Process Synchronization 1, 2

상태	완료
생성 일시	@2023년 7월 17일 오후 1:25
완성일	@2023년 7월 17일
유형	정리

임계 구역 문제를 해결하기 위한 충족 조건

- 상호 배제 (Mutual Exclusion)
  - 어떤 프로세스가 임계 구역 부분을 수행 중이면 다른 모든 프로세스는 그의 임계 구역에 들어가면 안 된다.
- 진행 (Progress)
  - 아무도 임계 구역에 있지 않은 상태에서 임계 구역에 들어가고자 하는 프로세스가 있으면 임계 구역에 들어가게 해 주어야 한다.
- 유한 대기 (Bounded Waiting)
  - 프로세스가 임계 구역에 들어가려고 요청한 후부터 그 요청이 허용될 때까지 다른 프로세스들이 임계 구역에 들어가는 횟수에 한계가 있어야 한다.
  - ex) 세 개의 프로세스가 있을 때 두 개의 프로세스만 번갈아가며 임계 구역에 들어가는 것은 유한 대기 조건을 만족하지 못한 것이다.

  즉 특정 프로세스 입장에서 임계 구역을 들어가지 못하고 지나치게 기다리면 안 된다는 것임

위 조건은 모든 프로세스의 수행 속도는 0보다 크며, 프로세스 간의 상대적인 수행 속도는 가정하지 않는다.

임계 구역 문제 해결 알고리즘

Algorithm 1

Synchronization variable

```
int turn;
initially turn = 0;// 몇 번 프로세스가 들어갈 수 있는지를 알려주는 turn 변수
```

Process P0

turn 변수가 0이 아닌 동안 while문을 계속 돌면서 자기 차례를 기다린다.

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

mutual exclusion은 만족하지만 progress는 만족하지 않는다. 반드시 한 번씩 교대로 임계 구역에 들어가야 하기 때문이다.

만약 프로세스 0은 임계 구역에 5번 들어가야 하고 프로세스 1은 한 번만 들어가면 된다고 가정해 보자. 프로세스 1은 한 번 임계 구역에 들어가고 나오면 끝나지만, 프로세스 0은 프로세스 1이 다시 들어갈 때까지 본인도 못 들어가므로 progess 조건이 성립하지 않는다. (과잉 양보)

Algorithm 2

Synchronization variable

```
boolean flag[2];
initially flag[모두] =false; //critical section에 들어가고자 하는 의사 표시
```

Process Pi

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

상대방이 임계 영역에서 빠져나왔을 때 내가 들어간다. (상대방 flag 확인 true라면 임계구역에 들어가고 싶어하네.. 나는 기다려야겠다. 상대방이 끝날 때까지)

mutual exclusion은 만족하지만, progress는 만족하지 않는다. 프로세스 A가 임계 구역에 들어가려고 flag를 true로 바꾼 상황에서 context switch가 발생(CPU를 뺏김)하여 프로세스 B에게 CPU가 넘어갔다고 하자. 이때 프로세스 B도 flag를 true로 바꿨는데 다시 context switch가 발생(CPU를 뺏김)하여 프로세스 A가 CPU를 잡는다면 그 누구도 임계 구역에 들어갈 수 없다. : 둘 다 들어가겠다고 깃발만 들고 아무도 들어가지 않았고, 들어가지도 못하는 상황이다.

Algorithm 3 (Peterson's Algorithm)

Combined synchronization variables of algorithm 1 and algorithm 2

위 두 개의 알고리즘에서 사용한 변수를 두 개 다 사용함

Process Pi

```
do {
    flag[i] = true;// 내가 임계 구역에 들어가고 싶다고 알림
    turn = j;// 자신의 다음 차례를 프로세스 j로 바꿔 줌
    while (flag[i] && turn == j); // 상대방이 깃발을 들고 있고, 턴이 상대방 차례인가 확인
    critical section
    flag[i] = false;
    remainder section
}while (1);
```

상대방이 임계 구역에 들어가 있지 않고, 들어갈 준비도 하지 않는다면 내가 들어간다.

