운영체제 2020-2학기 기말고사 정리

고려대 컴퓨터정보통신대학원 빅데이터융합학과 조송현

Chapter 6. Synchronization

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Entry section Each process must request permission to enter its critical section.

싱크로나이제이션 환경 : 자원을 공유하는 환경 .

Synchronization 핵심 : orderly execution (순서화) . <= synchronization tool 을 이용

concurrency control (동시성 제어)가 필요하고, os에서 동기화 대표적 tool이 semaphore

최대 buffer size-1 만큼을 동시에 허용

Producer Process와 Consumer Process

```
while (true)
        /* produce an item and put in next_produced */
    while (count == BUFFER_SIZE)   #버퍼가 꽉 차있는지 체크 (꽉 차있으면 더 이상 프로세스할 수 없음
        ; /* do nothing */         #꽉 차있으면 아무것도 안한다
    buffer[in] = next_produced;    #데이터값 하나 저장되면
    in = (in + 1) % BUFFER_SIZE;   #in 포인터 하나 이동
    count++;                       #버퍼 내 채워진 아이템 늘어났으므로
}                                  (counter값은 큐 내에 저장된 아이템 수 = 버퍼사이즈)
```

```
while (true) {
    while (count == 0)             #큐 내에 저장되어있는 아이템이 없으면
        ; /* do nothing */         #아무것도 안함 (즉, 소비할 아이템이 있어야 함)
    next_consumed = buffer[out];   #하나 소비되면
    out = (out + 1) % BUFFER_SIZE; #out 포인터는 deletion (consumption 발생
    count--;
    /* consume the item in next_consumed */
}
```

count라는 변수는 컨슈머 프로세스와 프로듀서 프로세스에서 공유되는 존재. 즉, 프로듀서 쪽과 컨슈머 쪽에서 값을 변경시킬 때, 동시에 변경시킬 수 없고, 한쪽에서 변경된 값을 가지고 또다른 쪽에서 사용해야 한다.

Concurrent Access는 데이터 일관성을 침해하는 부분.: 카운트 연산을 공유하는 데 , **공유하는 변수에 대해서 서로 interleaved 되면 간섭 발생해서 섞일 수 있다.** 프로듀서 프로세스에서 레지스터를 증가시키는 동안, 컨슈머 프로세스가 동시에 수행될 수 있다. 5->6->4 가 되버림. 데이터 inconsistency가 발생할 수 있다. 이게 동기화가 제대로 수행안된 것.

Race Condition : count란 공유 변수를 서로 사용하려는 상황.

Critical Section: 공유하는 자원에 대해 업데이트 현상이 발생하는 것. 즉, 프로세스들이 서로 Access해서 업데이트를 발생시킬 수 있는 코드의 일부. 어떤 특정한 프로세스 하나만 수행하게 해야 한다.. 서로 다른 프로세스 동시 접근 허용되어서는 안된다. Data Consistency 유지 위해 only one process만 허용됨.

모든 프로세스들은 permission을 주는 **entry section**이 존재. **모든 프로세스들은 엔트리 섹션을 수행함에 따라, 크리티컬 섹션을 수행할 수 있을지가 결정됨. 그리고, exit section 수행** (다른 프로세스들이 웨이팅하고있기 때문에)

Solution for Critical Section Problem

1) mutual exclusion(상호배제) 임의의 한 프로세스가 크리티컬섹션을 수행하는 동안에는 어떤 다른프로세스도 크리티컬섹션을 수행할 수 없다. (only one) (간섭받을 수 없다)

2) progress (진행) 노멀하게 잘 흘러가야 한다는 것. 크리티컬섹션을 수행하는 프로세스가 지금 없는데(비어 있는데), 진입(enter)을 하고 싶은데 못한다면 문제가 있다(수행할 수 있어야 한다)

3) Bounded Waiting (제한적 대기. 웨이팅이 무한해선 안되고, 어느 일정시간이 지나면 반드시 차례가 와야 한다.) 이 3가지 만족해야 크리티컬 섹션 프로브럼의 솔루션으로 적합하다

Race condition when assigning a pid : 서로 다른 프로세스에서 fork 명령 동시에 수행시, fork명령의 결과는 child 프로세스의 pid 를 리턴해준다. 포크명령 결과 리턴된 identifier 결과값이 동일하게 나올 수 있다 / **fork명령 시스템콜이 동일한 pid 값을 리턴한다면, fork 명령을 수행해서 identifier 결과내는 접근에서 mutual exclusion이 제공되지 않는 상황**. pid는 유니크하다. (차별화되어야 한다) (동일한 학번은 존재할 수 없다) 그럼에도 불구하고 동일한 identifier 발생한다는 것은 mutual exclusion이 제공되지 않는다는 의미.

Alternating Algorithm

p.11

```
repeat
  while turn ≠ i do no-op;
     CRITICAL SECTION
  turn := j;
     REMAINDER SECTION
until false;
```

#turn 값이 자기자신의 id 아닌 경우 wait
#c/s 수행
#빠져나가면서 상대방의 id값으로 turn값 변경

크리티컬 섹션 수행 할 수 있는 지 permission 얻는 것이 entry section . 빠져나가면서 다른 프로세스가 크리티컬 섹션을 수행해도 좋다 하는게 exit section./ 교대로 발생하기 때문에 Pi가 수행한 다음 Pi가 수행할 수 없음. Alternating Algorithm. / progress, bounded waiting은 보장 안된다.

```
repeat
  flag[i] := true;
  while flag[j] do no-op;
     CRITICAL SECTION
  flag[i] := false;
     REMAINDER SECTION
until false;
```

#자신의 flag를 true로 변경
#상대방 flag를 체크하여, flag[j]면 아무것도 안한다.
#c/s 수행
#자신의 flag 값을 다시 false로 반환.
(c/s 끝났으므로 초기값으로 돌아가는 것)

