6. Process_Synchronization_2

Classical Problems of Synchronization

동기화 과정에서 발생하는 3가지 문제를 볼 예정

1. Bounded-Buffer Problem
2. Readers and Writers Problem: scheduling
3. Dining-Philosophers Problem: deadlock

Bounded-Buffer: Shared-Memory Solution

말 그대로 공유 데이터에 버퍼(배열)가 있는데 이걸 여러 스레드가 공유

이 배열에 접근하는 주체를 크게 2가지로 나뉜다

1. Producer
2. Consumer

Producer는 버퍼에 데이터를 채워넣는 역할이고

Consumer는 버퍼에 있는 데이터를 소모하는 역할

그러면 여기서 신경써야 되는 점은 뭘까?

⇒ Producer는 배열이 다 차 있는지(full-buf) 확인하고 넣어야하고

⇒ Consumer는 배열이 다 비어 있는지(empty-buf) 확인하고 빼야돼

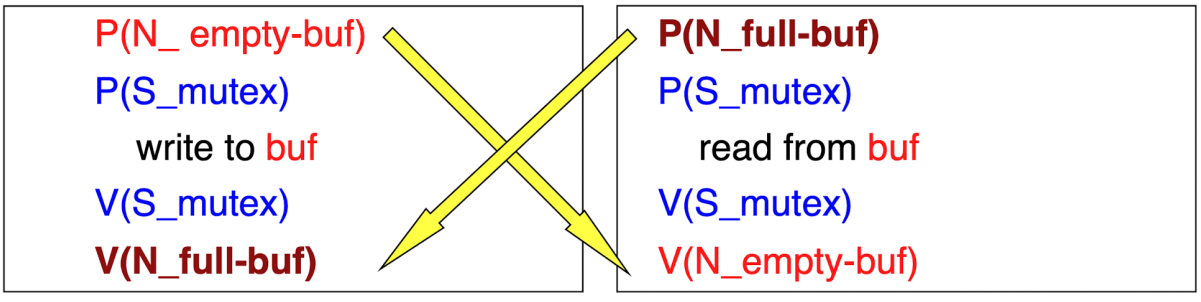
그래서 가장 간단한 해결방법은 배열이 다 차 있으면 Producer는 대기하고

배열이 비어있으면 Consumer가 대기하면 된다

이 상황을 이제 semaphore로 보게 되면

필요한 변수가

1. 비어있는지 확인하는 변수
2. 차있는지 확인하는 변수
3. lock



왼쪽이 producer, 오른쪽이 consumer

P ⇒ 값이 0보다 작거나 같으면 대기

V ⇒ 값을 1 증가시켜줌

producer는 empty_buf가 있는지 확인하고 할 일 다 끝내고 full-buf의 값을 1 증가 시켜줘

consumer는 full_buf가 있는지 확인하고 할 일 다 끝내고 empty-buf의 값을 1 증가 시켜줘

그럼 여기서 질문해볼 수 있는게 왜 lock을 별도로 만들고 empty-buf, full-buf를 확인하는 변수를 따로 둘까?

⇒ lock을 잡으려 드는 것 자체도 race condition 이기 때문에 lock을 잡기 위해서도 lock을 사용

그럼 만약 semaphore를 안쓰고 저걸 mutex로 관리한다면?

⇒ 이 때는 empty-buf, full-buf 없이 말 그대로 개수를 확인하는 count 변수를 두고 count 값이 full 이면 condition variable 을 통해 대기 하도록(block-wakeup)

Readers-Writers Problem

이거는 주로 DB에서 많이 쓰이는 기법인데 throughput을 올리기 위해서 사용

이걸 하기 전에 다시 근본적인 질문부터

왜 여러 프로세스가 동시에 접근하는 것이 문제일까?

⇒ 데이터가 의도하지 않은 값으로 변경될 수 있기 때문에

그러면 애초에 이 프로세스가 데이터를 변경하지 않는다 라는 것이 보장되어 있다면?

이러면 동시에 접근해도 괜찮지 않을까?

