서 들어가고 나면 다시 점프하는 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.

```
do {
    while (turn != 0); /* My turn? */
    critical section → turn이 0이면 while문이 끝나고
                        Critical section은 끝난다.
    turn = 1;           /* Critical section에 진입한 후에는 turn은 1로 바뀐다.
                        Now it's your turn */
    remainder section
} while (1);
```

- Process P_i

```
do {
    while (turn != 1); /* 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 고대하고 들어가야만 함 (swap-turn)
 그자 turn을 대신하고 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 P₀은 한 번만 들어가면 Critical section에 들어가면 끝나고 하면 성과값이 100%
 바꿔주지 않기 때문에 P₀도 영원히 Critical section에 들어가기 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 {
    /* Critical section */
    flag[i] = true;          /* Pretend I am in */
    while (flag[i]);         /* Is he also in? then wait */
    critical section         /* Same behavior as above */
    flag [i] = false;        /* I am out now */
    remainder section
} while (1);
```

두 프로세스 모두 While 문에 진입하기 전에
 flag[i]==true가 된다면 둘다 들어가지 못하고
 계속 While 문 안에서 A와 B에서 flag=false로
 바뀌기 전을 기다렸던 것

- Satisfies mutual exclusion, but not progress requirement

- 동시에 실행까지 수행 후 끊임없이 양보하는 상황 발생 가능

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

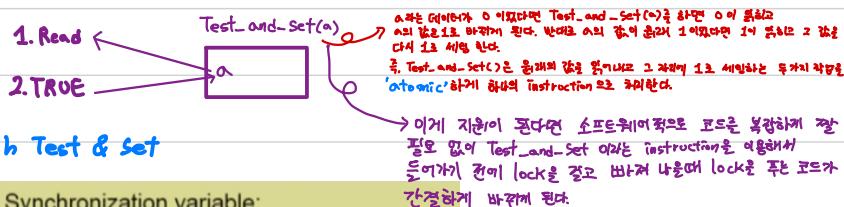
- Busy Waiting!! (계속 CPU와 memory를 쓰면서 wait)
 $(= \text{spin lock})$

