임계 영역에 관한 보고서

과목명 : 운영체제

과제 제목 : 임계 영역에 관한 보고서

제출일 : 2022.12.01 (월)

제출자 : (한라대학교 / ICT융합공학부 / 201932030 / 송현교)

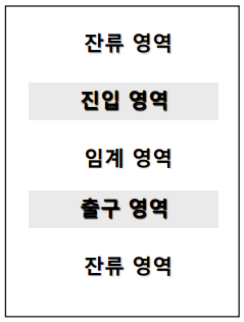
목차

임계 영역이란 ?

임계 영역(Critical Section) 이란, 다중 프로그래밍 환경에서 프로세스들이 공유 자원에 접근하는데 있어서, 한 순간에 무조건 하나의 프로세스만 접근할 수 있도록 보장해주는 영역을 의미한다.

공유 자원을 독점하게 해주는 영역이라고도 표현할 수 있다.

임계 영역 내에서의 프로세스는 결정성[1] 보장을 위해 항상 상호 배제적으로 수행되어야 한다.



병행성 프로세스의 코드영역

잔류 영역 : 임계 영역과 무관한 영역

진입 영역 : 프로세스가 임계 영역에 진입을 요청하는 단계

임계 영역 : 2개 이상의 프로세스가 동시에 액세스하면 안되는 공유자원을 액세스하는 코드 영역

출구 영역 : 프로세스가 임계 영역에서 빠져나왔음을 의미하는 영역

임계 영역을 갖는 프로세스의 일반적 구조 : 진입 영역 -> 임계 영역 -> 출구 영역 - > 잔류 영역

임계 영역에 대한 해결책

프로세스가 '임계영역 실행을 시작' 하면, '임계영역 실행을 종료' 할 때 까지 방해받지 않고 수행하도록(중단되지 않도록) 하는 보장이 모든 프로세스에 적용되어야 한다.

이러한 보장에 대한 상세조건을 다음 페이지에서 다루고자 한다.

[1] 어떤 환경이든 프로세스는 동일한 입력에 대한 처리 결과가 같아야 하는 성질

임계 영역 문제 해결 3조건

1. 상호 배제 (Mutual Exclusion)

한 프로세스가 임계 구역에 들어가 있으면, 다른 프로세스는 앞서 들어가 있던 프로세스가 작업을 마치기 전까지 임계 구역에 들어가서는 안 된다.

즉, 2개 이상의 프로세스가 동시에 임계 영역에 진입하지 못하게 하는 것.

2. 진행 (Progress)

임계 영역에 들어가 있는 프로세스가 없는 상황에서, 새로 들어가고자 하는 프로세스 후보가 여러 개 일 때, 후보들 중 하나를 선정하는 과정은 유한한 시간 내에서 이루어져야 한다.

즉, 진입을 원하는 프로세스는 진입할 수 있어야 한다.

3. 제한된 대기 (Bounded Waiting)

프로세스가 임계 영역에 접근을 요청했음에도 무한 대기 (Infinite Postpone) 하게 되는 상황이 발생하지 않아야 한다.

임계 영역 문제 해결 종류

- **소프트웨어적인 방법**

  두 (N = 2) 프로세스 간 임계영역 해결 방법

  - Dekker 1 알고리즘 : 상호 배제 만족, 진행 불만족, 제한된 대기 불만족 (불완전)
  - Dekker 2 알고리즘 : 상호 배제 만족, 진행 불만족, 제한된 대기 불만족 (불완전)
  - Peterson 알고리즘 : 상호 배제 만족, 진행 만족, 제한된 대기 만족 (완전)

  다중 (N >= 2) 프로세스 간 임계영역 해결 방법

  - Bakery Algorithm (빵집 알고리즘) : 상호배제 만족, 진행 만족, 제한된 대기 만족 (완전)

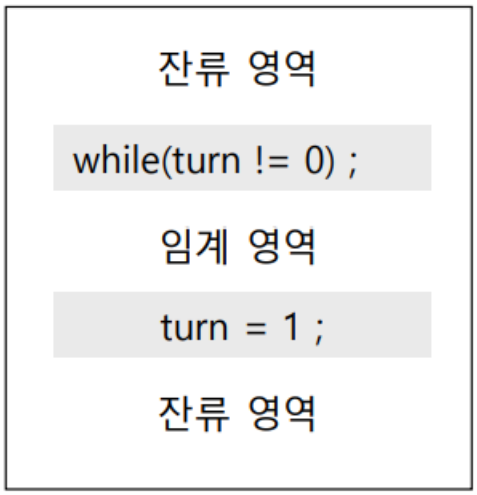
- **하드웨어적인 방법**

  - Test And Set 명령어 알고리즘 1 : 상호 배제 만족, 진행 만족, 제한된 대기 불만족 (불완전)
  - Test And Set 명령어 알고리즘 2 : 상호 배제 만족, 진행 만족, 제한된 대기 만족 (완전)
  - Swap 명령어 알고리즘 : 상호 배제 만족, 진행 만족, 제한된 대기 불만족 (불완전)

