6장. 프로세스 동기화 (Process Synchronization)

순천향대학교 컴퓨터공학과
이 상 정

강의 목표 및 내용

- 목표
  - 임계 구역 (critical section) 문제를 소개
  - 임계 구역 문제에 대한 소프트웨어 및 하드웨어 해결책
  - 원자적 트랜잭션을 소개하고 원자성을 보장하기 위한 기법
- 내용
  - 배경
  - 임계 구역 문제
  - 피터슨의 해결안
  - 동기화 하드웨어
  - 세마포
  - 고전적인 동기화 문제들
  - 모니터
  - Linux, Pthreads 동기화

배경 (Background)

- 공유 자료를 병행 접근(concurrent access)하면 자료의 불일치(data inconsistency)를 초래
- 자료의 일관성(data consistency)을 유지하려면 협력적인 프로세스들(cooperating processes)의 바른 순서로 수행(orderly execution)을 보장하는 메커니즘이 필요

- 경쟁 상황(race condition)
  - 여러 개의 프로세스가 동일한 자료를 접근하여 조작하고, 그 실행 결과가 접근이 발생한 특정 순서에 의존하는 상황
  - 경쟁 상황으로부터 보호하기 위해 한 순간에 하나의 프로세스만이 공유 자료를 조작하도록 보장하도록 프로세스들을 동기화(synchronization)

생산자-소비자 문제 예 (1)

- BUFFER_SIZE 개까지 버퍼에 저장하도록 수정
  - 0으로 초기화되어 있는 counter라는 정수형 변수를 추가
  - 버퍼에 새 항목을 추가 시 counter 증가, 삭제 시 counter 감소
- 생산자 코드

```
while (TRUE)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer [in ] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

생산자-소비자 문제 예 (2)

- 소비자 코드

```
while (TRUE)
{
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}
```


생산자-소비자 문제 경쟁 상황 (1)

- counter++ 기계어 수준 명령 구현

$$register_1 = counter$$

$$register_1 = register_1 + 1$$

$$counter = register_1$$

- counter-- 기계어 수준 명령 구현

$$register_2 = counter$$

$$register_2 = register_2 - 1$$

$$counter = register_2$$

생산자-소비자 문제 경쟁 상황 (2)

- "counter++"와 "counter—" 문장을 병행하게 실행하면 기계어 수준 명령들을 임의의 순서로 뒤섞어 순차적으로 실행

T_0 : 생산자가 $register_1$ = counter를 수행 {$register_1 = 5$}
T_1 : 생산자가 $register_1 = register_1 + 1$을 수행 {$register_1 = 6$}
T_2 : 소비자가 $register_2$ = counter를 수행 {$register_2 = 5$}
T_3 : 소비자가 $register_2 = register_2 - 1$을 수행 {$register_2 = 4$}
T_4 : 생산자가 counter = $register_1$을 수행 {$counter = 6$}
T_5 : 소비자가 counter = $register_2$를 수행 {$counter = 4$}

- 부정확한 결과
  - 실제로 5개의 버퍼가 채워져 있지만 4개의 버퍼가 채워져 있는 것을 의미하는 "counter= = 4"인 부정확한 상태에 도달
  - T_4와 T_5의 문장 순서를 바꾸면, "counter = = 6" 인 부정확한 상태에 도달

임계 구역 문제 (Critical-Section Problem)

- n 개의 프로세스가 공유 자료 접근을 위해 경쟁할 때 각 프로세스는 공유 자료를 접근하는 임계구역(critical section)이라고 부르는 코드 부분을 포함
- 임계 구역 문제
  - 한 프로세스가 자신의 임계 구역에서 수행하는 동안에는 다른 프로세스들은 그들의 임계 구역에 들어갈 수 없도록 보장

- 각 프로세스는 자신의 임계 구역으로 진입하려면 진입 구역(entry section)에서 진입 허가를 요청
- 임계 구역 뒤에는 퇴장 구역 (exit section)