• 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  
}
```

• 내가 lock을 걸고 critical section에 들어갈 때 lock을 Test_and_Set 하는
• Instruction으로 물리적 동시에 lock을 바꿔나와
• lock이 0이면 lock을 1로 바꾸고 동시에 while문을 빠져나와
• lock이 1이면 대체 lock을 1로 세팅하고 계속 while문을 돌고 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

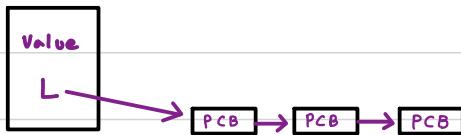
```
typedef struct
{
    int value;           /* semaphore */
    struct process *L;  /* process wait queue */
} semaphore;
```

- 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)

* 여기서 S.value는 busy wait 이후의 값을 다룬다. busy wait 아닌 자유한방 가능 예제를 나타냈지만 S.value 기준의 Value 값은 누군가가 잠들어 있다는 뜻

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

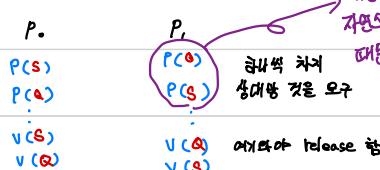
↳ 주로 mutual exclusion(lock/unlock에 사용)

Deadlock and Starvation

- Deadlock

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

- S와 Q가 1로 초기화된 Semaphore 라 차자



- Starvation

↳ indefinite blocking. 프로세스가 Suspend된 이유에 해제하는

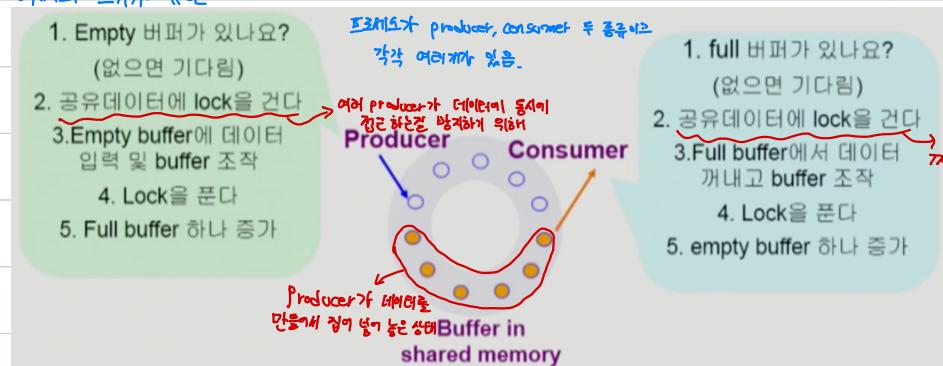
 Semaphore 큐에서 빠져 나갈 수 없는 현상.

Classical Problems of Synchronization

- Bounded-Buffer Problem (producer-consumer problem)
- Readers and writers problem
- Dining-Philosophers Problem

Bounded-Buffer Problem (producer-consumer problem)

버퍼의 크기가 유한



- Shared data

- buffer 자체/ 및 buffer 조작 변수 (empty/full buffer의 시작위치)

- Synchronization variables

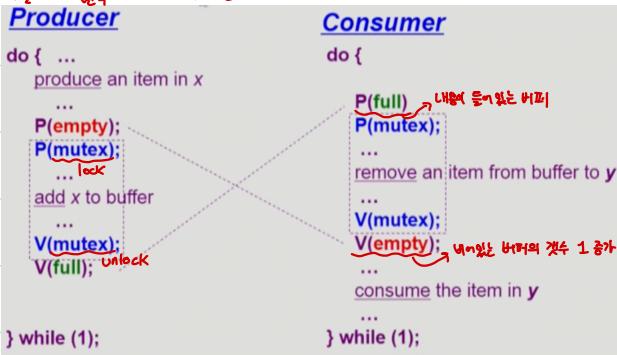
- mutual exclusion \rightarrow Need binary semaphore
(shared data의 mutual exclusion을 위해)
- resource count \rightarrow Need integer Semaphore
(남은 full/empty buffer의 수 표시)

Bounded - Buffer Problem

- Synchronization Variables

Semaphore full = 0, empty = 1, mutex = 1;
내부에 들어있는 buffer를
내부에 들어있는 mutex를
내부에 들어있는 lock을
제거하기 위해

세개의 변수.



Readers - Writers Problem

- 한 Process가 DB에 Write 중일 때 다른 process가 접근하면 안됨.
- read는 동시에 여러번 해도됨. writer는 안됨.

- Solution

- v Writer가 DB에 접근하기를 아직 일지 못한 상황에서는 모든 대기중인 Reader들을 다 DB에 접근하게 허용한다.
- v Writer는 대기 중인 Reader가 하나도 없을 때 DB접근이 가능하다.
- v 일단 Writer가 DB에 접근 중이면 Reader들은 접근이 금지된다.
- v Writer가 DB에서 빠져 나가자면 Reader의 접근이 가능된다.

- Shared data

- DB 자료
- readcount: 현재 DB에 접근 중인 Reader의 수

- Synchronization Variables

- mutex: 공유 변수 read count를 접근하는 코드 (critical section)의 mutual exclusion 보장을 위해 사용
- db: Reader와 Writer가 공유 DB 자체를 올바르게 접근하게 하는 역할

Readers - Writers Problem

- Shared data

```
int readcount = 0;
DB자체; 흡수
```

- Synchronization variables

```
Semaphore mutex = 1, db = 1;
```

<u>Writer</u>
P(db);
...
writing DB is performed
...
V(db);

readcount가 0은 공유 변수
위에 lock

Reader

P(mutex);
readcount++;
if (readcount == 1) P(db); /*block writer*/
V(mutex);
...
reading DB is performed
...
P(mutex);
readcount--;
if (readcount == 0) V(db); /*enable writer*/
V(mutex);

다른 reader들도 읽을 수 있어야 하니까 lock을 안 권하고 read하면 writer가 들어와 값을 바꿀 수 있기 때문에
readcount라는 공유 변수를 만들어서 read count가 0 → 1 같아
db에 lock을 걸고 readcount가 1 → 2 같아
화면 끝 때는 lock을 풀지 않고 그냥 돌아
와서 읽으면 된다.

! Starvation 발생 가능

위 코드는

reader가 읽는 중이 writer, reader 습으로 뭉개운다면
reader가 읽는 동안 그 동안은 writer들이 있는 reader가 먼저
나가게 접근하고 writer들은 어떻게 빠져드는 모든 reader가 나가고
lock을 끌어야 접근할 수 있다. 그런데 lock이 끊기 전에
계속 reader가 들어오면 writer에게 starvation 발생

* Starvation 해결 가능한 코드를 한번 생각해 보자!

⇒ queue에 우선순위를 두어서 writer가 일정연례로 기다리지 않도록 구현.

Dining - Philosophore Problem

- Synchronization variables