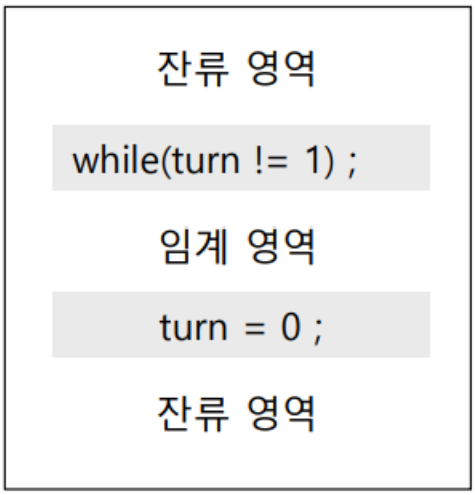
- **세마포어에 의한 방법 (Semaphore)**

  - 세마포어에 의한 상호 배제 알고리즘
  - 세마포어에 의한 동기화 문제 해결
  - 세마포어에 의한 Busy Waiting / Spin Lock 문제 해결

Dekker 1 알고리즘 : 소프트웨어적 방법, 두 프로세스 전용, 불완전한 알고리즘



(a) 프로세스 (P0)



(b) 프로세스 (P1)

*** turn은 공유 변수**

1) 상호 배제 조건을 만족시키는가 ?

프로세스 P0은, 진입 영역에서 turn = 0 일 때 임계 영역에 진입할 수 있다. 또한, 임계 영역에서의 수행을 마치고 출구 영역으로 나오며 turn의 값을 1로 수정한다.

프로세스 P1 또한, 진입 영역에서 turn = 1일 때 임계 영역에 진입 가능하며, 출구 영역으로 나올 때 turn의 값을 0으로 수정한다.

즉, turn의 값에 의해 교대로 임계 영역을 수행(동시 수행 X) 하므로 상호 배제 조건을 만족시킨다.

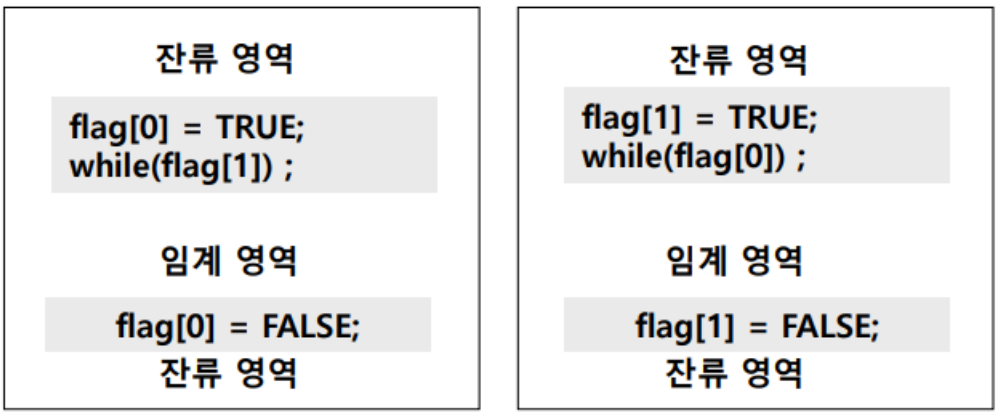
2) 진행 조건을 만족시키는가 ?

임의의 한 프로세스가 수행을 마친 후, 다른 프로세스가 임계 영역에 접근하려 하지 않아 재진입하려 할 때, turn의 값을 스스로 변화시킬 수 없기 때문에 진입이 불가능하다. 즉, 진행 조건을 만족시키지 못한다.

3) 제한된 대기 조건을 만족시키는가 ?

앞서 설명하였듯이, turn의 값을 다른 프로세스가 변화시켜 줄 때 까지 임계 영역에 진입 할 수 없기 때문에, 무한정 대기(Infinite Postpone) 현상이 발생한다. 즉, 제한된 대기 조건을 만족시키지 못한다.

Dekker 2 알고리즘 : 소프트웨어적 방법, 두 프로세스 전용, 불완전한 알고리즘



(a) 프로세스(P0) (b) 프로세스(P1)

*** flag[0], flag[1]은 공유 변수이며, 초기값은 모두 False로 설정.**

1) 상호 배제 조건을 만족시키는가 ?