```
do {
    진입 구역
        임계 구역
    퇴장 구역
        나머지 구역
} while (TRUE);
```

그림 6.1 전형적인 프로세스 P_i의 일반적인 구조

임계 구역 문제에 대한 해결안

- 다음의 세 가지 요구 조건을 충족해야 함
  1. 상호 배제 (mutual exclusion): 프로세스 P_i가 자기의 임계 구역에서 실행된다면, 다른 프로세스들은 그들 자신의 임계 구역에서 실행될 수 없음
  2. 진행 (progress): 자기의 임계 구역에서 실행되는 프로세스가 없고, 자신의 임계 구역으로 진입하려고 하는 프로세스들이 있다면 이들 프로세스들 중 임계 구역으로 진입을 선택하고, 이 선택은 무한정 연기될 수 없음
  3. 한정된 대기(bounded waiting): 프로세스가 자기의 임계 구역에 진입하려는 요청을 한 후부터 그 요청이 허용기 전까지 다른 프로세스들이 그들 자신의 임계 구역에 진입이 허용되는 횟수에 제한이 있어야 함

피터슨의 해결안(Peterson's Solution)

- 고전적인 소프트웨어 기반 해결책
  - 두 개의 프로세스로 한정
- 두 프로세스가 공유하는 두 개의 자료 항목
  - int turn;
  - boolean flag[2]
  - LOAD와 STORE 명령이 원자적이라고 가정
- 변수 turn은 임계 구역으로 진입할 순번을 나타냄
  - turn==i이면 프로세스 P_i가 임계 구역에서 실행
- flag 배열은 프로세스가 임계 구역으로 진입할 준비가 되었다는 것을 나타냄
  - flag[i]가 true이라면 이 값은 P_i가 임계 구역으로 진입할 준비가 되었다는 것을 나타냄

피터슨의 알고리즘

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

        임계 구역

    flag[i] = FALSE;

        나머지 구역

} while (TRUE);
```

그림 6.2 피터슨의 해결안에서 프로세스 P_i의 구조

동기화 하드웨어 (Synchronization Hardware)

- 많은 시스템에서 임계 구역 코드를 지원하는 하드웨어를 제공
- 단일 처리기 환경에서는 공유 변수가 변경되는 동안 인터럽트 발생을 허용하지 않음으로써 간단히 해결
  - 현재 실행되는 코드가 선점(preemption)없이 순서적으로 실행
  - 다중 처리기 환경에서는 너무 비효율적
    - 인터럽트의 불능화 메시지를 모든 프로세서에 전달되게 하기 때문에 상당한 시간을 소비
- 많은 현대 기계들은 특별한 원자적(atomic) 하드웨어 명령어들을 제공
  - 원자적 (atomic) = 인터럽트 되지 않음 (non-interruptable)
  - 한 워드(word)의 내용을 검사하고 변경 명령어 (test and set instruction)
  - 두 워드의 내용을 원자적으로 교환 (swap instruction)

락(Lock)을 사용한 임계 영역 문제 해결

- 프로세스는 임계 영역 진입 전에 반드시 락(lock)을 획득
- 임계 영역 나올 때는 락을 방출

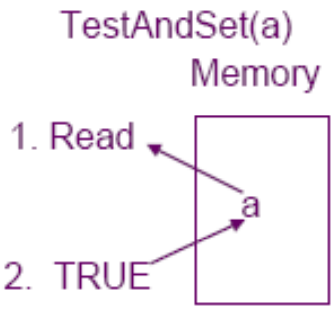
```
do {
    락 획득
        임계 영역
    락 방출
        나머지 영역
} while (TRUE)
```


TestAndSet 명령어 정의

- 명령어가 원자적(atomically)으로 실행

```
boolean TestAndSet(boolean *target)  {
    boolean rv = *target;
    *target = True;
    return rv;
}
```

그림 6.4 TestAndSet() 명령어의 정의



TestAndSet 명령어를 사용한 상호 배제 구현

- FALSE로 초기화되는 lock이라는 Boolean 변수를 선언하여 상호 배제를 구현

```
do {
    while (TestAndSet(&lock))
        ; // 아무일도 하지 않음

        // 임계 구역

    lock = FALSE;

        // 나머지 구역
} while (TRUE);
```

그림 6.5 TestAndSet() 명령어를 사용한 상호 배제 구현

Swap 명령어 정의

- 두 개의 워드의 내용에 대해 작동

```
void Swap(boolean *a, boolean *b)  {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

그림 6.6 Swap() 명령어의 정의

Swap 명령어를 사용한 상호 배제 구현

- 전역 Boolean 변수 lock을 선언하고 false로 초기화하고, 각 프로세스는 지역 Boolean 변수 key를 가지고 있다.

