# 6 병행제어2

## Deadlock

둘 이상의 프로세스가 서로 상대방에 의해 충족될 수 있는 event를 무한히 기다리는 현상

ex. semaphore 둘 다 얻어야만 할 수 있는 작업 → *하나씩 차지하고 상대방 것 요구* = 계속 기다림

→ 자원을 얻는 순서를 정해놓으면 문제 해결 가능

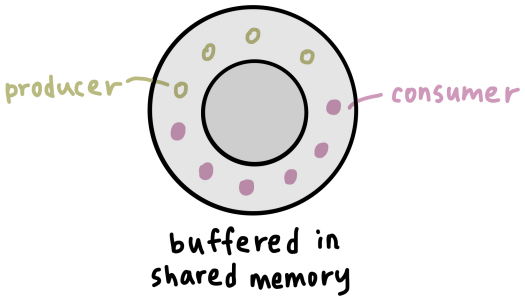
## Starvation

indefinite blocking → 프로세스가 suspend된 이유에 해당하는 세마포어 큐에서 빠져나갈 수 없는 현상

## Classical problems of Synchronization

### 1. Bounded-Buffer Problem (Producer - Consumer Problem)

크기가 유한한 공유버퍼




**Producer**

→ 데이터 만들어서 버퍼에 넣어줌

1. Empty 버퍼가 있는지 확인 → 없으면 기다림
2. 공유데이터에 Lock을 검
3. empty buffer에 데이터 입력 및 buffer 조작
   → pointer 다음 buffer 가리킴
4. lock을 품
5. full buffer 하나 증가

**Consumer**

→ 데이터 꺼내감

1. full buffer가 있는지 확인 → 없으면 기다림
2. 공유 데이터에 lock을 검
3. full buffer에서 데이터 꺼내고 buffer 조작
4. lock을 품
5. empty buffer 하나 증가

→ **shared memory**

*buffer 자체* 및 *buffer 조작 변수* (empty / full buffer의 시작 위치)

→ **synchronized variables**

- *mutual exclusion* : need binary semaphore → 공유 버퍼에 lock을 걸기 위해
- *resource count* : need integer semaphore → 남은 full / empty buffer의 수 표시

- *synchronized variables* : semaphore full = 0, empty = n, mutex = 1

**Producer**

```
do {
    produce an item in X
    P(empty); // 빈 버퍼 찾음
    P(mutex); // 공유 버퍼에 lock
    add X to buffer
    V(mutex);
    V(full); // 잠들어 있는 소비자 깨워줌
} while(1);
```

**Consumer**

```
do {
    P(full); // 내용 있는 버퍼 획득
    P(mutex);
    remove an item from buffer to y
    V(mutex);
    V(empty);
    consume the item in y
} while(1);
```

## 2. Readers and Writers Problem

한 프로세스가 DB에 *write 중일 때 다른 process가 접근하면 안됨*, *read는 동시에 가능*

→ writer가 DB 접근 허가를 아직 얻지 못한 상태에서는 모든 대기중인 reader들을 다 DB에 접근하게 해줌

→ writer는 대기 중인 reader가 하나도 없을 때 DB 접근이 허용된다

→ 일단 writer가 DB에 접근 중이면 reader들은 접근이 금지됨

→ writer가 DB에서 빠져나가야만 reader의 접근이 허용됨

- **shared memory**

  *DB 자체, readcount* (현재 DB에 접근 중인 reader의 수)

- **synchronization variables**
  - *mutex* → 공유 변수 readcount를 접근하는 코드의 mutual exclusion을 위해 사용
  - *db* → reader와 writer가 공유 DB 자체를 올바르게 접근하게 하는 역할 (lock을 거는 역할)

- *shared data* : int readcount = 0, DB 자체
- *synchronization variables* : semaphore mutex = 1, db = 1

Writer

```
P(db);
writing DB is performed
V(db);
```

Reader

```
P(mutex); // readcount에 대한 lock
readcount++;
if (readcount == 1) P(db); //DB lock
V(mutex);
reading DB is performed
P(mutex);
```

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

⇒ starvation 발생 가능

DB 접근해야 하는데 *reader들이 다 빠져나갈 때까지 writer 계속 기다려야 함*

→ 일정 시간까지 도착한 reader들만 동시 접근 가능하게 함

## 3. Dining-Philosophers Problem

- *synchronization variables* : semaphore chopstick[5]

```
do {
    P(chopstick[i]);
    P(chopstick[i+1] % 5);
    eat();
    V(chopstick[i]);
    V(chopstick[i+1] % 5);
    think();
} while(1);
```