각 프로세스는 임계 영역 수행을 마치고 출구 영역으로 나와야만 flag 변수의 변화를 통해 상대 프로세스가 임계 영역에 진입할 수 있게끔 허락하는 구조를 형성하고 있다.

즉, 각 프로세스가 각각 임계 영역을 수행하고 출구 영역으로 나오기 전까지 flag 변수의 변화가 없으므로 (다른 프로세스의 임계 영역 진입을 허용하지 않으므로) 상호 배제 조건을 만족시킨다.

2) 진행 조건을 만족시키는가 ?

임의의 한 프로세스가, 상대가 수행하지 않는 동안 재진입하려 할 경우 특별한 일이 없을 시 가능하나, P0 프로세스가 flag[0] 변수를 True로 설정한 후 문맥 교환[2]이 발생하는 경우, P1 프로세스가 flag[1] 변수를 True로 설정하더라도 임계 영역에 진입하지 못하고, P0 프로세스도 진입하지 못하게 되는 현상이 발생한다. 즉, 진행 조건을 만족시키지 못한다.

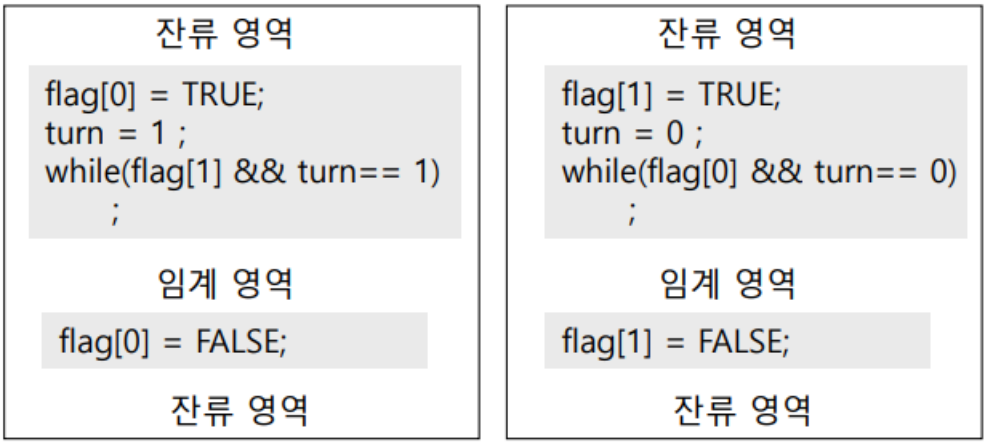
3) 제한된 대기 조건을 만족시키는가 ?

진행 조건의 불만족과 같이, 문맥 교환에 의한 예외로 flag[0]과 flag[1]이 모두 True일 시, P0과 P1이 둘 다 진입할 수 없기 때문에, 무한 대기하는 현상이 발생한다. 즉, 제한된 대기 조건을 만족시킬 수 없다.

---

[2] 하나의 프로세스가 CPU를 사용 중인 상태에서, 다른 프로세스가 CPU를 사용하도록 하기 위해 이전의 프로세스의 상태(문맥)를 보관하고 새로운 프로세스의 상태를 적재하는 것을 의미한다.

Peterson 알고리즘 : 소프트웨어적 방법, 두 프로세스 전용, 완전한 알고리즘



(a) 프로세스(P0) (b) 프로세스(P1)

*** turn, flag[0], flag[1]은 공유 변수이며, flag 변수의 초기값은 False 이다.**

1) 상호 배제 조건을 만족시키는가 ?

앞서 살펴본 Dekker 1, 2 알고리즘과 같이 본 알고리즘 또한 출구 영역의 flag 변수 변경(True, False) 으로 하여금 임계 영역 내에 프로세스 존재 여부를 탐색 가능케 하여 상호 배제 조건을 만족시킨다.

2) 진행 조건을 만족시키는가 ?

만약 P0 프로세스가 수행을 마친 후 P1이 수행하려 하지 않아 재진입을 원한다고 가정할 때, 여전히 flag[1] 변수가 False인 상태이므로 재진입이 가능하다. 반대의 경우도 동일하므로 진행 조건을 만족시킨다.

3) 제한된 대기 조건을 만족시키는가 ?

