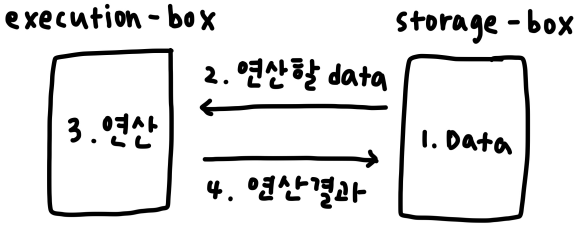
5️⃣ 병행 제어1

데이터의 접근



항상 연산할 data를 읽어와서 연산 결과를 저장

Race Condition

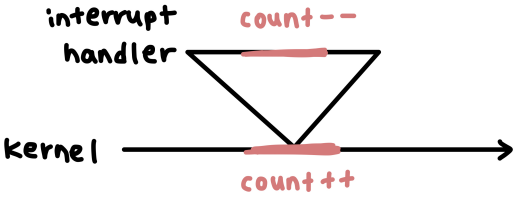
: 데이터를 여러 군데에서 읽어들여서 사용하면서 생기는 문제 → CPU가 여러 개일 때 생길 수 있음

ex. 한쪽에서 변수에 +1하고 저장, 다른 한 쪽에서 -1하고 저장

- 프로세스의 경우 변수가 프로세스 내부에 저장되어 있어 프로세스 내부 주소로 접근하기에 발생하지 않음
- **운영체제가 끼어드는 경우** 시스템 콜로 프로세스가 넘어가기 때문에 운영체제에 있는 데이터를 똑같이 건드려야 할 때 발생

OS에서 언제 race condition이 발생하는가

- **kernel 수행 중 인터럽트** 발생 시



  커널모드 running 중 interrupt가 발생하여 인터럽트 처리 루틴 수행

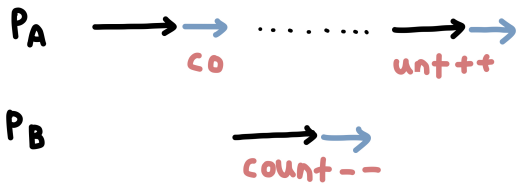
  → 양쪽 다 커널 코드이므로 kernel address space 공유

  → 변수를 건들일 때에 interrupt disable 시켜야 함

- process가 **system call**하여 kernel mode를 수행 중인데 **context switch**가 일어난 경우

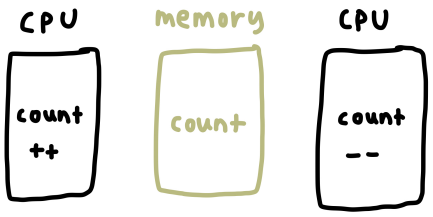
  → system call 하는 동안에 *kernel address space의 data를 access하게 됨* = share

  → 이 작업 중간에 CPU preempt하면 race condition 발생



⇒ 커널 모드에서 수행 중일 때는 CPU preemptive하지 않도록 함, 커널에서 사용자 모드 돌아갈 때 preempt

- multiprocessor에서 **shared memory내의 kernel data**



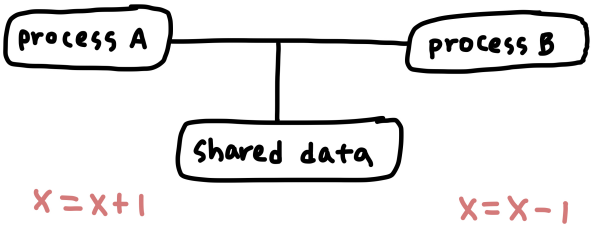
→ 한 번에 하나의 CPU만이 커널에 들어갈 수 있게 하는 방법 = *운영체제 자체가 lock*이 되는 방법

→ 커널 내부에 있는 각 공유 데이터에 접근할 때마다 그 데이터에 대한 lock/unlock하는 방법

Process synchronization

공유 데이터의 **동시 접근**은 **데이터의 불일치 문제**(inconsistency)를 발생시킬 수 있음

→ 일관성 유지를 위해서는 협력 프로세스 간의 실행 순서를 정해주는 매커니즘 필요



~ process A 실행 도중에 context switch 발생해서 process B 수행하면 문제가 생김

Critical-Section Problem

n개의 프로세스가 공유 데이터를 동시에 사용하기를 원하는 경우

→ 각 프로세스의 code segment에는 **공유 데이터를 접근하는 코드**인 **critical section**이 존재

⇒ 하나의 프로세스가 critical section에 있을 때 다른 모든 프로세스는 critical section에 들어갈 수 없어야 함

- 프로세스들의 일반적인 구조 → 공유 데이터의 접근 / 접근 X의 반복

```
do {
    entry section
    critical section
```

```
    exit section
    remainder section
} while(1)
```

- 프로세스들은 수행의 동기화를 위해 몇몇 변수를 공유할 수 있다 → **synchronization variable**

CS 문제 해결 알고리즘 1

- *synchronization variable* : int turn

  → **누구 차례인지 알려주는 변수**로 동시 접근을 막기 위해 사용

Process P_0

```
do{
    while (turn != 0);
    critical section
    turn = 1;
    remainder section
} while (1);
```

Process P_1

```
do{
    while (turn != 1);
    critical section
    turn = 0;
    remainder section
} while (1);
```