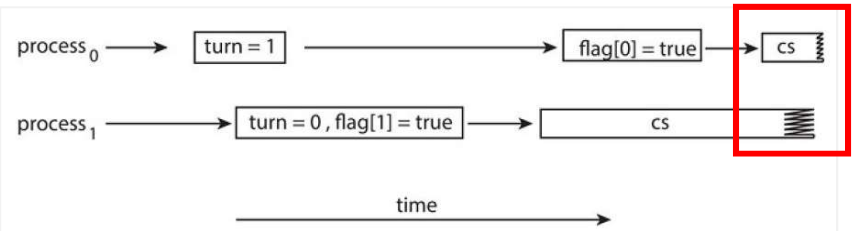
p12. flag는 Pi Pj 입장에서 Ciritcal Section Problem Solution . Flag는 각 프로세스의 상태를 나타냄. Pi에 대한 flag는 flag[i]. / 이 알고리즘도 특정한 프로세스가 ciritical section 수행하고 있을 때는 mutual exclusion 보장. / 만약에 flag[i] flag[j] 둘 다 true면 Pi Pj 둘 다 wait 함. 즉, 그냥 계속 wait 하면서 deadlock 발생 :

```
while (true) {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}
```

#

p.14 알고리즘은 p.12 알고리즘 기본룰을 따른다. normal 상태에서는 flag 값 하나로 다 결정된다. Pi가 크리티컬 섹션 시작 가정. flag i = true -> 데드락 상황 발생않기 위해서 turn이 역할.

Pi Pj가 동시에 수행하려 할 때, turn값이 역할 위해서는 flag I flag j 둘다 true / **글로벌 변수 turn 값을 변경시키는 것은 어느 한 순간에 하나의 프로세스만 접근가능해야 한다. (atomic process)** / turn값이 i로 결정되면 Pi가 turn이 j인지 체크하고, 아니니까 false가 되서 크리티컬 섹션 수행. 따라서 **turn값이 i이면 Pi가 크리티컬 섹션 수행 (Pj가 웨이팅)** 거꾸로 turn값이 j이면 Pi가 기다리고 Pj가 크리티컬 섹션 수행. **turn 체크하는 것은 누가 크리티컬 섹션 수행할 수 있는지 결정해준다.** Peterson's Solution(Algorithm)은 따라서 Mutual Exclusion과 progress가 만족된다.



두 프로세스 동시에 c/s 수행

Figure 6.4 The effects of instruction reordering in Peterson's solution.

p.15 그런데, 프로세스 수행하다보면 문제의 소지 발생할 수 있다. 피터슨 솔루션 엔트리 섹션 내 플래그 값, 턴 값 변경시켜주는 구문이 reordering 하게 되면 문제가 발생할 수도 있다. (리오더링 하기 위해서는 statement간의 dependency가 없어야 함) **turn 값 변경할 때는 서로 다른 프로세스에서는 순차적으로 수행될 수 밖에 없다.**

p.16

Example: data that are shared between two threads

```
boolean flag = false;
int x = 0;
```

✓ Thread 1:
```
while (!flag)
    ;
print x;
```

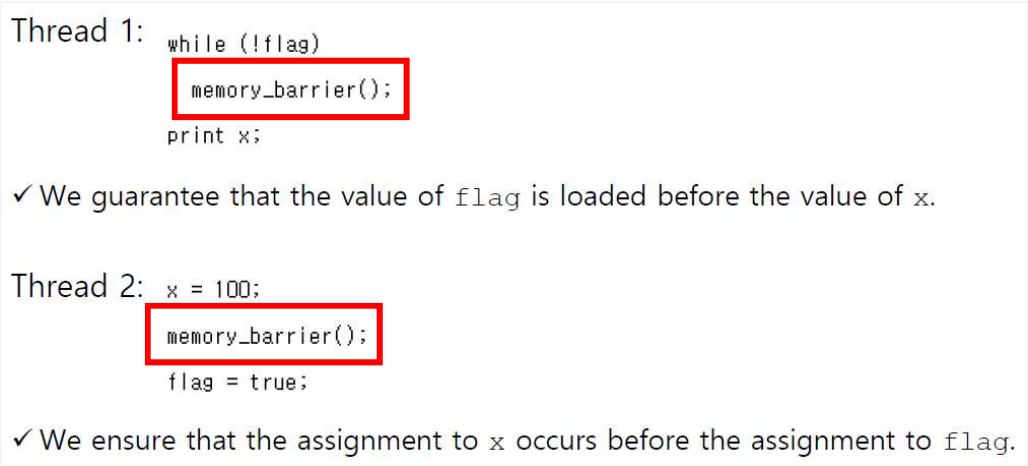
Thread 2:
```
x = 100;
flag = true;
```

thread 1 에서는 wait를 할건지, 빠져나갈지 결정

즉 초기 flag값 false 이므로 flag값이 false 인 동안에는 waiting.

flag가 처음에 false 니까 앞에 !이 붙으면 true가 되고 , **결국 while 조건문이 true 가 되서 계속 수행. thread 2가 flag를 true로 변경시켜주면 thread 1에서 빠져나감** (flag=true) 결국 여기 두 연산에서 의도하는 건 x값이 100으로 assign된 걸 출력하고 싶은 것.