```
do {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);

        // 임계 구역

    lock = FALSE;

        // 나머지 구역
} while (TRUE);
```

그림 6.7 Swap() 명령어를 사용한 상호 배제 구현

한정된 대기 조건을 만족하는 상호 배제 (1)

- TsetAndSet() 명령어를 사용하고, 한정된 대기 조건을 만족하는 상호 배제
  - 한 프로세스 i가 임계 구역 떠날 때 waiting 배열 순회 (i+1, i+2, ……, n-1, 0, ……, i-1)
  - 순회 중 waiting [j]가 true인 첫번째 대기 프로세스가 임계 영역에 진입하고 witing[j]는 false로 지정
  - 임계 영역에 진입하고자 하는 프로세스는 최대한 n-1 양보

한정된 대기 조건을 만족하는 상호 배제 (2)

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;

        // critical section

        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;

        // remainder section

} while (TRUE);
```


세마포 (Semaphore)

- 하드웨어를 기반(TestAndSet과 Swap 명령어 등 사용) 해결안들은 응용 프로그래머가 사용하기에는 복잡
  - 임계구역 진입 전 바쁜 대기(busy waiting) 사용
- 바쁜 대기가 필요없는 세마포(Semaphore)라고 하는 동기화 도구를 이용
- 세마포 S는 정수 변수로서두 개의 표준 원자적 연산 wait()와 signal()로만 수정
  - 원래는 P()와 V()라고 함

wait()와 signal()의 정의

- wait()와 signal() 연산 시 세마포의 정수 값을 변경하는 연산은 반드시 원자적으로(분리되지 않고) 수행
- 세마포의 값이 0이 되면 모든 자원이 사용 중임을 나타내고 이후 자원을 사용하려는 프로세스는 세마포 값이 0보다 커질 때까지봉쇄됨

```
wait(S) {
   while (S <= 0)
      ; // 아무런 작업을 하지 않음
   S --;
}

signal(S) {
  S++;
}
```


동기화 도구 세마포

- 카운팅(counting) 세마포와 이진(binary) 세마포를 구분
  - 카운팅 세마포의 값은 제한 없는 영역(domain)을 가짐
  - 이진 세마포의 값은 0과 1사이의 값만 가짐
    - 이진 세마포가 상호 배제(mutual exclusion)를 제공하는 락이기 때문에 mutex 락이라고도 함
- 세마포를 이용한 상호배제 구현
  - Semaphore S; // 1로 초기화
  - wait (S);
    임계 구역 (Critical Section)
    signal (S);

세마포 구현 (1)

- 앞의 세마포 정의의 주된 단점은 이들이 모두 바쁜 대기(busy waiting)를 요구
  - 바쁜 대기는 다른 프로세스들이 생산적으로 사용할 수 있는 CPU 시간을 낭비
  - 프로세스가 락을 기다리는 동안 회전하기 때문에 이런 유형의 세마포를 스핀락(spinlock)이라고도 함
    - 짧은 시간 락을 기다리는 경우 문맥 교환이 필요하지 않아 스핀락이 유용
- 바쁜 대기 대신에 프로세스는 자신을 봉쇄하고(block) 프로세스를 세마포에 연관된 대기 큐(waiting queue)에 삽입
  - 프로세스의 상태를 대기 상태로 전환
  - 이 후에 제어가 CPU 스케줄러로 넘어가고, 스케줄러는 다른 프로세스를 실행하기 위하여 선택

세마포 구현 (2)

- 대기 큐와 연관된 세마포의 자료
  - 세마포 값을 나타내는 정수 value
  - 대기 큐를 가리키는 포인터 리스트 list

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- 세마포 관련 연산
  - block() 연산
    자기를 호출한 프로세스를 중지하고 대기 큐에 삽입
  - wakeup(P) 연산
    봉쇄된 프로세스 P의 실행을 재개
    대기 큐에서 프로세스를 제거하고 준비 완료 큐에 삽입