그래서 나온게 readers-writers problem

그래서 이 문제는 크게 2가지로 나뉘어

1. Reader-Writer
2. Reader-Reader

이 상황에서 writer는 무조건 단일로 접근해야 하기 때문에 mutex를 쓰고

reader는 동시에 접근할 수 있기 때문에 semaphore 사용

이거에 대한 해결책은 일단 2개

1. reader들이 동시에 들어가서 모든 reader가 다 끝나고 나면 그 때 writer 시작
2. writer부터 데이터 처리한 다음에 reader들이 진입

1번에 대해서 살펴보면, writer는 언제 진입해야 할까?

⇒ readcount가 0일 때

근데 이거 문제있어

⇒ readcount 값을 변경하는데 얘가 race condition이 되어버려(Reader-Reader problem)

그래서 readcount에 대해서도 lock이 필요해 이걸 mutex 라고 하자

그리고 지금 Reader-Writer 문제에 대해서 아직 해결을 안 했는데, 현재 db에 reader들과 writer가 접근하는 상황일 때 reader-writer 간에 동시 접근을 막아야 돼. 그래서 이걸 db 라고 하자

Writer

```
P(db);



// writing
```

Reader

```
P(mutex);
readcount++;
if (readcount == 1) {
    P(db);
}
// reading
```

```
V(db);
```

```
P(mutex);
readcount--;
if (readcount == 0) {
    V(db);
}
V(mutex);
```

readcount 값을 변경 시킬 때는 mutex를 걸고 변경함

db: reader, writer 간에 접근을 하나만 허용하도록 하는 lock

⇒ reader들 중 최초로 접근하는 reader는 db에 lock을 걸어줘야 돼, 반대로 마지막에 나오는 reader가 db lock을 풀어줘야 돼

⇒ writer는 어차피 단일로 밖에 접근을 못해서 writing 작업을 하기 전에만 lock을 걸어주면 돼

Dining-Philosophers Problem

이전의 문제들과는 다르게 이건 동기화에 대한 문제가 아니야

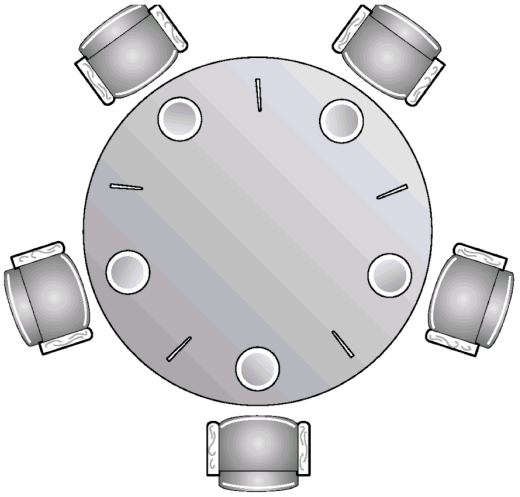
이건 deadlock이 발생하는 상황에 대한 문제

Shared data

```
semaphore chopstick[5];
// Initially all values are 1
```

Philosopher i

```
do {
    P(chopstick[i])
    P(chopstick[(i+1) % 5])
     ...
    eat
     ...
    V(chopstick[i]);
    V(chopstick[(i+1) % 5]);
     ...
    think
     ...
} while (1);     ==> deadlock may occur!
```



이런 상황에서 철학자들이 밥을 먹는대

근데 젓가락이 양 옆에 하나씩 밖에 없어서 젓가락을 공유하는 사이

모든 철학자들이 자기 왼쪽 젓가락만 점유했다고 가정하면, 그 누구도 밥을 먹지 못해

(젓가락 2개 다 들어야 먹을 수 있는데 자기 오른쪽 젓가락은 다른 철학자가 가져가버려서)

이렇게 아무 것도 못하는 상황이 데드락

그럼 이거 어떻게 해결해?