Dekker 2 알고리즘에서 문제가 되었던 문맥 교환의 경우를 살펴보자. 예를 들어, P0 프로세스의 임계 영역 실행 도중 문맥 교환이 발생하였다고 가정하면, P1 프로세스는 flag[1]을 True로, turn을 0으로 변경한 후 대기하게 된다. 이후 P0 프로세스가 출구 영역으로 나옴으로써 flag[0]이 False로 설정되고, 결과적으로 문맥 교환이 발생하더라도 P0 프로세스가 임계 영역 수행을 마친 후 P1 프로세스의 진입이 가능함을 알 수 있다. 재진입의 경우는 이미 진행 조건에서 가능함을 확인하였으므로 각 프로세스가 무한 대기에 빠질 가능성을 배제하였다. 즉, 제한된 대기 조건을 만족시켰다.

Bakery Algorithm : 소프트웨어적 방법, N >= 2 프로세스 전용, 완전한 알고리즘

잔류 영역

```
choosing[i] = TRUE;  //Pi가 임계영역 진입 시도
number[i]= max(number[0],number[1],...,number[n-1])+1;  //일련번호 부여. 최고 보다 1 높은 값 배정
choosing[i] = FALSE;
for(j=0; j<n;  j++){  //모든 프로세스에 대해 조사하여
  while(choosing[j]) ;  //일련번호 부여 시 정지
  while( (number[j] != 0) && ( number[j] < number[i] ) ) ; //낮은 번호 먼저
}
```

임계 영역

```
number[i] = 0 ;
```

잔류 영역

*** choosing[n]과 number[n] 은 공유 변수이며, 초기값은 False와 0으로 선언한다.**

*** 자신의 choosing을 True로 설정하고, 번호를 부여 받은 뒤 False로 변경. 번호는 기존 부여된 번호보다 1 큰 번호 배정.**

*** for 문에서는 가장 작은 프로세스를 임계 영역에 진입시킨다. 부여 받은 번호가 같은 경우 작은 프로세스 우선 진입.**

1) 상호 배제 조건을 만족시키는가 ?

Choosing을 통해 True로 설정된 프로세스는 number에 의해 번호표를 부여받고, 다시 False로 되돌아온다. 그 후 for문에 의해 작은 번호표를 부여받은 프로세스부터 임계 영역에 진입하고, 수행을 마치면 번호표를 반납하고 잔류 영역으로 퇴장한다. 이 때 임계 영역 내에 프로세스가 존재할 경우 while((number[j] != 0 && (number[j] < number[i])) 문장에서 타 프로세스의 임계 영역 출입 시도를 통제함으로써 상호 배제를 만족시킨다는 것을 알 수 있다.

2) 진행 조건을 만족시키는가 ?

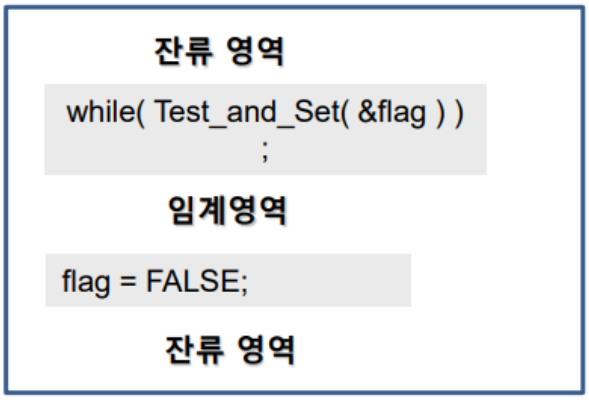
임계 영역 진입을 원하는 프로세스가 하나도 없는 상황에서, 실행을 마치고 빠져 나온 임의의 프로세스가 재진입을 원하는 상태라고 가정하자. 이 경우 작은 프로세스가 우선 진입한다는 전제 조건에서 임의의 프로세스가 가장 작은 프로세스에 해당하므로 당연하게도 임계 영역에 진입하게 된다. 따라서 진행 조건을 만족시킨다.

3) 제한된 대기 조건을 만족시키는가 ?

임계 영역에 이미 프로세스가 들어간 상태일 때, 다음 순번으로 임계 영역에 진입할 프로세스는 번호표를 부여받은 상태로 대기한다. 먼저 들어간 프로세스가 수행을 마치고 빠져나오며 번호표를 반납하면, 대기하던 프로세스는 가장 작은 수의 번호표를 가진 프로세스가 되며, 대기 중이던 while((number[j] != 0 && (number[j] < number[i]))문에서 빠져나오며 자연스레 임계 영역으로 진입할 수 있게 된다. 이와 더불어 재진입 문제가 해결된 것을 진행 조건에서 확인하였으니 프로세스들이 무한 대기하는 현상이 존재하지 않는다. 이를 통해 제한된 대기 조건을 만족시켰다는 것을 확인 가능하다.