```
Semaphore Chopstick[5];
Initially all values are 1 ~
```

- philosopher :

```
do {
    P(chopstick[i]); → 왼쪽 젓가락
    P(chopstick[(i+1)%5]); → 오른쪽 젓가락
    ...
    eat();
    V(chopstick[i]); → 다 먹었으면
    V(chopstick[(i+1)%5]); → 젓가락 놓음
    ...
    think();
    ...
} while(1);
```



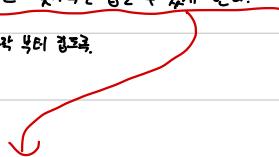
Dining-Philosopher Problem

- 앞의 Solution의 문제점

- v Deadlock 가능성이 있다.
- v 모든 철학자가 동시에 배가 고파져 원격 젓가락을 잡기 어렵 경우

- 해결 방안

- v 4명의 철학자만이 테이블에 동시에 앉을 수 있도록 한다.
- v 젓가락을 두 개 모두 잡을 수 있을 때에만 젓가락을 잡을 수 있게 한다.
- v 비단계적
· 짹수(혹은) 철학자는 원칙(오로지) 젓가락 뿐만 경고록.



```
enum {thinking, hungry, eating} state[5];
semaphore self[5]=0;
semaphore mutex=1;

Philosopher i
do {
    pickup(i);
    eat();
    putdown(i);
    think();
} while(1);

void pickup(int i) {
    P(mutex);
    state[i] = hungry;
    test((i+4) % 5);
    test((i+1) % 5);
    V(mutex);
}

void putdown(int i) {
    P(mutex);
    state[i] = thinking;
    test((i+4) % 5);
    test((i+1) % 5);
    V(mutex);
}

void think() {
    if (state[(i+4)%5]==eating && state[i]==hungry
        && state[(i + 1) % 5] != eating) {
        state[i] = eating;
        V(self[i]);
    }
}
```

Semaphore의 문제점

- 코딩하기 힘들다.
- 정확성 (correctness)의 암증이 어렵다.
- 자발적 협력 (voluntary cooperation)이 필요하다.
- 한 번의 실수가 모든 시스템에 치명적 영향

- 예

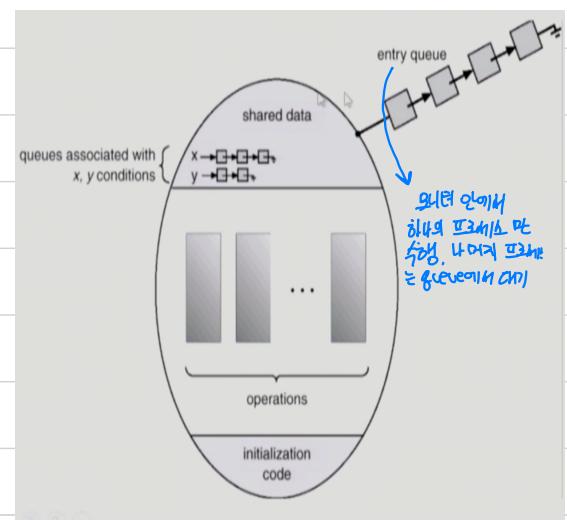


Monitor → 프로그래밍 언어 차원에서 동기화와 관련된 공유데이터를 접근하는 문제를 모니터가 자동으로 해결해 주도록 함으로써 프로그래머의 부담을 확연히 줄여주는 방법

→ lock을 걸 필요가 없다.

동시 수행중인 프로세스 사이에서 abstract data type의 안전한 공유를 보장하기 위한 high-level synchronization construct

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...){
        ...
    }
    procedure body P2 (...){
        ...
    }
    procedure body Pn (...){
        ...
    }
    {
        initialization code
    }
}
```