세 조건을 모두 만족하지만, 계속 CPU와 메모리를 쓰면서 기다리기 때문에 busy waiting (spin lock)이 발생한다. 쉽게 말해 임계 구역에 들어가려면 상대방이 CPU를 잡고 flag 변수를 false로 바꿔주어야 하는데, 내가 CPU를 잡고 있는 상황에서 의미 없이 while 문을 돌며 CPU 할당 시간을 낭비해야 한다. (어차피 상대방이 끝날 때 까지는 조건 만족도 못 하는데 계속 반복)

+중간중간 CPU를 빼앗길 수도 있다는 가정하에 코드를 작성하다 보니 이러한 복잡한 코드가 완성된 것이다.

C++ 예시

```
#include <iostream>
#include <thread>
usingnamespace std;

int cnt;
bool flag[2] = { false, false };
int turn = 0;

void func0() {
```

```
for (int i = 0; i < 10000; i++) {
        flag[0] = true;
        turn = 1;
        while (flag[1] == true && turn == 1) {}

        cnt++;
        printf("cnt1 :: %d\n", cnt);

        flag[0] = false;
    }
}
void func1() {
for (int i = 0; i < 10000; i++) {
        flag[1] = true;
        turn = 0;
        while (flag[0] == true && turn == 0) {}

        cnt++;
        printf("cnt2 :: %d\n", cnt);

        flag[1] = false;
    }
}

intmain() {
    threadt1(func0);
    threadt2(func1);

    t1.join();
    t2.join();

    cout << "cnt : :" << cnt << endl;

return 0;
}
```

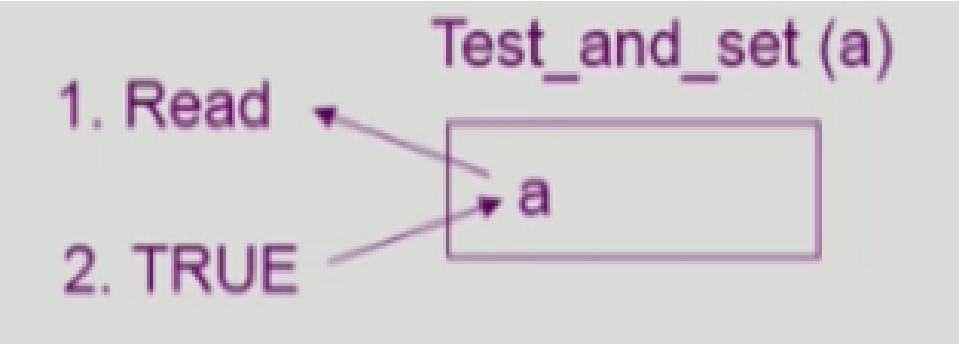
fun0만 보면 fun1은 자동으로 이해가 된다.

1. flag[0] = true로 설정하여 0번 스레드가 임계 영역 진입을 하고 싶다고 표시한다.
2. 이때 turn = 1으로 설정하여, 1번 프로세스가 먼저 임계 영역에 들어가라고 양보한다.
3. 만약 이때 context switch가 되지 않았다면 while문에 갇히게 된다.
4. 1번 프로세스가 모든 작업을 끝내면 turn = 0, flag[1] = false가 되므로 0번 프로세스가 임계 구역에 진입할 수 있다.

동기화 하드웨어

하드웨어적으로 Test&modify를 atomic하게 수행할 수 있도록 지원하는 경우 앞의 문제는 간단히 해결

임계 구역 문제가 발생한 근본적인 이유는 데이터를 읽고 쓰는 동작을 하나의 명령어로 수행할 수 없기 때문이다.(읽는 중에 빼앗기고 쓰는 중에 빼앗기고) 따라서 명령어 하나만으로 데이터를 읽는 작업과 쓰는 작업을 `atomic` 하게 수행하도록 지원하면 앞선 임계 구역 문제를 간단하게 해결할 수 있다.