Test And Set 명령어 알고리즘 1 : 하드웨어적 방법, 불완전한 알고리즘

```
bool Test_and_Set(bool * lock) {
      bool ret;
      ret = *lock;
      *lock = TRUE;
      return  ret;
}
```



*** Test And Set 명령어 알고리즘은 하드웨어적 방법이므로 수행 중 문맥 교환, 인터럽트 발생하지 않음.**

*** flag는 공유 변수로써, 초기값은 False로 선언되었다.**

1) 상호 배제 조건을 만족시키는가 ?

임의의 프로세스가 임계 영역 진입을 위해서 Test_and_Set 함수에 flag를 대입하여 검사받는다. Flag의 초깃값은 False이므로 ret에는 False가 들어가고, lock이 가리키는 주소에 있는 값(flag) 을 True로 설정, 이후 False 상태인 ret를 반환한다. 즉, while(Test_and_Set(&flag)) 는 False이므로 임계영역에 진입할 수 있다. 이 상태에서 다른 프로세스가 진입을 원하는 상황을 가정해보자. Flag는 True인 상태이다. 이 때 Test_and_Set 함수에서 True가 반환되므로 임계 영역에 이미 프로세스가 들어가 있는 상태에서 다른 프로세스의 진입이 통제되었고, 프로세스가 출구 영역으로 나오며 flag 변수를 False로 설정해줌으로써 다른 프로세스의 진입이 허용되는 것을 알 수 있다. 즉 상호 배제 조건이 만족된다.

2) 진행 조건을 만족시키는가 ?

상호 배제 조건을 살펴보며, 임의의 프로세스가 수행을 마치고 출구 영역으로 나올 때, flag 변수를 False로 변경하므로 임계 영역에 프로세스가 없으면 다른 프로세스가 진입할 수 있음을 알 수 있었다. 재진입 또한 가능하다. 즉, 진행 조건을 만족시켰다.

3) 제한된 대기 조건을 만족시키는가 ?

진행 조건을 살펴보며 재진입이 가능하다는 것을 확인하였다. 그러나 본 알고리즘에서는 재진입 한 프로세스가 실행을 마치고 다른 프로세스가 진입 하기 전에 또 다시 진입하려는 현상이 발생 가능하다. 이러한 경우 다른 프로세스가 임계 영역에 진입하지 못하고 무한정 대기하는 여지가 생겨 버리기 때문에 제한된 대기 조건을 만족시킬 수 없다.

Test And Set 명령어 알고리즘 2 : 하드웨어적 방법, 완전한 알고리즘

```
bool Test_and_Set(bool * lock) {
      bool ret;
      ret = *lock;
      *lock = TRUE;
      return  ret;
}
```

잔류 영역

```
waiting[i]=TRUE;
key=TRUE;  //flag 상태를 읽어 오기 위한 변수
while ( waiting[i] && key ) key=Test_and_Set( &flag );
waiting[i] = FALSE;
```

임계 영역

```
j = (i+1) % N;
while( (j != i) && ( waiting[j] == FALSE ) )
      j = (j+1)%N;  //진입 의사 표시 프로세스 찾기…
if (j == i) flag = FALSE;//대기자가 없어 자기 진입 허용
else waiting[j] = FALSE;//대기자가 진입할 수 있게  함
```

잔류 영역

*** Test And Set 명령어 알고리즘은 하드웨어적 방법이므로 수행 중 문맥 교환, 인터럽트 발생하지 않음.**

*** flag 변수 접근 및 변환은 Test_and_Set에게만 허용**

*** waiting[N]과 flag는 공유 변수로써, 초기값은 False로 설정한다.**

1) 상호 배제 조건을 만족시키는가 ?

Test_and_Set 명령어는 Test And Set 명령어 알고리즘 1의 명령어와 완전히 동일하다.

임의의 프로세스 P0이 임계영역에 처음 진입하고, 이후 P1이 진입을 시도하는 상황을 가정해보자. 우선, 진입 영역에서 P0 프로세스의 waiting[0]과 key가 True로 설정된다. 즉, while문 내의 조건들을 모두 만족하므로 key를 False, waiting[0]을 False, flag를 True로 설정하고 임계 영역에 문제 없이 진입한다.

이렇게 P0이 임계 영역에 진입한 상태에서 P1이 진입하고자 하면, waiting[1]과 key가 True이기 때문에 while문에 들어가게 되는데, flag값 또한 True이기 때문에 P1은 임계 영역에 진입하지 못하고 while문에 갇혀 있게 된다. 이를 통해 상호 배제 조건을 만족시켰다는 것을 알 수 있었다.

2) 진행 조건을 만족시키는가 ?