만약 thread2 가 'Reordering'(flag=true와 x=100 순서가 바뀌는 것) 된다면 thread1에서 flag를 true로 인식해서 !flag결과로 빠져나감 . print x = 0이 됨. 결국 100이 아니라 **초기값 0을 출력.**



p.18 이렇듯 **의도되지 않은 상황이 발생하는 것을 방지하기 위해 memory barriers라는 도구를 사용**

Special Hardware Instructions (test_and_set & compare_and_swap)

멀티 프로세서 환경에선 코어1 코어2 코어3 특정한 코어에서 flag값을 변경, 다른 코어에서도 변경하면 문제, 따라서 다른 코어들엔 blocking함. 복잡해진다. 변수를 가지고 변경이 발생할 때는 다른 코어들에서 동시에 못하도록 interrupt 해주어야 함. 복잡해짐. **그러나 special hardware instruction 사용하면, 즉 RAM과 같음. 하나의 저장소를 두고 하나의 function의 형태로 저장되고, 코어들이 여기에서 공유, 업데이트됨. 모든 코어에서 접근해서 업데이트됨. 기본적으로 Atomic 성질 보장**. (하나의 코어가 접근할 때 다른코어들이 접근 못함)

test_and_set solution

Progress, Bounded waiting은 보장 안된다. / Starvation이 발생할 수도 있다 (빨리 인지하는 프로세스가 critical section 수행하기 때문에..)

```
do {
   while (test_and_set (&lock))
          ;   // do nothing

      //    critical section

   lock = false;

      //      remainder section

} while (true);
```

#Pi가 먼저 수행되고 있기 때문에
Pj Pk는 waiting (mutual exclusion 만족)
#Pi 가 c/s 수행
#exit section 수행 : lock을 false로 만들어준다

#그러면 대기하고 있던 프로세스들 중에 먼저 false 인식한 프로세스가 test_and_set 호출한다 -> 해당 프로세스는 c/s 수행한다

compare_and_swap() instruction

마치 system call 사용하듯 사용. 유저 입장에서는 compare_and_swap 사용만 할 줄 알면 됨

파라미터 3개 사용 **int*value** (주소값 받는 call by reference 형태) , **int_expected**. **int new_value** 두 개는 값만 받음 세개 다 integer값. 호출하는 쪽에서는 (&lock, 0, 1)과 같은 형태로 보내줌. 보내주는 값을 그대로 받는다. / 0 값이 int_expected, 1값은 int new_value로 값만 넘겨받는 거. lock이 처음에는 0로 초기화 되어 있음. / Pi쪽에서는 lock이라 부르고, function쪽에서는 value라 부르지만 같은 주소를 가리킴.

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

처음에는 0값을 호출. 받은 쪽에서는 0값을 받아서 int temp에 저장. value==expected (맨 처음엔 0으로 같음)이면 value값을 new_value로 치환 (1로 변경). function 쪽에서 1로 변경해주면 호출하는 쪽에서도 1로 변경 (동일한 주소공간 나타내기 때문) 여기서 value 리턴하면 0값 리턴.

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ;  /* do nothing */

        /* critical section */

    lock = 0;

        /* remainder section */

} while (true);
```

Pi가 critical section 수행하고 있을 때 Pj Pk가 compare_and_swap 호출 -> lock값이 1로 되어 있음 (Pi가 1로 변경시켜뒀음) Compare_and_swap 의 결과가 1이다. 0과 같지 않으면 No operation(그냥 waiting한다) 그러다가 Pi가 exit code 수행 lock이 0으로 바뀌어지면 Pi가 처음 호출할 때와 같아진다. lock이 0이 된 것을 인지한 프로세스가 compare_and_swap에 lock을 0값을 가지고 갔다가 돌아오면 compare_and_swap 값이 0이니 critical section 수행한다. **test_and_set과 같은 상황이지만 instruction의 의미가 다를 뿐**.

swap() instruction도 똑같다. 두개의 값을 받아서 a값을 b로 b값을 a로 바꿔주는 거.

Bounded Waiting mutual exclusion with CAS (세가지 조건 다 만족함)

28쪽에 있는 것은 test_and_set 솔루션을 사용한 것, 26쪽은 compare and swap 사용한 경우. 나머지 부분들은 같고, hardware construction을 test_and_set를 사용했냐 CAS를 사용했냐 차이.

CAS : 이 솔루션은 **n개의 프로세스들이 있다 가정 하에서, 각 프로세스들마다 waiting 값을 유지함. 그리고 lock을 공유. 즉, waiting과 lock은 공유변수이므로 모든 프로세스들이 안다.**

waiting의 의미는 P0부터 Pn-1 까지 waiting[0] ~ waiting[n-1] n개의 프로세스의 상태를 나타냄.

크리티컬섹션 수행 위해서 waiting 한다고 보면 됨. critical section 수행 의지 없다면 원래 waiting 값이 false로 되어있으므로 수행 의지 없음을 명확히 하는 것. **waiting이 false에서 true로 변경이 되는데 그 변경되는 프로세스는 크리티컬섹션 수행하려 한다 명확한 의사표현**이다.

lock이란 integer 변수. 0으로 초기화되어 사용.

key는 각 프로세스에서 사용하는 로컬변수.

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
      key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
    /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;
  else
    waiting[j] = false;
    /* remainder section */
}
```

#entry section
#waiting[i]=true: '수행의지가 있다'
#key 1로 세팅
#자신의 wating[i]이고, key=1이면 수행(처음엔 당연)
lock값이 0이므로 CAS 리턴값이 0, 결국 while문 빠져나옴

c/s 수행 (다른 프로세스들은 lock=1 이므로 waiting)
#exit section
#j값을 증가시키며 waiting 값을 체크
#waiting이 false이면 수행할 의지가 없는 것
#waiting이 true이면 !waiting이 false가 되어 while문 빠져나간다

id값 index값을 증가시켜 가면서 나와 가까운 프로세스 중 c/s 수행할 의지가 있는 프로세스 (자신의 웨이팅 값을 첫번째 문장에서 true로 변경시켜놓은 프로세스를 찾아서, 그 프로세스에게 웨이팅 값을 false로 넣어주면, while문에서 waiting값이 false가 되니까 자연스럽게 크리티컬섹션 수행할 수 있는 것. / 꼭 집어서 해주는 것. 기다리면 자기 순서가 반드시 온다. / 최대 n-1개 프로세스만 기다리면 된다. 왜냐면 빠져나가는 프로세스 가장 가까이 있는 프로세스 선택해서 수행할 수 있도록 해주니까 의사표현만 해 주면 된다.