위 그림은 a 변수를 읽은 후, 그 변수를 무조건 1로 설정하도록 명령어가 구성되어 있다.

ex Test_and_set(a) 를 수행 시 a가 0이면 1이 되고 1이었다면 1로 다시 셋팅 : 원래값을 읽어내고 그 자리에다 1로 셋팅하는 두 가지 작업을 atomic 하게 하나의 명령으로 처리하는 함수임

Mutual Exclusion With Test & Set

Synchronization variable

```
booleanlock = false; //0
```

Process Pi

```
do {
    while (Test_and_Set(lock));
    critical section
    lock = false;
    remainder section;
}while (1);
```

`Test_and_Set()` 함수는 매개 변수를 읽어내고, 그 변수를 1로 바꿔 주는 역할을 한다. 위 예시에서 lock을 읽고 난 뒤 1로 바꿔준다.

만약 처음에 lock의 값이 0이었다면(아무도 임계구역에 들어가 있지 않음 내가 lock을 걸고 들어가면 되는 것임), while문을 탈출하면 lock 값이 1이 된다.(0을 1로 바꾸고 임계구역에 들어갔다 나왔으니까) 반대로 lock의 값이 1이면(누군가 이미 lock을 걸어두었다면) while

문에서 갇히고 lock 값은 그대로 1이 된다.(계속 기다림 남이 임계구역 나와서 0으로 만들어 줄 때까지)

세마포어 (Semaphore)

사용이유

앞의 방식들을 추상화시킴 lock을 걸고 lock을 풀고 이런 것을 개발자에게 좀 더 쉽게 제공하기 위해 사용

공유자원 쓰고 반납하는 것을 세마포어가 처리해 줌

p 연산는 공유데이터를 획득하는 과정

v 연산는 다 사용하고 반납하는 과정

Integer variable

S를 사용함. (s가 5면 자원이 5개 있는 것임 p 연산 5번 해서 자원을 가져갈 수 있는 것임 v 연산을 5번하 면 자원을 다시 5개 반납)

아래의 두 가지 atomic 연산에 의해서만 접근이 가능

P(S): 공유 데이터를 획득하는 과정 (lock)으로 S가 양수이어야 한다.

```
// P(S)
while (S <= 0)do no-op; // ex) wait. 자원이 있는지 확인하고 있으면 가져와서 하나 뺌
S--;
```

V(S): 공유 데이터를 사용하고 반납하는 과정 (unlock)

```
// V(S)
S++;
```

---

n개의 프로세스가 임계 구역에 들어가려는 경우

Synchronization variable

```
semaphore mutex;
```

위 세마포어 변수 값만큼 여러 개의 프로세스가 임계 구역에 접근할 수 있다. 이번 예제에서는 mutex의 값이 1이라고 가정한다. (1개의 프로세스만 임계 구역 접근 가능)

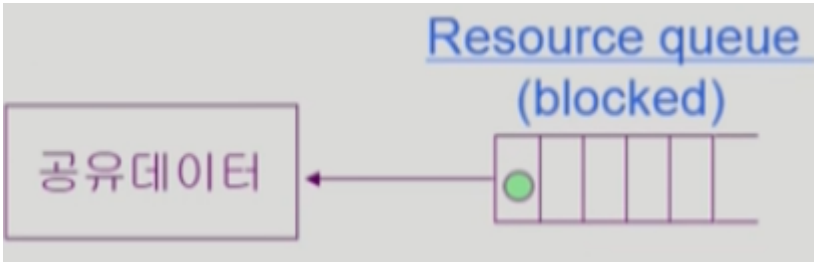
Process Pi

```
do {
    P(mutex);// mutex의 값이 양수면 임계 구역 접근하고, 아니면 기다린다.
    criticalsection
    V(mutex);// mutex의 값을 1 증가한다. 다 쓰고 나서
    remaindersection
}while (1);
```

P(mutex) 연산 수행 시, mutex의 값이 양수가 아니면 계속 기다려야 하는데, 프로세스의 CPU 할당 시간이 끝날 때까지 무의미하게 CPU를 낭비하는데, 이러한 현상을 busy-wait 또는 spin lock 이라고 부른다. 이러한 단점을 보완하기 위해 Block & Wake-up 혹은 Sleep lock 이라고 부르는 기법이 생겨났다.