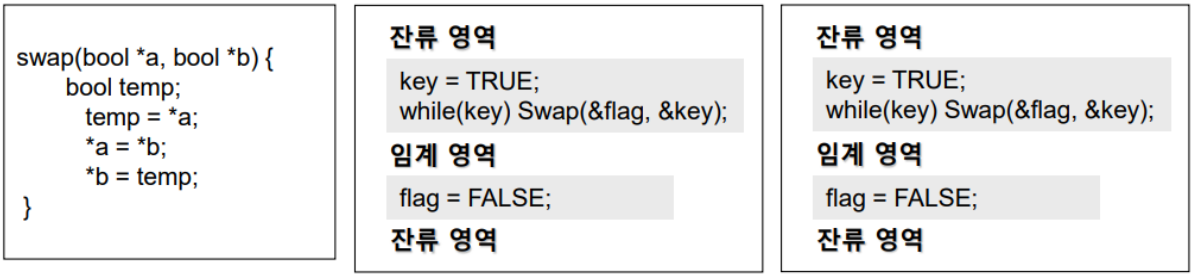
상호 배제 조건을 알아보면서, P1이 while문 안에 갇혀 있음을 알 수 있었다. 그러면, P0이 임계 영역 수행을 마치고 나왔을 때 P1이 임계 영역에 진입할 수 있는지 알아보겠다.

우선, P1은 P0이 임계 영역에 있을 때 waiting[1], key, flag를 각각 True로 설정하고 대기 상태에 들어간다. 이후, P0이 임계 영역 수행을 마친 뒤 출구 영역으로 나오며, j의 값을 1로 설정한다. j와 i의 값이 다르기 때문에 whlie문과 if문을 쭉 지나쳐 else waiting[j] = False문으로 들어간다. 따라서 waiting[1]이 False가 되었으므로 진입 영역의 while문을 빠져나가게 되고, 임계 영역에 진입할 수 있게끔 되었다. 이를 통해 진행 조건을 만족시킴을 확인하였다.

3) 제한된 대기 조건을 만족시키는가 ?

진행 조건에서 확인하지 못한 재진입의 경우를 확인해야 하는데, 본 알고리즘의 경우, if ~ else 조건문을 통하여 진입하고자 하는 프로세스가 더 있는지 검사해, 없으면 재진입을 가능하게, 있으면 대기 중이던 프로세스가 진입할 수 있게끔 경우를 각각 설정 해 줌으로써 재진입 시 무한 반복 현상이 일어나지 않게 설계하였다. 여기서 제한된 대기 또한 만족시켰음을 확인할 수 있었다.

Swap 명령어 알고리즘 : 하드웨어적 방법, 보완에 의한 완전한 알고리즘



(Process01) (Process02)

*** flag는 공유 변수로써, 초기값은 False이다.**

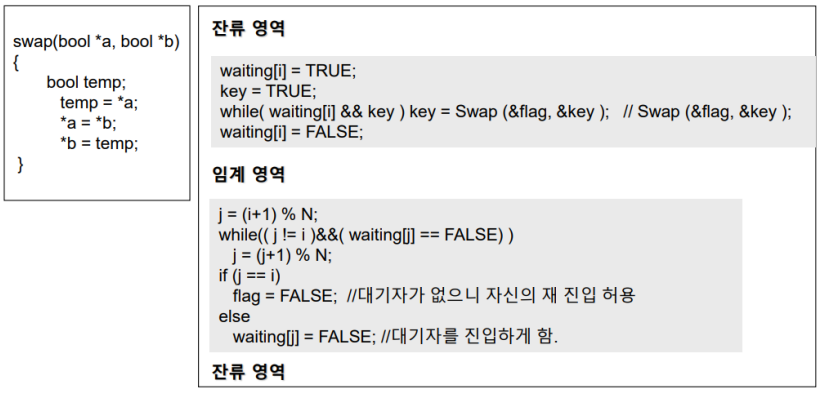
1) 상호 배제 조건을 만족시키는가 ?

임계 영역 진입을 원하는 프로세스는 Swap 명령어를 호출하여 진입 가능성을 검사 받는다. 임계 영역에 진입하기 위해서는 key가 False가 되어 while문을 빠져나가야 한다. 임의의 프로세스는 flag를 True로 설정한 후 임계 영역에 진입한다. flag가 True이면, Swap 명령어에 의해 key도 True로 설정되므로 타 프로세스의 접근은 while문에서 통제당하게 된다. 따라서 상호 배제 조건을 만족시킨다.

2) 진행 조건을 만족시키는가 ?

임계 영역 수행을 마친 프로세스는 출구 영역으로 나오며 flag 변수를 False로 설정한다. 그러면 진입을 기다리고 있던 다른 임의의 프로세스는 Swap 명령어에 의해 key 값을 False로 설정하며 진입이 가능해진다. 고로 진행 조건 또한 만족시키게 된다.