Semaphore : Synchronization tool로서 사용되는 대표적인 변수로서. 그냥 integer 변수. wait operation(wait())과 signal operation(signal(). function 개념으로 호출됨. **c/s 수행 전 entry section에서는 wait() 수행되고 c/s 수행 후 exit section에서는 signal() 수행된다.**

wait()은 c/s 수행할 수 있는지 권한 부여. signal()은 c/s 수행 후 다 끝났다 의사표현 하는 것. 일반적으로 S=1 로 가정했을 때 wait() signal()수행내용은 p31과 같다. S값 초기값이 0보다작거나 같으면 ; (기다림) while 문 빠져나와서 0으로 만들어줌. .

빠져나가는 연산은 signal() . 증가시켜주는 거. S를 0에서 1로 바꿔줌. 1로 바뀌는 순간, 대기하는 프로세스들 중 인지한 프로세스가 S가 0과 같지 않기 때문에 S를 0으로 만들어놓고 critical secition으로 들어가는 것. **wait()이 locking , signal()이 unlocking으로 이해**하면 될 듯.

공유하는 자원 접근하여 사용할 땐 지금 나만 사용하면 wait()하여 locking해주고 다 사용하면 signal()하여 unlocking하여 다른 프로세스들에게 접근 허용해주는 것.

p32. P1 P2 있는데 각각 signal() wait() 있다. 처음에 synch 값이 0라고 하는 건 , P2는 무조건 처음에 wait (P1이 signal()하여 synch를 1로 변경시켜주면) 그래서 P1과 P2의 관계는.. P2는 wait() 연산을 통과해야 S2를 수행할 수 있으므로 S1 수행한 다음에 S2 수행한다는 순서를 정해줄 수 있다.

Concurrency Contorl 기법과 유사하고, 대표적 기법이 two-phase locking control 이다. 하나가 locking, 또 하나가 unlocking 두개의 연산 수행. 공유하는 데이터에 접근을 해서 데이터값을 변경하는 연산은 critical section 사례와 동일. 현재 프로세스 작업하는 동안 다른 프로세스가 작업 접근 못하도록 locking. 완료되면 unlocking. oS에서도 자원 할당받고 release(해제)하는 것과 동일.

세마포어에서도 Locking에 해당하는 것이 wait(), unlocking에 해당하는 연산이 signal(). 세마포어 S에 대해 어떻게 연산하는지 정의한 것이 두개의 function처럼 정의한 것(코드). **S의 값을 체크하여 0보다 작거나같으면 wait한다. 그렇지 않으면 S값을 1만큼 감소시키고 wait연산 끝냄.**

p.4 counting semaphore 일반적인 integer값을 semaphore값으로 갖고, binary semaphore는 0과 1 사이 값 갖는다. 대부분의 경우는 counting semaphore 사용.

```
P1:                          P2:
    S1;                          wait(synch);
    signal(synch);               S2;
```

synch S1이 끝난후에야 S2가 수행. 순서**. semaphore연산 가지고 프로세스 순서를 정할 수 있다는 것을 보여줌.** synch<=0 이면 waiting. 따라서 synch가 0으로 초기화됐기 때문에 wait 연산 결과는 waiting S2는 더 이상 진행 안되고, **P1이 S1 수행하고 signal(synch) 수행해줬을 때 비로소 synch 값이 1이 되고, P2에서 wait연산 결과로서 wait에서 빠져나가서 S2 수행**.

p5 busy waiting, spinlock cpu자원 낭비. busy waiting를 피해보자 목적

sleep(=wait()) & wakeup(=signal())

p8 semaphore 의 문제점 : deadlock

프로세스들이 지속적 실행하는 것이 아니라, 무한히 기다리는 상태. 모든 프로세스는 살아있는데 실행하는 프로세스는 없는 것. S와 Q

p9 **semaphore 잘못된 사용으로 인한 오류** : wait() signal() 은 크리티컬 섹션 진입하기 전, 그리고 빠져나오면서 각각 수행해야 함. 그런데 만약 1) **두 개 순서가 바뀌었을 때**, signal() 즉 release 하게 되면 모든 프로세스 free pass 즉, mutual exclusion 위배.

두번째로, 2) **signal() 사용해야 하는 부분에 wait() 사용**하면 wait() wait() 됨. 빠져나오면서 wait() 또 취하면 P0가 두번째 단계에서 wait(mutex) (다른 프로세스들은 첫번째 단계에서 wait 하면서 결국 deadlock 발생.

p11 **Bounded buffer problem** : 큐를 사용하는 프로세스 간에 큐에 대한 접근도 동기화 해주고 큐에 대한 연산도 동기화해주는 과정 필요함. n개 데이터 저장공간 존재 가정. 각각의 semaphore 사용 목적 : 1) mutex는 버퍼에 대한 접근 통제, 2) full은 버퍼를 얼마나 채웠느냐를 카운트한다. 0으로 초기화되어있음. , 3) empty는 비어있는 버퍼 양이 얼마나 되는지 카운트한다. 이런 3개의 세마포어 가지고 바운디드 버퍼 문제를 어떻게 해결했는지 살펴본다.

```
while (true) {
        //   produce an item in next_produced
      wait (empty);
      wait (mutex);
        //  add next_produced to the  buffer
      signal (mutex);
      signal (full);
}
```