프로그래머는 세마포어가 지원된다면 p, v 연산만 해주면 되는 것이고 어떻게 구현할지는 그때그때 프로그래머가 고민해야 한다. (추상 자료형이기 때문에)

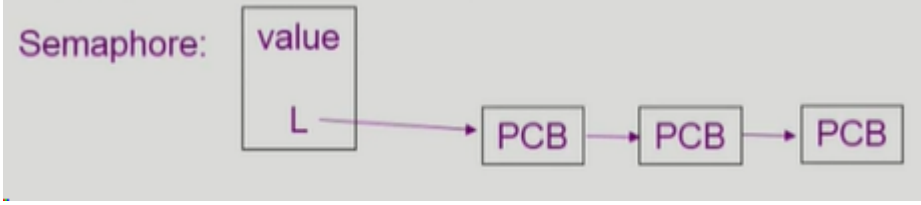
Block / Wake-up 기법



다른 프로세스가 공유데이터 사용 시 해당 데이터를 줄 때까지 내 차례가 오지 않으니까 계속 반복문을 통해 기다리지 말고 기다리는 프로세스 자체를 Block 시켜서 CPU를 아예 반납하고 잠들게 한 다음 공유 데이터를 가지고 있던 프로세스가 내놓으면 그때 Wake-up 깨어나서 레디 큐에 들어와 CPU를 얻게 하는 방법으로 세마포어를 구현하는 방법임

세마포어는 다음과 같이 정의

```
typedefstruct
{
    int value;// 세마포어 변수
    structprocess *L;// 프로세스 Wait Queue : 잠들어있는 프로세스 연결하기 위한 큐
} semaphore;
```



block과 wake-up을 다음과 같이 가정

- block
  - 커널은 block을 호출한 프로세스를 suspend하고, 이 프로세스의 PCB를 세마포어에 대한 Wait Queue에 넣음. (위 그림처럼 세마포어 획득하지 못한 프로세스의 PCB를 큐에 연결해 매달아 둠)
- wakeup(P)
  - block된 프로세스 P를 wake up하고, 이 프로세스의 PCB를 Ready Queue로 옮김.

세마포어 연산은 다음과 같이 정의

P(S): 세마포어 변수 S를 무조건 1 줄이고, 그 변수의 값이 음수(자원을 누군가 다 가져가고 없다는 이야기)면 해당 프로세스를 Wait Queue로 보낸 후 Block 상태로 만든다.

```
// P(S)
S.value--;
if (S.value < 0)
{
    add this process to S.L;// 큐에다 추가하고
    block(); // Block 상태에 있다가 자원이 생기면 일어날수 있음
}
```

V(S): 세마포어 변수 S를 무조건 1 늘리고(자원을 다 쓰고 나면), 그 변수의 값이 0보다 작거나 같으면 이미 기다리고 있는 프로세스가 있으므로(값을 빼고 잠들기 때문에 양수라면 그냥 자원을 쓰는 것 음수면 잠드는 것이고) 프로세스 P를 Wait Queue에서 꺼내서 Ready

Queue로 보낸다. 세마포어 변수 S값이 양수면 아무나 임계 구역에 들어갈 수 있으므로 별다른 추가 연산을 하지 않는다. V 연산은 특정 프로세스가 공유 데이터를 반납한 후 임계 구역에서 나가는 연산임을 기억해야 한다.

나 이제 임계 구역에서 나갈 거니까 기다리는 친구 있으면 임계 구역 들어가~

```
// V(S)
S.value++;
if (S.value <= 0)
{
    remove a process Pfrom S.L;
    wakeup(P); // 자원을 반납하고 끝나는게 아니라 해당 자원을 기다리는 친구 있다면 깨워주고 끝내야함
}
```

Busy wait vs Block Wake-up

- 일반적으로 Block Wake-up 기법이 좋음.
- 임계 구역의 길이가 긴 경우 Block/Wake-up 기법이 적합함.
- 임계 구역의 길이가 매우 짧은 경우 Block/Wake-up 기법의 오버헤드가 Busy-wait 기법의 오버헤드보다 크므로 Busy Wait 기법이 적합할 수 있음.

세마포어의 종류

계수 세마포어 (Counting Semaphore)

- 도메인이 0 이상인 임의의 정수 값
- 여러 개의 공유 자원을 상호 배제함.
- 주로 resource counting에 사용됨.

이진 세마포어 (Binary Semaphore)

- 0 또는 1 값만 가질 수 있음.
- 한 개의 공유 자원을 상호 배제함.
- mutex와 유사함. (완전히 같지는 않음.)