⇒ mutual exclusion but not progress

과잉양보 : 반드시 한 번씩 교대로 들어가야 함 → *상대방이 turn을 내 값으로 바꿔줘야만 내가 들어갈 수 있음*

특정 프로세스가 더 빈번히 critical section에 들어가야 하면 상대가 바꿔줄 때까지 기다려야 함

프로그램적 해결법의 충족 조건

- **mutual exclusion** (상호 배타적)

  프로세스 P_i가 critical section 부분을 수행 중이면 다른 모든 프로세스들은 그들의 *critical section에 들어가면 안된다*

- **progress** (진행)

  아무도 critcal section에 있지 않은 상태에서 critical section에 들어가고자 하는 프로세스가 있으면 *critical section에 들어가게 해주어야 한다*

- **bounded waiting** (무한 대기)

  프로세스가 critical section에 들어가라고 요청한 후부터 그 요청이 허용될 때까지 다른 프로세스들이 *critical section에 들어가는 횟수에 한계가 있어야 한다* → starvation 막아야 함

CS 문제 해결 알고리즘 2

- *synchronization variable* : boolean flag[2]

  → flag가 true면 critical section 사용을 요청

Process P_i

Process P_j

```
do {
    flag[i] = true;
    while (flag[j]);
    critical section
    flag[i] = false;
    remainder section
} while (1);
```

```
do {
    flag[j] = true;
    while (flag[i]);
    critical section
    flag[j] = false;
    remainder section
} while (1);
```

⇒ mutual exclusion but not progress requirement

둘 다 2행까지 수행 후 끊임없이 양보하는 상황 발생 가능 → *while문 돌기 전 CPU 빼앗기면 둘 다 true여서 못 들어감*

CS 문제 해결 알고리즘 3 (Peterson's algorithm)

- *synchronization variable* : int turn , boolean flag[2]

  → 동시에 flag가 true여도 turn으로 순서를 정함

Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
} while (1);
```

Process P_j

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);
    critical section
    flag[j] = false;
    remainder section
} while (1);
```

⇒ 세 개 조건 모두 충족

but busy waiting (= spin lock) → *계속 CPU와 memory를 쓰면서 while문* 돌면서 체크 wait

Synchronization hardware

하드웨어적으로 test & modify를 **atomic**하게 수행할 수 있도록 지원하는 경우 앞의 문제 간단히 해결

→ 데이터를 읽어가는 것과 저장하는 것 모두 한꺼번에 수행

- *synchronization variable* : boolean lock = false

  → false이면 아무도 critical section에 들어가지 않음

```
do {
    while (Test_and_Set(lock));
    critical section
    lock = false;
```

```
    remainder section
} whlie (1);
```

Semaphore

추상자료형으로 semaphore S 변수는 **정수로 정의**됨

*두 가지 atomic 연산*에 의해서만 접근 가능

P(S)

```
while (S <= 0) do no-op;
S--;
```

→ 자원을 획득하는 과정

S ≤ 0일 때 자원의 여분이 없어서 계속 while문을 돔

V(S)

```
S++;
```

→
자원을 반납하는 과정

- P(S)와 V(S) 사이에는 자원을 쓰는 과정이 담긴 코드 생략
- 자원의 개수 S = 1일 때는 P(S)가 lock을 거는 과정, V(S)가 lock을 푸는 과정
- ex. S = 5 : 자원이 5개 존재, 자원을 획득하기 위해 P연산을 먼저 하고 반납할 때는 V연산

Critical section of n Process

- *synchronization variable* : semaphore mutex = 1

  → 1개만 critical section에 들어갈 수 있음

```
do {
    P(mutex);
    critical section
    V(mutex);
    remainder section
} while (1);
```

⇒ busy wait 발생

block / wakeup (=sleep lock) Implementation

Semaphore 구조

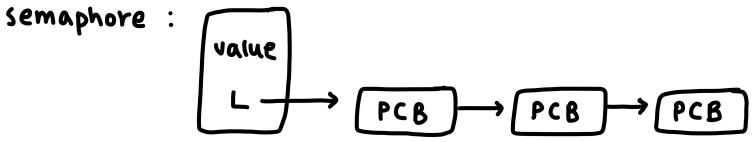
```
typedef struct {
    int value;
    struct process *L;
} semaphore
```

- **block**

: 커널은 block을 호출한 프로세스를 suspend시킴 → 이 프로세스의 PCB를 semaphore에 대한 wait queue에 넣음

- **wakeup(P)**

  : block된 프로세스 P를 wakeup시킴 → 이 프로세스의 PCB를 ready queue로 옮김



→ 자원 여분이 없으면 프로세스들을 blocked 시켜서 L에 줄 세워서 기다리게 함

P(S)

```
S.value--;
if (S.value < 0) {
    add this process to S.L
    block();
}
```

V(S)

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

P에서 먼저 value-- 시키기 때문에 ≤0 일때의 조건

- *critical section의 길이가 긴 경우* block/wakeup이 적당
- *critical section의 길이가 매우 짧은 경우* block/wakeup 오버헤드가 busy-wait 오버헤드보다 더 커질 수 있음
- 일반적으로는 block/wakeup이 더 좋음

Two types of Semaphore

- counting semaphore

  : 도메인이 0 이상인 임의의 정수 값 → 주로 resource counting에 사용

- binary semaphore (=mutex)

  : 0 또는 1의 값만 가질 수 있는 semaphore → 주로 mutex exclusion (lock / unlock)에 사용