바쁜 대기가 없는 wait()와 signal()의 정의

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            이 프로세스를 S->list에 넣는다;
            block();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            S->list로부터 하나의 프로세스 P를 꺼낸다;
            wakeup(P);
        }
}
```


교착 상태 (Deadlock)

❑ 교착 상태 (deadlock)

- 두 개 이상의 프로세스들이, 오로지 대기중인 프로세스들 중 하나에 의해서만 야기될 수 있는 사건(signal() 연산 실행)을 무한정 기다리는 상황
- 두 개의 프로세스 P_0과 P_1에서 1로 초기화된 세마포 S와 Q를 접근하는 시스템 예

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

기아 (Starvation)

- 기아(starvation)
  - 교착 상태와 연관된 무한 봉쇄(indefinite blocking)
  - 프로세스들이 세마포에서 무한정 대기하는 것으로, 프로세스가 중지된 세마포 큐에서 제거되지 않음

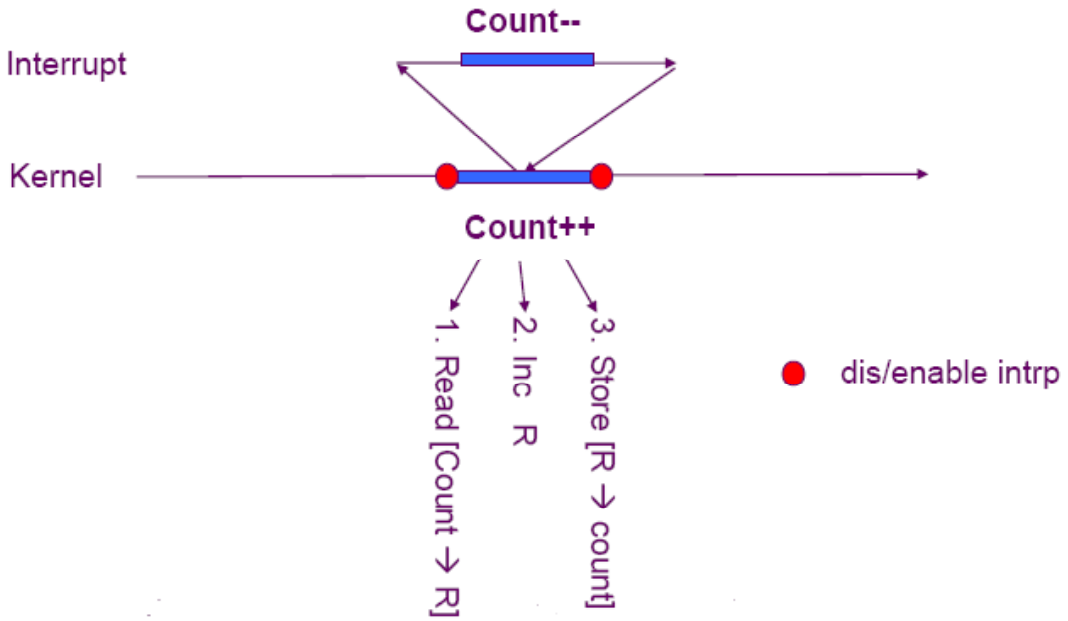
임계 구역 문제 발생

- 운영체제에서 임계 구역 문제가 언제 발생하는가?

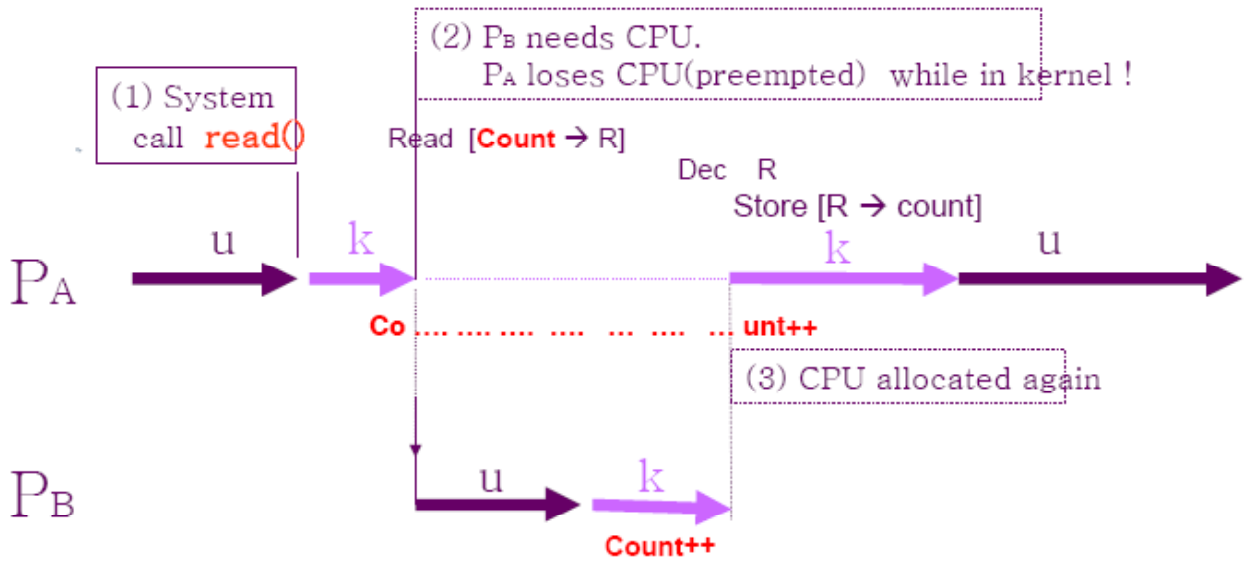
1. 커널에서 인터럽트 루틴 처리
2. 프로세스가 커널에서 선점