3) 제한된 대기 조건을 만족시키는가 ?



*** waiting[N]과 flag는 공유 변수로써, 초기값은 False이다.**

특정 프로세스의 반복된 진입을 막지 못하여 제한된 대기 조건은 만족시키지 못하였으나,

Test and Set 2 알고리즘과 같은 방법으로 보완하여, 해결 할 수 있게 되었다.

세마포어(Semaphore)에 의한 방법 : 특별한 P, V 연산자에 의존한 방법

용어 정리

- **세마포어 변수** : 초기값이 부여된 변수로 P, V 연산자에 의해서만 변경할 수 있는 정수형 변수
- **P 연산자** : Proberen(검사하다) => wait(기다리다)

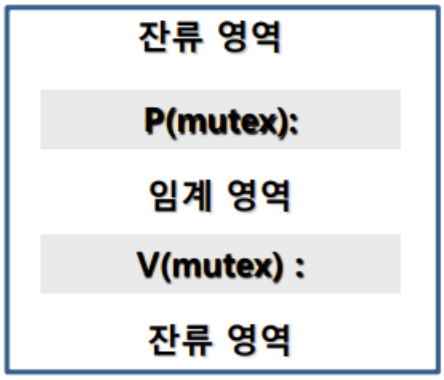
  P(S) : while(S <= 0);

  S :=[3] S – 1;
- **V 연산자** : Verhogen(증가하다) => Signal(신호를 전하다)

  V(S) : S:= S + 1;

[3] 변수에 값을 대입할 때 사용하는 이항 연산자로써, 왼쪽 피연산자에 오른쪽 피연산자를 대입함.

세마포어에 의한 상호 배제 알고리즘 – 1



P(S): while(S <= 0) ;
S = S – 1;

V(S): S = S + 1;

*** 임계 영역에 1개의 진입만 허용.**

*** 세마포어 변수 mutex 사용, 임계 영역 진입을 1개만 허용하므로 mutex 초기 값을 1로 설정.**

1) 상호 배제 조건을 만족시키는가 ?

임의의 프로세스가 임계 영역에 진입하기 위해서는 P 연산을 거쳐야 한다. P 연산을 통해 mutex <= 0 일 시 대기, 0보다 클 시 mutex를 1 감소시키고 임계 영역에 진입한다.

임계 영역 수행을 마친 프로세스는 출구 영역으로 나오며 V 연산을 커쳐 mutex를 1 증가시키는데, 이를 통해 임계 영역에 하나의 프로세스만 들어가고 나올 수 있음을 확인할 수 있다. 이를 통해 상호 배제 조건을 만족시켰음을 확인 할 수 있었다.

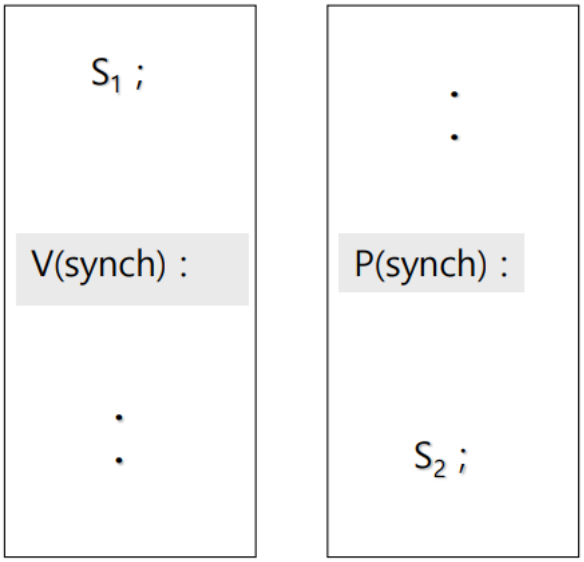
2) 진행 조건을 만족시키는가 ?

상호 배제 조건에서 프로세스는 하나씩만 들어가고 나올 수 있음을 확인하였다. 즉, 진행 조건 또한 만족되었다.

3) 제한된 대기 조건을 만족시키는가 ?

그러나, 임의의 프로세스가 계속 재진입하여 타 프로세스의 무한 대기 현상이 일어났을 시 막을 방도가 없기 때문에 제한된 대기 조건을 만족시키지 못하였다.

세마포어에 의한 동기화 문제 해결 : S1이 S2보다 먼저 수행해야 하는 경우



P(S): while(S <= 0) ;
S = S – 1;

V(S): S = S + 1;

*** 동기(순서 제어) 문제를 해결하는 것이므로 synch 변수를 이용. 초기값은 0으로 설정.**

1) 동기화 문제가 해결 되었는가 ?