⇒ deadlock이 생김

1. 4명의 철학자만이 테이블에 동시에 앉을 수 있도록 함
2. 젓가락을 두 개 모두 잡을 수 있을 때에만 젓가락을 잡을 수 있게 함
3. 비대칭 : 짝수 철학자는 왼쪽 젓가락부터 잡도록 함

**2번 방법**

- *synchronization variables*

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

  state → 철학자의 상태 / self → 젓가락 두 개 다 잡을 수 있는지 / mutex → 상태 변수에 대한 lock

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

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

```
void pickup(int i) {
    P(mutex);
    state[i] = hungry;
    test(i);
    V(mutex);
    P(self[i]);
}
```

```
void test(int i) {
    if (state[(i+4)%5] != eating && state[i] == hungry && state[(i+1)%5] != e
        state[i] = eating;
```

```
        V(self[i]); // V 연산으로 1(available) 만들어 줌
    }
}
```

## Semaphore의 문제점

- 코딩하기 힘들다
- 정확성의 입증이 어렵다
- 자발적 협력이 필요하다
- 한 번의 실수가 모든 시스템에 치명적 영향을 미침

⇒ **Monitor**

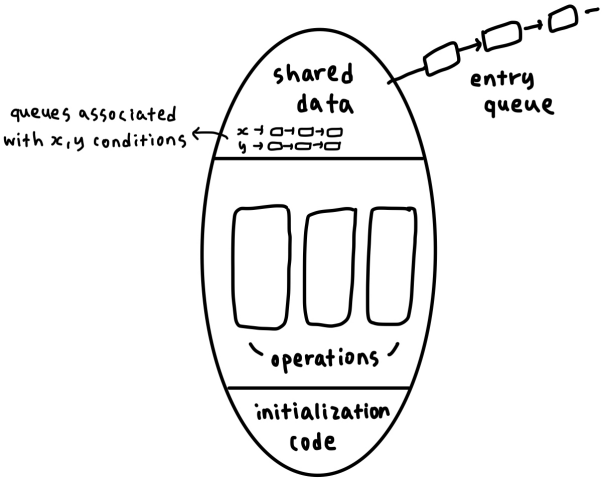
## Monitor

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

→ 공유 데이터를 중심으로 함수가 정의되어 있음

semaphore는 프로그램이 lock을 걸고 풀어야 했다면 monitor는 *공유 데이터의 접근은 monitor 내의 함수를 통해서만 가능*

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




→ 공유 데이터를 monitor 안에 정의

→ monitor가 알아서 공유데이터에 동시 접근하는 것을 막기 위해 entry queue에 넣음

- 모니터 내에서는 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

  → semaphore와 비슷해 자원 여분이 있으면 실행하며 queue의 역할
- condition variable은 wait와 signal 연산에 의해서만 접근 가능
  - **x.wait()** : x.wait()을 invoke한 프로세스는 다른 프로세스가 x.signal()을 invoke하기 전까지 suspend 됨

    → blocked된 상태로 여분이 없으면 queue에 줄 서서 잠듦
  - **x.signal()** : x.signal()은 정확하게 하나의 suspend된 프로세스를 resume함 suspend된 프로세스가 없으면 아무 일도 일어나지 않음

    → 잠들어 있는 상태에서 깨움

## 1. Bounded-Buffer Problem

```
monitor bounded_buffer {
    int buffer[N];
    condition full, empty;
    void produce(int x) {
        if there is no empty buffer
            empty.wait();
        add X to an empty buffer
        full.signal() // 잠들어 있는 프로세스 있으면 깨움
    }
    void consume(int *x) {
        if there is no full buffer
            full.wait();
```

```
        remove an item from buffer and store it to *x
        empty.signal();
    }
}
```

→ 직접 lock을 걸고 풀지 않아도 됨

→ condition은 값을 가지지 않고 자신의 큐에 프로세스를 매달아서 sleep 시키거나 큐에서 프로세스를 깨우는 역할을 함

## 2. Dining Philosophers Problem

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

```
void test(int i) {
    if (state[(i+4)%5] != eating && state[i] == hungry && state[(i+1)%5] != e
        state[i] = eating;
        self[i].signal();
    }
}
```

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

```
Each Philosopher: {
    pickup(i); //
    eat();
    putdown(i); // enter monitor
    think();
} while(1);
```