#critical section 영역

데이터 아이템을 저장할 공간(버퍼)가 있는지를 확인하기 위해서 empty연산. (wait(empty). 그 다음 full값을 증가. (signal(full). 버퍼에 대한 접근을 하기 위해서는 wait(mutex), signal(mutex) 연산 수행. (c/s 영역)

p13. 컨슈머 입장에서는 버퍼의 데이터가 채워진 게 있느냐를 체크 . full값 초기화된 것은 0. 채워진 상태를 체크하는 것이 바로 wait(full)

p14. Readers-Writers Problems : Readers 연산 Writers연산 간에 transaction 필요. 동시에 수행할 수 없다. (충돌)

p15 **Readers-Writers Problem Solution** 첫번째 : 리더에게 권한이 더 부여되는 것처럼 보이는 것. (Reader가 하나라도 발생하면 연속적으로 리딩하는 프로세스 발생할 수 있음. 그러나 Writer는 못함. Writers starvation 우려)(Reader에게 우선 준다) 두번째 : Writer가 준비되면 writer는 최대한 신속하게 작업 수행.(Reader의 starvation 우려)

p16 **Shared Data** 알고리즘 read_count : read 하는 상황 카운트 . mutex 는 1 (read count 0에서 1로 1에서 2로 바뀌는데 이 공유변수가 업데이트 발생하는데, 업데이트 외에는 관여해선 안됨. 업데이트 하는 동안 mutual exclusion 보장하기 위한 세마포어)

rw_mutex 역할은 **writer가 발생할 땐 다른 reader가 발생하지 못하게 wait(rw_mutex),** (반대로

#reading 발생시 read count ++
#임의의 한 read 발생했따 가정 -> wait(mutex) 1에서 0으로

도) (이렇게 해서 mutual exclusion 보장)

```
while (true) {
        wait (mutex) ;
        read_count ++ ;
        if (read_count == 1)  wait (rw_mutex) ;
        signal (mutex)

              // reading is performed

        wait (mutex) ;
        read_count -- ;
        if (read_count  == 0)  signal (rw_mutex) ;
        signal (mutex) ;
}
```

p18 reading 발생할 때 마다 read count ++ 임의의 한 reader 발생했다 가정 .wait(mutex) 1에서 0으로 , read count는 0에서 1로 바뀐다. if read count ==1 이면 wait(rw mutex) 취해준다. 즉, rw mutex가 지금 1로 초기화 되어있기 때문에 이걸 0으로 바꿔준다. 즉, **동시에 writer가 발생하지 않도록 막아준다.**

두번째 reader부터는 if(read count == 1) 만족하지 않기 때문에 reading을 바로 수행하는 과정 시작한다. 리딩 끝내게 되면 read count 값을 decrement 해줘야 한다. if(read count ==0) 즉, 더 이상 reading을 수행할 프로세스가 없다면 , 즉 마지막으로 reading 을 수행하는 프로세스라면 signal(rw mutex)하여 1로 만들어준다. (막아놨던 걸 해제한다)

p19 **Dining Philosophers Problem** : 테이블에 다섯개의 single 젓가락이 있는데, thinking하다가 배고프면 eating하는데 한짝만으론 못하니까 Pi가 식사하기 위해서는 양쪽의 젓가락이 필요. Shared resource가 된다. 이렇게 공유하는 자원을 적절히 사용하여 밥을 먹는 과정이 어떻게 이루어지는가를 보여줌. chopsticks[5]는 1로 초기화된다. Pi가 젓가락 사용하면 Pi+1, Pi-1은 동시에 사용할 수 없다.

p22. 이런 솔루션이 완벽한 솔루션인지 생각을 해봐야하는데, 심각한문제가 있다. Deadlock이 발생할 수 있다. 모든 프로세스들이 기다리는 상황. 즉 thinking을 하다가 배고파서 젓가락을 잡는 과정에서 기다림 발생할 수 있다. 모든 철학자들이 동시에 배가 고파졌다->밥을먹어야겠다->젓가락을잡는다->만약 왼쪽젓가락 먼저 잡는다면 , ->잡았다->오른쪽 잡으려한다 ->다른철학자가 이미 잡았다->왼쪽 잡은채로 계속 기다린다->deadlock

: solution : 1) 많아야 4명의 철학자가 앉게 한다. 2) 양쪽 젓가락 가용한 경우에만 철학자들이 젓가락 잡게 한다. 3) asymmetric solution 사용. 홀수넘버 가진 철학자들은 왼쪽부터 잡게, 짝수넘버 철학자들은 오른쪽부터 잡게.

■ Chapter 8. Deadlock

데드락 다루는 방법들. Prevention (방지) , Avoidance (회피) , Detection (탐지) & Recovery (회복)

Backgrounds : 전체 시스템에서 하나의 프로세스만 생성이 되서 독점적 사용한다면 데드락은 발생하지 않겠지만, multiprogramming 환경에서 발생함. allocation이 원활하다면 문제가 없겠지만, 자원을 서로 사용하려는 얽히는 상황에 의해 발생하는 것이 deadlock.

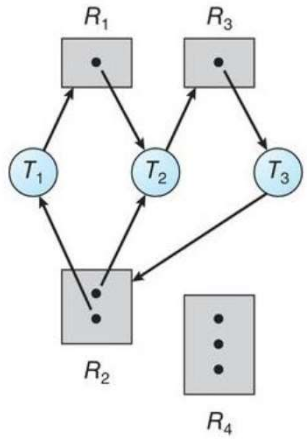
p8 livelock : thread가 지속적으로 수행하려 노력은 하는데, 실패하는 action (결국 wait하는 상황) deadlock은 blocking 상황에 빠지는건데, livelock은 계속 try한다는 점에서 차이

p10 : **deadlock** 정의 : (데드락이 발생하는 조건) **1) mutual exclusion**이 이뤄지는 상황 2) **hold and wait** (특정 자원을 가지고 있으면서, 추가 자원을 요청하는 것) 3) **no preemption (**선점 하지 않는다. 리소스를 뺏지 않는다) 4) **circular wait** (물고 물리는 관계. 즉, Ti는 Tj 가 가진 자원을 요청하고, Tj 는 Tk 가 가진 자원을 요청 ,Tk 는 Ti가 가진 자원을 요청) 이런 4가지 만족하는 시스템 환경에서 deadlock 발생할 수 있는 것.

p11. 특정 스레드가 자원을 홀드하고 요청하는 상황을 Resource Allocation 그래프로 표현. Graph G : (V, E). Thread의 집합, Resources의 집합.