   - 시스템 호출 중에 선점
3. 다중 처리기(multiprocessor)
   - 공유 메모리에서 커널 데이터

커널에서 인터럽트 루틴 처리

1. Read [Count → R]
2. Dec R
3. Store [R → count]

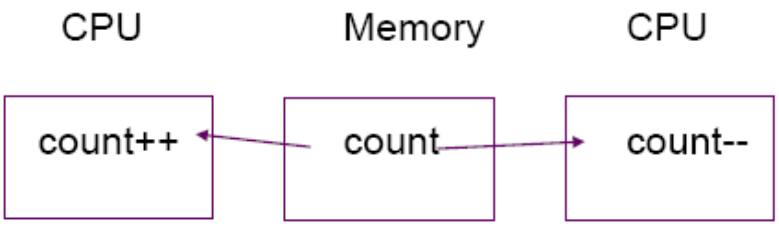


프로세스가 커널에서 선점



1. Read [Count → R]
2. Dec R
3. Store [R → count]

다중 처리기



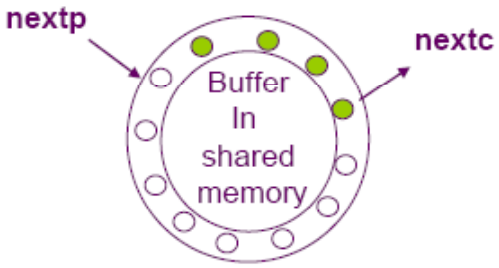
고전적인 동기화 문제들 (Classic Problems of Synchronization)

- 널리 사용되는 대표적인 동기화 문제들을 제시
  - 새로 제안된 거의 모든 동기화 방법들을 테스트하는 데 사용
  - 여기서는 동기화를 위하여 세마포가 사용

- 유한 버퍼 문제 (Bounded-Buffer Problem)
- Readers-Writers 문제 (Readers-Writers Problem)
- 식사하는 철학자들 문제 (Dining-Philosophers Problem)

유한 버퍼 문제 (Bounded-Buffer Problem)

- n 개의 버퍼들로 구성된 풀(pool)이 있으며 각 버퍼들은 한 항목(item)을 저장
- mutex 세마포는 버퍼 풀을 접근하기 위한 상호 배제 기능을 제공하며 1로 초기화
- empty 세마포는 세마포들은 비어 있는 버퍼의 수를 기록하며 n 값으로 초기화
- full 세마포는 꽉 찬 버퍼의 수를 기록하며 0으로 초기화



유한 버퍼 문제 – 생산자 프로세스