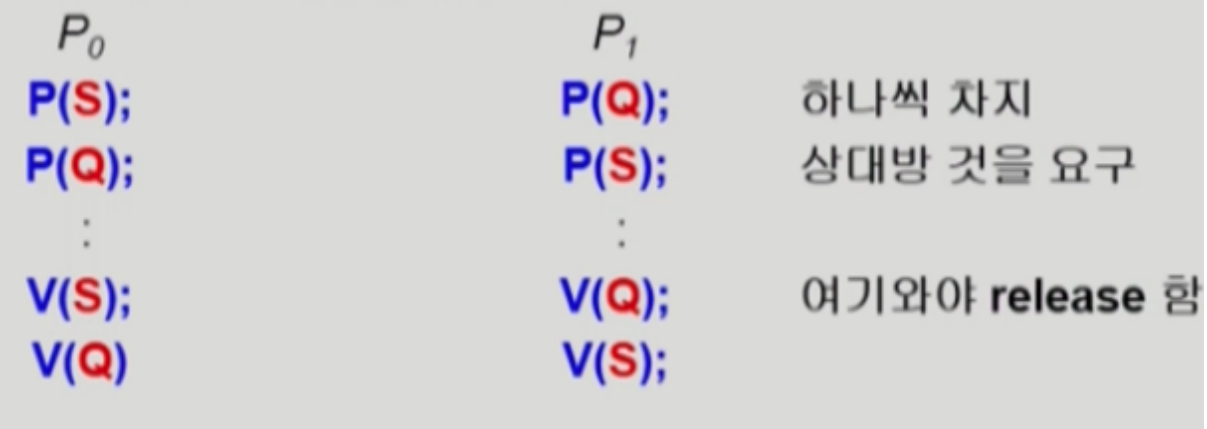
Deadlock과 Starvation

Deadlock

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

Deadlock 예시

S와 Q가 1로 초기화된 세마포어라 하자.



P0가 CPU를 얻어서 P(S) 연산까지 수행하여 S 자원을 얻었다고 가정해 보자. 그런데 여기서 P0의 CPU 할당 시간이 끝나 Context Switch가 발생하여 P1에게 CPU 제어권이 넘어갔다. P1은 P(Q) 연산을 수행하여 Q 자원을 얻었으나 또 다시 CPU 할당 시간이 끝나 Context Switch가 발생하여 P0에게 CPU 제어권이 넘어갔다.

P0은 P(Q) 연산을 통해 Q 자원을 얻고 싶지만, 이미 Q 자원은 P1이 갖고 있는 상태이므로 Q 자원을 가져올 수가 없다. 마찬가지로 P1도 P(S) 연산을 통해 S 자원을 얻고 싶지만, 이미 S 자원은 P0이 갖고 있는 상태이므로 S 자원을 가져올 수 없다.

이렇게 P0와 P1은 영원히 서로의 자원을 가져올 수 없고, 이러한 상황을 Deadlock이라고 한다.

Deadlock 해결 방안

해결 방안은 추후 자세히 배우겠지만, 자원의 획득 순서를 정해주어 해결하는 방법이 있다. S를 획득해야만 Q를 획득할 수 있게끔 순서를 정해주면 프로세스 A가 S를 획득했을 때 프로세스 B가 Q를 획득할 일이 없다.

Starvation (기아)

- indefinite blocking
- 프로세스가 자원을 얻지 못하고 무한히 기다리는 현상이다.(특정 프로세스만 자원을 얻으며)

Deadlock vs Starvation

언뜻 보기에 둘 다 자원을 가져오지 못하고 무한히 기다리니까 같은 단어라고 혼동할 수 있다.

Deadlock은 P0 프로세스가 A 자원을 보유하고 있고, P1 프로세스가 B 자원을 보유하고 있을 때 서로의 자원을 요구하여 무한히 기다리는 현상이다. 반면 Starvation은 프로세스가 자원 자체를 얻지 못하고 무한히 기다리는 현상이다. Starvation은 기아라는 단어에 맞게 어떤 프로세스가 자원이 없어서 굶어 죽는다(?)고 생각하면 이해하기 편하다.