Request edge : 스레드가 리소스를 요청(스레드->리소스 방향) / **Assignment edge** : 리소스가 스레드가 할당이 되어 있다(리소스->스레드 방향).

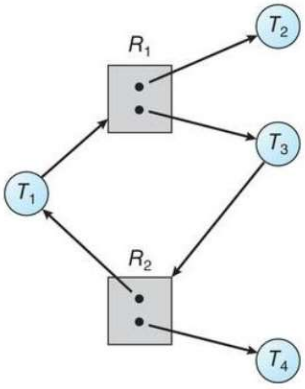
p14. resource allocation graph 데드락이 발생할 수 있는 상황인지 판단. 방향성 가지고 있는 edge



를 따라서 이었을 때 사이클 존재여부에 따라 **사이클이 존재하면 deadlock이 발생할 수 있고**, 사이클이 없다면 발생하지 않는다. 또한 이 resource allocation graph는 resource type이 1개냐 여러개냐에 따라서 그림의 의미는 달라질 수 있는데, **1개의 인스턴스만 가지고 있으면 사이클의 의미는 곧 데드락을 의미**하는 것.

예) 추가적인 리소스 요청하는 스레드 T3가 R2를 요청하는 상황. T3->R2 화살표 추가. 그림 8.5. 이로써 사이클이 생긴다. 이런 경우 **데드락이 발생**했다고 얘기한다 **(사이클이 있다고 다 데드락은**

아니지만 여기서는 인스턴스들이 전부다 사용됐기 때문에)그런데 R4는 독립적으로 Resource 존재하는 것. 상관없는 것.



p16. 사이클이 존재하지만 **no deadlock** . 인스턴스가 2개 이상인 경우로서 지금 현재로선 모든 인스턴스가 스레드에 할당이 되어있는데 , **사이클을 형성하는 구성원으로 포함이 되어있으면 데드락이 발생하겠는데, T2, T4는 사이클에 포함되어있지 않다**. T2 T4 수행하고 나서 가용자원을 내놓으면, Request edge가 assignment edge로 바뀌면서 no cycle로 바뀌게 되어 데드락이 없어진다.

p18 **Deadlock Prevention** : 데드락 방지. 데드락 발생조건을 만족하지 않는다면 가능.

1) **mutual exclusion** : sharable mode 에서는 데드락이 없다. (리소스를 공유하니까)

2) **hold and wait** : hold만 하든지 wait만 하든지. 둘 중 하나만 가능하도록. 각 스레드가 작업 시작하기 전에 필요한 모든 리소스를 미리 할당받는 것.

3) **no preemption** : 뺏어서 할당해라. 그러나 뺏더라도 기본 룰이 있어야 한다. preemption 하는 두 개 상황은 같다. 뺏기는 조건이 상대방입장에서 보냐 , 내입장에서 보냐 차이. 1) 이미 한 스레드가 리소스를 홀드하고 있고, 또다른 리소스를 요청하지만 바로 할당될 순 없는 것. 그런데, 현재 내가 홀드하고 있는 자원은 뺏기는 것. 즉, 프로세스 수행이전상태로 rollback 하는 것.

alternatively ~ 부분은 뺏는 입장에서 설명한 것. 내가 요청한 자원을 홀드하고 있는 스레드가 또다른 자원을 기다리는 상황이면, 그걸 뺏어온다.

4) Circular wait : 물고 물리는 상황이 아니라 linear wait 상황만 만들어 주는 것. Resource 각각에 매핑되는 unique integer number 부여. 자원을 요청할 때, 임의의 한 스레드가 Ri에 자원을 가지고 있다 가정. 추가적으로 자원 요청할 땐 Ri 가진 이후에는 Ri에 매핑되는 넘버보다 큰 넘버를 가진 리소스에만 요청할 수 있다.

Deadlock Avoidance

p21 회피가 어렵다. 추가적인 정보들이 필요하기 때문. 각 프로세스들이 앞으로 필요한 리소스들이 어떤 것들이 있으며, 얼마나 많은 양이 필요할 지 모르기 때문. 따라서 사용되기 어렵다. 그러

나 이론적으로 공부하는 것.

사이클을 형성한다는 것 자체는 unsafe state. deadlock이 걸린 상태는 unsafe에 포함되는 개념. 그래프 기반으로 평가하는 방법, 알고리즘 기반으로 보는 방법.

claim edge : Pi에서 Rk 로 요청하는 Request Edge (wait) , Rk 에서 Pj 로 할당되어 있는 Assignment Edge (hold) 에 추가적으로 claim edge를 볼 것이다. 지금 현재상태가 아니라, 미래에 요청할 수 있다 는 것을 표현하는 개념. 미래에 발생할 것을 표현. (미래에 발생할 것을 인지하고 시뮬레이션 해보는 것)(실제적으로 적용하는 상황은 아니고)

p25 algorithm 으로 deadlock avoidance 하겠다 : 각 프로세스들이 작업을 완료할 때 까지 얼마나 많은 자원들이 필요로하는지 모르는데.. 최대 얼마나 많은 리소스들을 필요하냐 ! (해당 리소스타입의 인스턴스)

스레드 개수 n , 리소스 타입의 개수 m

Available(배열 변수): 현재시스템 내에서 가용한 자원의 개수. available[j] 라고 표현. j는 서로 다른 리소스타입을 의미.