```
do {
        ...
    // produce an item in nextp
        ...
    wait(empty);
    wait(mutex);
        ...
    // add nextp to buffer
        ...
    signal(mutex);
    signal(full);
} while (TRUE);
```

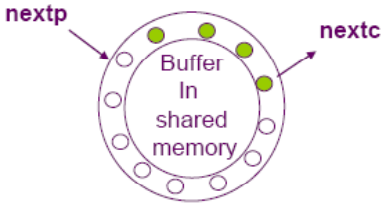


그림 6.10 생산자 프로세스의 구조

유한 버퍼 문제 – 소비자 프로세스

```
do {
    wait(full);
    wait(mutex);
        ...
    // remove an item from buffer to nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    // consume the item in nextc
        ...
} while (TRUE);
```

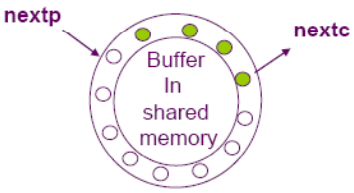


그림 6.11 소비자 프로세스의 구조

유한 버퍼 문제 – 생산자,소비자

```
do {
        ...
    // produce an item in nextp
        ...
    wait(empty);
    wait(mutex);
        ...
    // add nextp to buffer
        ...
    signal(mutex);
    signal(full);
} while (TRUE);
```

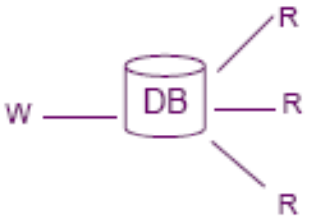
그림 6.10 생산자 프로세스의 구조

```
do {
    wait(full);
    wait(mutex);
        ...
    // remove an item from buffer to nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    // consume the item in nextc
        ...
} while (TRUE);
```

그림 6.11 소비자 프로세스의 구조

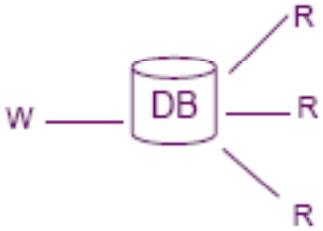
Readers-Writers 문제 (Readers-Writers Problem)

- 하나의 데이터베이스는 다수의 병행 프로세스들 간에 공유
  - 일부 프로세스들은 데이터베이스의 내용을 읽기만 수행
    => Readers
  - 어떤 프로세스들은 데이터베이스를 갱신(즉, 읽고 쓰기) 수행
    => Writers
- 문제
  - 하나 이상의 reader가 동시에 공유 자료를 접근 허용
  - 오직 하나의 writer만 공유 자료에 접근
    - Writer와 다른 프로세스(write 또는 reader) 동시 접근 불능



Readers-Writers 문제, 공유 자료

- 공유 자료
  - 데이터베이스
  - mutex 세마포
    - 1로 초기화
    - readcount를 갱신할 때 상호 배제
  - wrt 세마포
    - 1으로 초기화
    - Writer를 위한 상호 배제 세마포
    - 또한 임계 구역으로 진입하는 첫 번째 reader와, 임계 구역을 빠져 나오는 마지막 reader에 의해서도 사용
  - readcount 정수
    - 현재 몇 개의 프로세스들이 객체를 읽고 있는지 알려줌
    - 0으로 초기화



Writer 프로세스

```
do {
    wait(wrt);
        ...
    // writing is performed
        ...
    signal(wrt);
} while (TRUE);
```

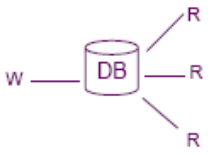


그림 6.12 Writer 프로세스의 구조

Reader 프로세스

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
        ...
    //reading is performed
        ...
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
} while (TRUE);
```

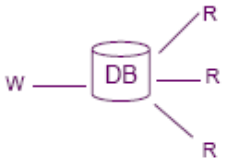
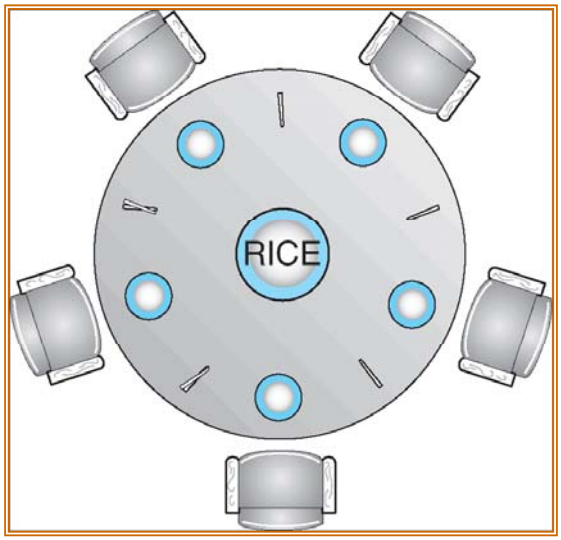


그림 6.13 Reader 프로세스의 구조

식사하는 철학자들 문제 (1) (Dining-Philosophers Problem)

- 배고픈 철학자가 가장 가까이 있는 두 개의 젓가락(왼쪽/오른쪽)를 집어야만 식사하는 예
- 공유 자료
  - 데이터 세트 (밥 그릇)
  - 세마포 chopstick[5] (젓가락)
    - 1로 초기화



식사하는 철학자들 문제 (2)