1. 철학자 한명 빼서 4명이서(단, 젓가락은 그대로)
2. 홀수 철학자들은 왼쪽 먼저, 짝수 철학자들은 오른쪽 먼저

3. 2개 드는 것을 보장하지 못하면 다시 내려놓음

Problems of Semaphore

1. 코드를 쓰기가 어려워
2. 동일한 문제를 재현하기 어려워 (스레드 동작 순서는 아무도 모르니까)
4. 남용하게 되면 성능 저하됨 (대기해야되는 프로세스의 수가 증가하니까)

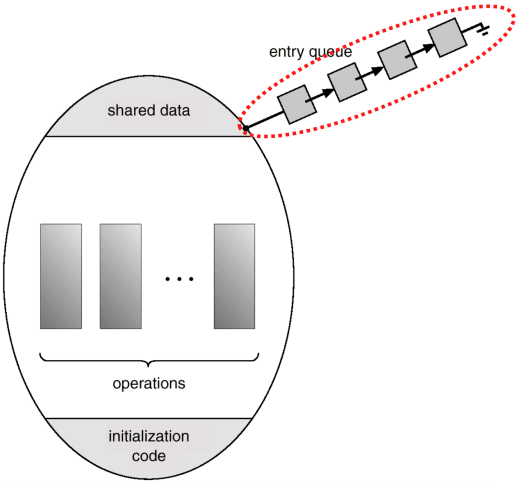
⇒ 그래서 이걸 개발자가 하지 말고 언어나 프레임워크 차원에서 처리를 하고 개발자는 키워드 딸깍으로 끝낼 수 있도록 하면 좋지 않을까? 해서 나온게 Monitor, 자바에서는 synchronized 키워드 쓰면 해결

Monitors

High-level language 에서 제공하는 synchronization

private 영역에 접근되는 데이터가 공유되는 데이터인데 이걸 개발자가 동기화하는데 신경 쓸 필요가 없어

(수업 때 쌤이 말했던 스레드를 직접 관리할 일이 없다는 게 이 소리인듯)

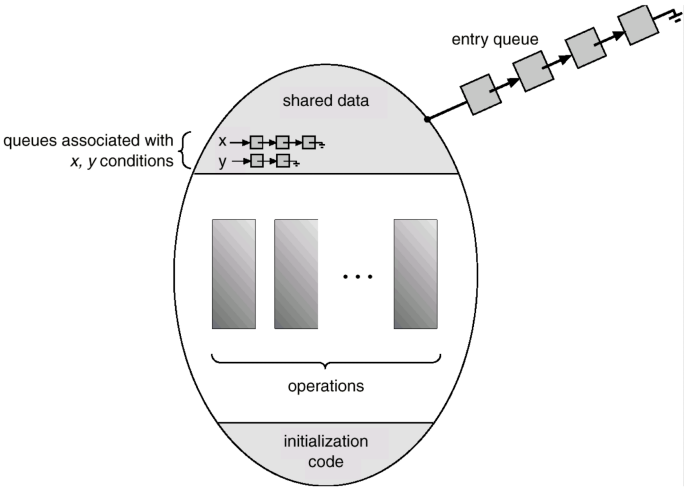


그림은 이렇게 생겨먹었는데, shared data에 접근하고자 하는 프로세스를 저렇게 주렁주렁 매달아 놔

이러려면 condition variable을 통해서 wait, signal을 쓸 수 있어

모니터 내부로 진입하게 되면 wait을 통해 프로세스가 sleep상태로 들어가

entry queue에 있는 프로세스가 모니터 내부로 진입할 때 이제 끝날(?) 프로세스가 맨 앞에 있는 프로세스를 signal로 깨워



그래서 shared data 내에서 발생하는 race condition 마다 condition variable을 둬서 관리할 수 있어

Monitor: Dining-Philosophers Problem

동시에 2개를 들도록 하는 상황

```
class dining_philosopher
{
    enum {thinking, eating, hungry} state[5];
    condition self[5];    // 여기서 sleep

    void pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating)
            self[i].wait(); // 아직 자원을 점유할 수 없어 => 나중에 signal로 깨워줄거야
    }

    void putdown(int i) {
        state[i] = thinking;
        test((i + 4) % 5); // 내 왼쪽 사람 확인
        test((i + 1) % 5); // 내 오른쪽 사람 확인
    }
}

// 잡으려는 의지가 있고, 양 옆이 젓가락을 소유하고 있지 않으면 내가 들어가
void test(int i) {
    if ((state[(i + 4) % 5] != eating) && (state[i] == hungry)
        && (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}

void init() {
    for (int i = 0; i < 5; i++) {
        state[i] = thinking;
    }
}
```