Max: 각 스레드(프로세스)가 작업을 끝내는데 필요한 최대 리소스 타입.

Allocation: 현재 각 스레드에 할당되어 있는 리소스 개수

Need: Need[i][j] : 최대 필요로 하는 자원에서 현재 할당된 자원 뺀 것.

Safety 알고리즘 : 현재 각 스레드에 자원들이 할당되어 있을 때 , 예를 들어 프로세스 10개가 있는데 원하는 자원을 동시에 만족시켜줄 수 있따면 문제가 없다 그러나 실제로는 시스템에 자원은 한정되어 있으므로 한정된 자원을 가지고 스레드들에 하나하나 만족시켜줘야 한다.(=작업을 완료하게 해준다) (one by one 자원 할당) 이렇게 전체 스레드들에 순서대로 작업 완료. 이러한 순서를 구해주는 것을 safety algorithm 이라 한다. Available 값은 계속 커진다. (작업이 완료되는 프로세스들이 늘어날수록 자원은 쌓인다) Available 초기값들 변해가며 자원 관리한다. 완료된 자원 상태를 true 로 변화. 모든 스레드 피니시 값이 true가 되면 safe state가 된다.

Resource Request 알고리즘 : 어떤 특정한 스레드가 자원을 조금 더 요청했다 가정. 그 리퀘스트를 시스템 입장에서 허용을 할 거냐. (할당해 줄거냐) 결정하는 것. 리퀘스트 만족시켜줬다 가정하고서 safe algorithm 실행시켜보고 , need 값 줄어들고 가용자원 줄어들고 , 이러한 변경된 값을 가지고 safety algorithm 수행해본다. safe sequence 구해보고, 구해지면 완료.

Deadlock Detection : 데드락을 풀게하고 리커버리. 뱅커스와 비슷한데, 뱅커스는 미래에 대한 정보를 안다고 가정했지만, 여기서는 현재 주어진 정보만 가지고 데드락이 주어질 수 있을 거 같다는 판단을 하는 것. 1) 그래프 기반 (리소스 타입 1개일때만) 2) 알고리즘 기반(리소스타입 2개 이상)

p32. wait for graph는 resource allocation graph를 조금 단순화한 형태이다

p33. Resource allocation graph에서는 리소스와 스레드 관계 모두 표현. wait-for-graph에서는 스레드간 관계만 표현 ..(T2가 가지고있는 리소스를 T1이 wait 하고 있다)

p34. 알고리즘 기반.

Available : 가용 자원 개수 / Allocation: 각 스레드에 어떤 리소스가 얼마나 할당되어 있는가 (현재상태) / Resource: 각 스레드가 자원을 얼마나 요청하고 있는가. (현재)

p35 work보다 작거나 같은 request 값을 찾는 것. request값을 만족시킨다고 했을 때 그 작업은 끝나고, 해당 스레드가 갖고 있는 allocation 값을 work에다가 더해준다. 나머지 똑같다.

p38 얼마나 자주 데드락이 발생하느냐, 얼마나 많은 스레드가 영향받느냐 에 따라서 데드락 detection algorith을 invoke 할지를 생각해볼수 있다. 그러나 considerable overhead 발생한다.

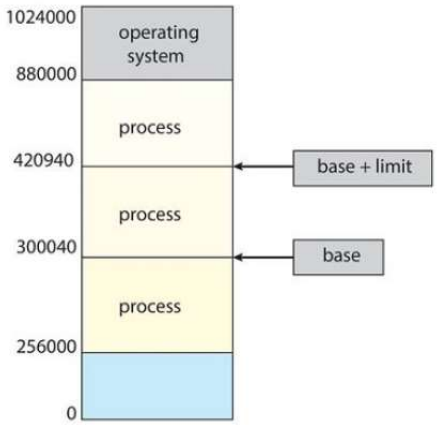
p39 recovery

p40 그냥 abort 해서 데드락 없애는 것. (중요도, 속성들을 고려하여)

Resource Preemption : 리소스를 그냥 뺏는 것. selecting victim : 비용을 최소화 , rollback : 이전상태 돌아가는 것 또는 abort 하는 것 , starvation : 리소스 뺏기는 스레드는 starvation 발생.

■ Chapter 9 Main Memory

시작 주소에 대한 내용을 base register 에서 관리를 한다 Limit register 는 프로세스의 사이즈다. 프로세스가 끝나는 위치는 base + Limit 이다. (시작주소 + 크기 = 끝 주소) 이것이 일반적인 유저



의 관점이다.

process 2 특정한 instruction이 있다 (i = i +1) 해보자 . 전체 프로그램 내 어딘가 위치할 것. (프로세스 2에 위치한다 가정) **명령어(statement) 하나는 base register와 base+limit범위 내에 존재해**

야 process 2 범위 내에 존재한다 말할 수 있다.

p6 Address Binding : **유저가 보는 주소값과 시스템이 보는 주소값은 다르다**. 유저가 보는 것은 시작은 0번지 끝주소는 알파번지 개념.(논리주소) 즉, 유저가 보는 주소값이 시스템 보는 주소 체게로 변경이 되어야 한다. 시스템 관점에서는 물리주소로 변경되어야.

p8 Binding 되는 과정에 시기 compile 시에 되느냐 load 시에 되느냐, execution 시에 되느냐 에 따라 다양. 바인딩 개념은 프로그래밍 언어 속성에 따른 dependency를 가지고 있다.

p9 논리주소를 물리주소로 매핑하는 것이 9장에서 배울 내용 페이징 기법 사용한다.memory management unit(MMU).

p11. **memory protection**: 프로세스간 영역 침범 안하도록 영역 설정 명확히. base와 base+liimt 범위 벗어나지 않게.

p.13. **여러 개 fixed sized n개 파티션을 만들고 , 각 파티션은 정확히 하나의 프로세스만 저장**. 최대 로드할 수 있는 프로세스 수는 n개 degree of multiprogramming 이란 것은 메인메모리에 상주하고 있는 프로세스가 몇 개인지를 의미 결국 파티션 개수만큼만 로딩될 수 있는 것.