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        ...
    // eat
        ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        ...
    // think
        ...
} while (TRUE);
```

그림 6.15 철학자 i의 구조

- 문제는 교착 상태 (deadlock)
  - 5명의 철학자 모두가 동시에 자신의 왼쪽 젓가락을 잡는 경우
  - 모든 chopstick이 0이 되어 영원히 서로 기다림

세마포의 문제점

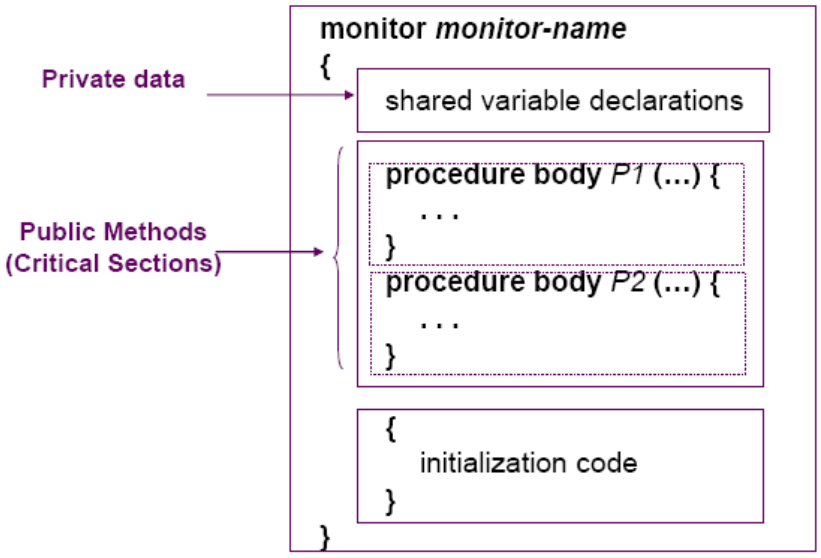
- 세마포를 이용하여 임계 구역 문제를 해결할 때 프로그래머가 세마포를 잘못 사용하면 다양한 유형의 오류가 너무나도 쉽게 발생
  - 상호 배제 요구 조건을 위반하든지 교착 상태가 발생
  - 아래의 경우
  - 프로세스에서 wait(mutex)나 signal(mutex) 또는 둘 다 생략된 경우

