

Process Synchronization

❖ 왜 동시성 이슈가 발생하는가? (Single Processor라고 가정한다.)

1. Process Scheduling에 의해 발생하게 된다.

- Process Scheduling이 발생하지 않는다면, 즉 하나의 작업을 계속해서 수행한다면, 당연히 동시성 이슈가 발생하지 않는다.
- 하지만, 이는 너무나 비효율적이다.

2. Process Scheduling은 "Interrupt → Interrupt Service Routine(ISR)"에 의해 수행된다.

3. 따라서, **Interrupt**에 의해 동시성 이슈가 발생한다고 말할 수 있다.

❖ 동시성 이슈 해결책은? :: version 1

Step1

Critical Section을 수행하는 도중에는 interrupt를 disable시킨다.

즉, interrupt가 발생하지 않도록 한다.

Step2

이후, 작업을 완료하면 interrupt를 enable시키면 된다.

참고) H/W가 지원해주는 기능이다.

```
void main (void){  
    int iTemp0, iTemp;  
  
    while(TRUE) {  
        disable(); /*Disable interrupts using array*/  
        iTemp0 = iTemperatures[0];  
        iTemp1 = iTemperatures[1];  
        enable();  
        if (iTemp0 != iTemp1)  
            !! set off howling alarm;  
    }  
}
```

Semaphore

❖ 동시성 이슈 해결책 version 1 문제점

Interrupt를 disable한다는 것은 경쟁하는 Thread 뿐만 아니라, 모든 Thread를 Stop 시키는 것을 의미한다.
따라서, 비효율 적이다. → [Semaphore를 통해 해결](#)

Semaphore 구성

① Semaphore

- 사실상 INT 변수라고 생각하면 된다.
- Semaphore 변수 하나 당,

```
struct Semaphore {  
    int cnt;  
    Queue queue;  
}  
  
semaphore S1 = 1;  
semaphore S2 = 1;
```

↳ 두 종류의 Critical Section 생성
즉, “두개의 방”을 생성했다고 생각

② P() 연산

Lock 요청 or 없으면 wait()

```
P(S) {  
    disableInterrupts();  
    if(S.cnt-- > 0)  
        enableInterrupts();  
    else  
        sleep(S.queue);  
}  
  
sleep(Q) {  
    // cur_p is the current proc  
    enqueue(cur_p, Q);  
    enableInterrupts();  
    yield_cpu();  
}
```

② V() 연산

Lock 반납 and 대기 Process 깨운다.

```
V(S) {  
    disableInterrupts();  
    if(S.cnt++ >= 0) {  
        enableInterrupts();  
    } else  
        wakeup(S.queue);  
}  
  
wakeup(Q) {  
    p = dequeue(Q);  
    enableInterrupts();  
    // execute p immediately  
    reschedule(p);  
}
```

Semaphore

② 따라서, wait() 오퍼레이션이라고 불린다.

```
P(S) {  
    disableInterrupts();  
    if(S.cnt-- > 0)  
        enableInterrupts();  
    else  
        sleep(S.queue);  
}
```

Semaphore도 결국 H/W가 지원하는 기능을 사용할 수 밖에 없다.
하지만, disable interrupt 구간을 상당히 짧게 줄였다.

⚠ 따라서, Critical Section이 정말 작은 경우, H/W의 기능을 직접 사용하는 것이 더 좋다.
왜냐하면, Semaphore는 sleep 때문에, Process Scheduling 까지 발생한다.

```
sleep(Q) {  
    // cur_p is the current proc  
    enqueue(cur_p, Q);  
    enableInterrupts();  
    yield_cpu();  
}
```

① Semaphore를 Scheduling에 사용 할 수 있다.

V(S) { → ③ 반대로, signal() 연산이라고 불린다.

```
    disableInterrupts();  
    if(S.cnt++ >= 0) {  
        enableInterrupts();  
    } else  
        wakeup(S.queue);  
}
```

```
wakeup(Q) {  
    p = dequeue(Q);  
    enableInterrupts();  
    // execute p immediately  
    reschedule(p);  
}
```

mutex_lock 의 경우 이 부분은

- sleep 대신 spin lock 방식을 사용한다.
- 즉, 계속 CPU를 점유하며 열쇠 획득까지 무한루프

Semaphore ※내가 강조하고 싶은 부분 정리

Task1 () {

P (S1)

방 내부에서 하고싶은 행동

V (S1)

}

Task2 () {

P (S1)

같은 방에서 다른 행동 가능
즉, 같은 공유 자원에 대해 다른 행동 가능

V (S1)

}

Task3 () {

P (S2)

여기는 다른 방이다.
즉, 다른 공유 자원이 존재

V (S2)

}

Semaphore – Producer / Consumer

Producer () {

P (S1)
buf = data;
V (S2)

Consumer () {

P (S2)
data = buf;
V (S1)

1. Producer 방 진입 후 잠금

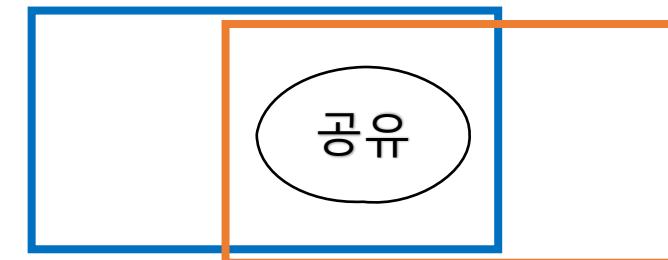
2. Consumer 방 Open

3. Consumer 방 진입 후 잠금

4. Producer 방 Open



Producer 방



Consumer 방