.p14 프로세스 끝나면 hole -> 가용자원이므로 새로운 프로세스 할당 -> 남은 홀이 100m, 150m 있는데 200m 프로세스 할당하려 하면, 할당이 안된다 (연속적인 공간이 아니라서) 이러한 문제를 해결하기 위해서 제안하는 비연속적 할당 기법이 페이징 기법 연속적 공간 필요없이 비연속적인 할당 기법

p15. Memory allocation 할당 유형. 앞에서는 고정크기 나누어 할당 . 프로세스를 할당할 수 있는 충분한 사이즈의 첫번쨔 홀을 찾는다. 찾은 첫번쨔 공간에 할당하는 것이 First Fit . 할당하고 남는 공간이 가장 적은 것이 Best Fit. 가장 큰 공간 찾는게 Worst Fit .

p16. **할당하며 발생하는 hole 관련, 문제점**. **1) external fragmentation (외부 단편화)** : 100m 150m 홀이 있고 200m 할당하려 하는 상황. 프로세스를 할당할만한 충분한 공간이 있음에도 연속적이지 못하여 할당하지 못하는 상황. **2) internal fragmentation(내부 단편화)** : 120m 할당할 때 First fit 가정 하면 100m 못하고 150m 할당할 수 있다. 150m 에 할당하고 나면 나머지 30m 남게 된다. 하나의 프로세스에는 정확히 하나의 프로세스만 할당할 수 있으므로, 30m은 버려야 한다. 그래서 이러한 상황이 internal fragmentation 이라 한다. (재활용이 안된다)

p17. external 발생하는 경우에는 연속적인 공간을 만들어준다면, 즉, process 9를 이동시켜주는 것. 이렇게 해결하는 방법을 **Compaction.** 이라 한다. 스토리지에서 옮기는 것이 아니라, 메인메모리에서 옮기는 것이라 쉬운작업은 아니다 (주소값이 변하는 것) (only if relocation is dynamic)

페이징 기법에서는 external fragmentaion은 없다 internal fragmentation은 있다.

p18. 프로세스를 나눠서(쪼개서) 집어넣기 메인메모리를 fixed-size blocks로 다 쪼갠다. page(유저관점)=frame(시스템관점). 이 두개 사이즈는 동일해야 한다. 나누다보면 맨 마지막 페이지는 사이

즈가 작다.

Page number : 페이지 개수. 시작주소가 어딘지는 page number를 통해 찾는다. (페이지 넘버는 그냥 눈에 보이는 페이지 개수)

Page Offset : 페이지 사이즈(덩어리 크기). 작성한 프로세스 가정했을 때, i=i+1 시작위치에서 알파만큼 떨어져있다. 시작위치로부터 떨어져 있따. 항상 알파번지에 위치하고 있다. 유저입장에서는 항상 제로번지부터 시작. 알파번지만큼 떨어져있다는 정보를 page offset에서 가지고 있다. 각각의 덩어리 내에서의 위치정보를 의미한다. Page offset이 addressing 할 수 있는 하나의 사이즈가 페이지사이즈다.

Page Table : 덩어리 관리. 유저입장에서 보는 페이지들은 피지컬 메모리들로 할당하는 걸 관리해주는 것을 **page table**이라 한다. page table은 유저 page가 어디 저장되어 있는 지를 알고 있다. 프로세스 하나하나마다 page table이 따로따로 존재한다. (kernel 단에 존재)

page offset 은 상대주소값 가지고 있어서 시작위치는 0을 갖고 있다. 논리주소를 물리주소로 변환시켜주는데, i=i+1은 항상 알파만큼 떨어져 있다.

페이징 기법이란 것은 프로세스 전체 덩어리를 페이지 또는 프레임이란 것으로 동일 사이즈로 나눠서 메인메모리에 흩어져 저장. 흩어져 있는 페이지 (프레임)들은 page table에서 관리한다. 프로세스들마다 나눠지는 페이지 개수는 다르지만, table 자체가 충분히 사이즈가 크다. page offset 비트 수가 작아지면 page size가 작아지고, page number는 커진다. page table 사이즈는 커진다.

페이징기법에서 논리주소 체계에서 앞부분은 page number, 뒷부분은 page offset. page offset이란 것은 page하나의 사이즈를 의미. 그러한 논리주소 체계가 page number에 할당된 비트수와 page offset에 할당된 비트수가 변한다는 것의 의미를 이해 **pageoffset에 할당된 비트수가 줄어든다는 것은 페이지 사이즈 크기가 작아진다. 반대로 페이지넘버 비트수가 커지는 것이므로 페이지 테이블의 크기가 커진다.** 의미. 페이지 하나의 사이즈가 작아지면, 즉, page offset의 비트수가 줄어들면 page 사이즈가 작아지는 것. 페이지 단위 나눠진 메인메모리 내에 프로세스 읽어올 때는 페이지 단위로 읽어온다. 페이지사이즈가 크면 몇 개만, 사이즈가 작으면 많이 읽어와야 한다. 큰덩어리 작은덩어리 똑 같은 덩어리 읽어오는 거지만, access 횟수가 늘어난다. **페이지사이즈가 작으면 i/o time 커진다**

p24. page size가 2048바이트면 2^11 . 페이지사이즈에 해당되는 것이 페이지 offset. 즉, 페이지옵셋에 해당되는 비트수가 11비트.

internal fragmentation = 2048 / 72766 몫 35, 나머지 1086. 프레임개수는 36개이고 2048-1086=962 만큼이 internal fragmentation 이다.