Monitor

- 모니터 내에서 한 번에 하나의 프로세스만이 활동 가능
- 프로그래머가 동기화 제약 조건을 명시적으로 코딩할 필요 없음
- 프로세스가 모니터 안에서 기다릴 수 있도록 하기 위해
Condition Variable 사용

Condition x,y;

Semaphore처럼
값을 가지는 변수처럼
증가와 감소를
하면서 큐의 변수

- Condition Variable은 Wait 와 Signal 연산에 의해만 접근 가능

x.wait(); → 잠들게 하기

x.wait() 을 invoke 한 프로세스는 다른 프로세스가

x.signal() 을 invoke 하기 전까지 Suspend 된다.

x.signal(); → 어떤 프로세스가 x를 다 쓰고 빠져 나가면서 잠들어 있는 프로세스가 있으면

그 중에 하나를 깨워주는 것

x.signal() 은 정확하게 하나의 suspend 된 프로세스를 resume 한다.

Suspend 된 프로세스가 없으면 아무일도 일어나지 않는다.

Bounded Buffer Problem

```
monitor bounded_buffer
{
    int buffer[N];
    condition full, empty;
    /* condition var.은 값을 가지지 않고 자신의 큐에 프로세스를
       매달아서 sleep 시키거나 큐에서 프로세스를 깨우는 역할만 함 */
    void produce (int x)
    {
        if there is no empty buffer
            empty.wait(); → 인 버퍼를 기다리는 줄에 세우고
        add x to an empty buffer → 큐에 넣어주는 프로세스
        full.signal();
    }

    void consume (int *x)
    {
        if there is no full buffer
            full.wait(); → 큐에 버퍼에 data를 꺼내주는
        remove an item from buffer and store it to *x
        empty.signal();
    }
}
```

일반적인 프로그래머 입장에선 Monitor가 더 자연스럽고 수월하다.

Monitor 버전의 코드와 Semaphore 버전의 코드는 쉽게 구분할 수 있다.

Semaphore에서는 생산자가 공유 buffer에 데이터를 집어 넣기 위해서는 buffer 전자를 lock 을 걸어서 다른 생산자, 소비자가 접근을 못하게 한 다음에 만들었을 데이터를 집어 넣어야 했고 소비자도 데이터를 꺼내기 위해 다른 프로세스의 버퍼 접근을 막고 버퍼에서 데이터를 꺼내갔다면 Monitor에서는 생산자는 소비자든 코드를 실행하는 도중에 다른 프로세스의 접근을 Monitor가 막아주기 때문에 공유 buffer에 lock 을 걸거나 풀여주는 코드가 나와 필요 없다.

Dining philosophers Example

```
monitor dining_philosopher
```

```
{  
    enum {thinking, hungry,  
          eating} state[5];  
    condition self[5];  
    void pickup(int i) {  
        state[i] = hungry;  
        test(i);  
        if (state[i] != eating)  
            self[i].wait(); /*wait here*/  
    }  
  
    void putdown(int i) {  
        state[i] = thinking;  
        /* test left and right neighbors */  
        test((i+4) % 5); /*if L is waiting*/  
        test((i+1) % 5);  
    }  
}
```

고지 변수
본인 및 본인 상태를
바꿀 수 있는지 여부
인접한 편의로 바꿀 수 있는지 여부

```
void test (int i) {  
    if ( (state[(i + 4) % 5] != eating) &&  
        (state[i] == hungry) &&  
        (state[(i + 1) % 5] != eating)) {  
            state[i] = eating;  
            self[i].signal(); /*wake up Pi */  
        }  
}  
void init() {  
    for (int i = 0; i < 5; i++)  
        state[i] = thinking;  
}
```

Each Philosopher:

```
{  
    pickup(i);  
    eat();  
    putdown(i);  
    think();  
} while(1)
```

Enter monitor

Pick UP() 고는
Monitor 내부에

Semaphore 사용하는 lock 을 걸고
Monitor OT/IC는 강제로 한다!