

두개의 프로세스가 있다고 가정

critical section 프로그램적 해결법의 충족 조건

mutual Exclusion : 프로세스 P_i 가 critical section부분을 수행 중이면 다른 모든 프로세스들은 그들의 c.s에 들어가면 안된다.

progress: 아무도 C.S에 있지 않은 상태에서 CS에 들어가고자 하는 프로세스가 있으면 그렇게 해주어야 한다

bounded waiting: 프로세스가 CS에 들어가려고 요청한 후부터 그 요청이 허용될 때까지 다른 프로세스들이 CS에 들어가는 횟수에 한계가 있어야 한다

가정

- 모든 프로세스의 수행 속도는 0보다 크다
- 프로세스들 간의 상대적인 수행 속도는 가정하지 않는다.

해결 - 어떻게 소프트웨어적으로 락을 잘 걸었다 풀 수 있는가

알고리즘 1

턴을 교대로

```
do {  
    while (turn != 0) # 체크 누구 차례인가(어느 프로세스)  
        critical section  
    turn = 1 # 턴을 변경해준다  
    remainder section  
} while(1)
```

progress 조건을 만족하지 못함!

알고리즘 2

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

process i랑 j 둘다 flag만 true로 바뀌고 CS에 못들어가고 기다릴 수 있음

즉 둘다 2행까지 수행 후 끊임 없이 양보하는 상황 발생 가능

알고리즘 3 (peterson's algorithm)

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

3가지 조건을 다 만족하나 Busy Waiting(=spin lock)!!! 계속 cpu와 메모리를 쓰면서 wait

하드웨어적으로 test & modify를 atomic하게 수행할 수 있도록 지원하는 경우 앞의 문제는 단단히 해결

- 데이터를 읽는거랑 처리하는 걸 하나의 인스트럭션으로 처리할 수 없기 때문에 문제 발생

ex) test and set ==> a라는 데이터의 값을 읽고 1로 바꾸는 것을 하나의 인스트럭션으로 처리

Semaphores

앞의 방식들을 추상화시킴

추상자료형 - 오브젝트와 오퍼레이션으로 구성

Semaphore S

- integer variable
- 아래의 두가지 atomic 연산에 의해서만 접근 가능

```
P(s): while (S <=0) do no-op;
S--; # 자원이 있으면 하나 가져가고 없음 while문을 돌면서 기다린다

V(s): S++ # 자원을 반납
```

그래도 busy-wait이긴 함

block & wakeup방식의 구현!! (sleep-lock)

semaphore를 다음과 같이 정의

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

```

P(s): S.value--;
if(S.value < 0)
{
    block();
}

V(s): S.value++;
if(S.value <= 0) {
    wakeup(P)
}

```

block과 wakeup을 다음과 같이 가정

block:

- Critical Section의 길이가 긴 경우 block/wakeup이 적당
- 짧은 경우 busy-wait이 나올수도
- 일반적으로는 B/W가 더 좋음

semaphore의 타입

- counting semaphore
 - 도메인이 0이상인 임의의 정수값
 - 주로 resource counting에 사용
- Binary semaphore
 - 0 또는 1만 가질 수 있음
 - 주로 mutual exclusion (lock/unlock)에 사용

DEADLOCK

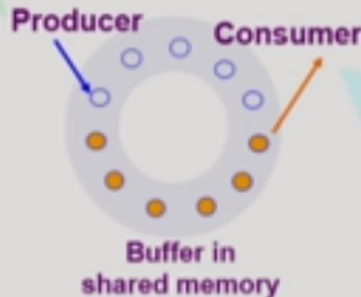
둘 이상의 프로세스가 서로 상대방에 의해 충족될 수 있는 event를 무한히 기다리는 현상

Bounded buffer problem

임시로 데이터를 저장하는 버퍼의 크기가 유한하기 때문에 발생

Bounded-Buffer Problem (Producer-Consumer Problem)

1. Empty 버퍼가 있나요?
(없으면 기다림)
2. 공유데이터에 lock을 건다
3. Empty buffer에 데이터
입력 및 buffer 조작
4. Lock을 푼다
5. Full buffer 하나 증가



1. full 버퍼가 있나요?
(없으면 기다림)
2. 공유데이터에 lock을 건다
3. Full buffer에서 데이터
꺼내고 buffer 조작
4. Lock을 푼다
5. empty buffer 하나 증가

Shared data

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

Synchronization variables

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

Readers-Writers Problem

Readers-Writers Problem

- ➔ 한 process가 DB에 write 중일 때 다른 process가 접근하면 안됨
- ➔ read는 동시에 여럿이 해도 됨
- ➔ solution
 - ✓ Writer가 DB에 접근 허가를 아직 얻지 못한 상태에서는 모든 대기중인 Reader들을 다 DB에 접근하게 해준다
 - ✓ Writer는 대기 중인 Reader가 하나도 없을 때 DB 접근이 허용된다
 - ✓ 일단 Writer가 DB에 접근 중이면 Reader들은 접근이 금지된다
 - ✓ Writer가 DB에서 빠져나가야만 Reader의 접근이 허용된다

Shared data

- ❖ DB 자체
- ❖ readcount; /* 현재 DB에 접근 중인 Reader의 수 */

Synchronization variables

- ❖ mutex /* 공유 변수 readcount를 접근하는 코드(critical section)의 mutual exclusion 보장을 위해 사용 */
- ❖ db /* Reader와 writer가 공유 DB 자체를 올바르게 접근하게 하는 역할 */

Dining-philosophers Problem

5명의 철학자는 생각하거나 / 밥을 먹거나 이걸 계속 반복

근데 밥을 먹으려면 왼쪽 오른쪽 젓가락을 다 잡아야됨

근데 왼쪽을 잡고 싶은데 옆에 애가 잡고 있다면 밥을 못먹겠지..

공유자원의 문제!

Dining-Philosophers Problem

Synchronization variables

```
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);
```



Dining-Philosophers Problem

- ➔ 앞의 solution의 문제점
 - ✓ Deadlock 가능성이 있다
 - ✓ 모든 철학자가 동시에 배가 고파져 왼쪽 젓가락을 집어버린 경우
- ➔ 해결 방안
 - ✓ 4명의 철학자만이 테이블에 동시에 앉을 수 있도록 한다
 - ✓ 젓가락을 두 개 모두 집을 수 있을 때에만 젓가락을 집을 수 있게 한다
 - ✓ 비대칭
 - 짝수(홀수) 철학자는 왼쪽(오른쪽) 젓가락부터 집도록

Dining-Philosophers Problem

```
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 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] != eating) {  
        state[i] = eating;  
        V(self[i]);  
    }  
}
```