역으로, S2가 먼저 수행되는 경우를 가정해보자. synch 변수의 초기값이 0임에 따라 P(synch) 내의 while 문에서 결국 대기할 수 밖에 없는 상황이 형성된다.

즉, S1을 지나 V(synch)에서 synch가 1 증가 되어야만 S2의 실행이 가능하므로, 동기화 문제를 해결했음을 의미한다.

세마포어에 의한 임계 영역 운용 : Busy Waiting[4] / Spin Lock[5] 문제 해결

```
struct {
      integer   value; //초기값을 1로 설정
      struct pcb *ptr; //대기 상태의 큐 포인팅
     } S;  //세마포어 변수

P(S): S.value = S.value-1;
       if(S.value < 0) {
              이 프로세스를 S.ptr에 등록;
              block();
    }

 V(S): S.value = S.value+1;
       if(S.value <= 0) {
              S.ptr로부터 프로세스(P)를 제거;
              wakeup(P);
 }
```

잔류 영역

P(S):

임계 영역

V(S) :

잔류 영역

*** S.value의 값이 마이너스 일 때, Sleep 된 프로세스 수를 의미한다.**

1) Busy Waiting / Spin Lock 이(가) 해결 되었는가 ?

앞서 살펴본 Dekker, Test And Set , ... 등의 알고리즘은 임계 영역 접근 가능 여부를 무한 루프문을 사용하여 체크하였다. 반면 세마포어의 경우는 조금 다르다. 상단의 알고리즘을 살펴보면,

P 연산 실행 시 S.value를 1 감소시키고, 그 값이 0보다 작으면 프로세스를 S.ptr에 등록한 후 중지 상태로 전환시킨다.

V 연산은, S.value를 1 증가시키며, 그 값이 0보다 작거나 같을 시 S.ptr로부터 프로세스를 제거한 후 프로세스를 다시 활성화 시킨다.

여기서 중요한 것은 block() 와 wakeup() 이다. 세마포어는 앞 페이지의 알고리즘들 과는 달리 대기하고 있는 스레드를 '중지' 상태로 만들고, 수행 할 때 '깨우는' 방식을 통해 Busy Waiting / Spin Lock의 발생을 차단하였다.

Block-Wakeup 방식은 연산 효율의 상승을 기대할 수 있다는 장점이 있지만, Critical Section의 길이가 짧아 프로세스가 빈번하게 바뀌는 경우 Busy-Wait의 오버헤드보다 더 큰 오버헤드를 발생 시킬 수도 있다는 단점도 존재한다. 따라서 오버헤드를 예측하고 상황에 맞는 알고리즘을 사용하는 것이 중요하다고 생각된다.

---

[4] 프로세스 동기화 상황에서 프로세스가 자원에 대한 접근 권한을 얻기 위해, 될 때까지 접근 조건을 반복적으로 확인하는 일을 뜻한다.

[5] lock을 얻고자 하는 쓰레드가 CPU점유를 다른 쓰레드에 내주지 않고, 계속해서 lock을 얻을 수 있는지 무의미한 루프를 돌면서 (busy wait) 체크하는 방법이다.

과제 작성 관하여 검토 및 추가 코멘트

'무엇인가를 안다는 것은, 말이나 글로써 설명 할 수 있는 것'

본 과제를 진행하면서 저에게 가장 크게 와닿은 말이었습니다.

강의 중, 저는 제 자신이 이해했으리라 굳게 믿었고

반증으로, 막상 내용을 작성하려니 막히는 부분이 생각보다 많았습니다

카페에서 이전 수업 자료, 혹은 외부 웹사이트에서 관련자료를 찾아가며 다시 공부하였습니다.

배우고자 하는 의지보다는, 점수를 받는다 라는 의무감에 더 치우친 마음으로 시작한 과제였지만,

여기저기 흩어져 있는 지식을 흡수하여 내 것으로 만들어 가는 듯한 중간 과정들이 좋았습니다.

성적이라는 부담은 학생이라면 피치 못할 무게일 것이고, 비단 저만이 느끼는 것이 아니지만

저를 비롯한 학생들이 부담 속에서도 이렇게 소소한 재미를 찾는 긍정적 능력을 찾는다면

거짓 출석 후, 시험기간에 벼락치기 등의 편법을 통한 점수 향상 위주의 공부법보다는

자기 향상심을 극대화 시킬 수 있는 긍정적 방식의 공부가 앞으로 진행 될 수 있을 것이라 느낍니다.

이번 학기도 고생 많으셨습니다.

감사합니다.