```
signal(mutex);
...
임계 구역
...
wait(mutex);
```

```
wait(mutex);
...
임계 구역
...
wait(mutex);
```


모니터 (Monitor)

- 모니터는 쉽고 효율적인 프로세스 동기화 수단을 제공하는 고급 언어수준의 동기화 구조물(high-level language synchronization construct)
  - 추상화된 데이터 형(abstract data type)을 안전하게 공유



모니터 구조

- 모니터 구조물은 모니터 안에 항상 하나의 프로세스만이 활성화되도록 보장

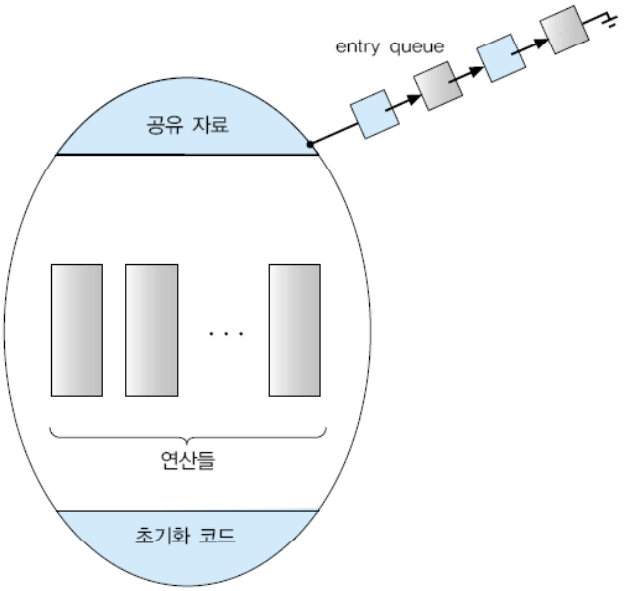


그림 6.17 모니터의 구조

condition 형의 변수

- 모니터 내부의 프로세스 대기를 위해 condition 변수 선언
  - condition x, y;
- condition 형 변수에 호출될 수 있는 연산은 오직 wait()와 signal()
  - x.wait(); 연산을 호출한 프로세스는 다른 프로세스가 x.signal();을 호출할 때까지 일시중단
  - x.signal() 연산은 정확히 하나의 일시 중단 프로세스를 재개. 만약 일시 중단된 프로세스가 없으면, signal() 연산은 아무런 효과가 없음

조건 변수를 갖는 모니터

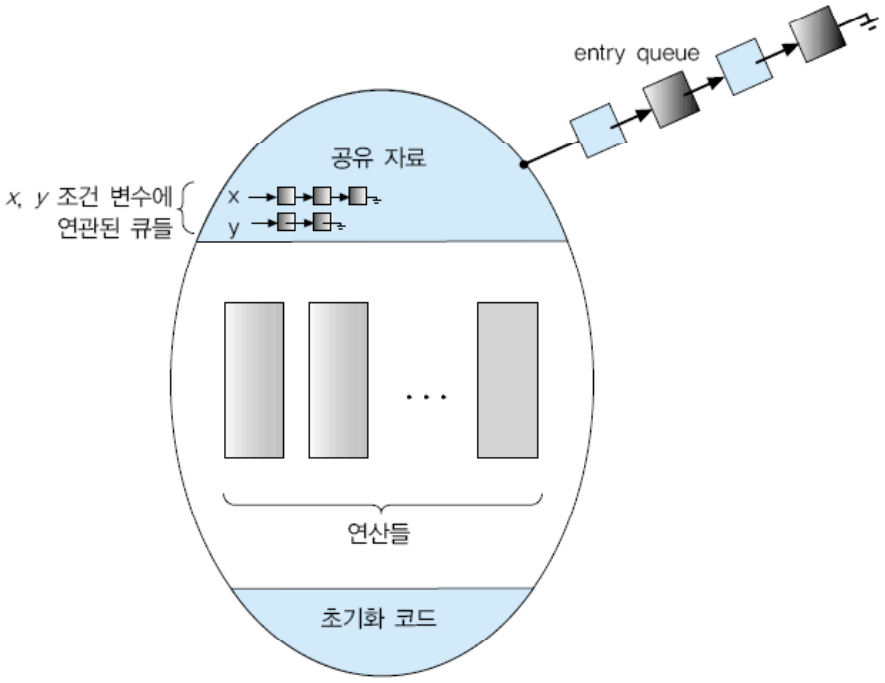


그림 6.18 조건 변수를 갖는 모니터

모니터를 사용한 식사하는 철학자 해결안

- 모니터를 사용하여 식사하는 철학자 문제에 대한 교착 상태가 없는 해결안을 제시
  - 철학자는 양쪽 젓가락 모두 얻을 수 있을 때만 젓가락을 집을 수 있다는 제한을 강제
  - 세가지 상태 자료구조
    enum { thinking, eating, hungry } state [5];
  - 철학자 i는 그의 양쪽 두 이웃이 식사하지 않을 때만 변수 state [i] = eating으로 설정
    - 왼쪽 이웃 조건 state [(i + 4) % 5]!= eating
    - 오른쪽 이웃 조건 state [(i + 1) % 5]!= eating
  - 조건 변수 선언
    condition self [5];

```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/
  void   pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating) /*two cases*/
              self[i].wait(); /*wait here*/
  }

  }

  void   test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
          state[i] = eating;
          self[i].signal();   / *=no_op. I'm
                                already running */
    }
  }
  void   init() {
    for (int i = 0; i < 5; i++)
          state[i] = thinking;
  }
```

Program using Monitor

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


```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/
  void   pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating) /*two cases*/
              self[i].wait(); /*wait here*/
  }
```


Case 1 – state[i] = eating
pick up & proceed


```
  }

  void   test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
          state[i] = eating;
          self[i].signal();   / *=no_op. I'm
                                already running */
    }
  }
  void   init() {
    for (int i = 0; i < 5; i++)
          state[i] = thinking;
  }
```

Program using Monitor

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

```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/

  void  pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating) /*two cases*/
              self[i].wait(); /*wait here*/
  }

  void  test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
          state[i] = eating;
          self[i].signal();   / *=no_op. I'm
                                already running */
    }
  }

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

Condition variable – queue process here

Case 2 – state[i] != eating
block myself
CPU → other process

Program using Monitor

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

```
monitor dining_philosopher
{
  enum {thinking, eating
        hungry (3rd state)} state[5];
  condition          self[5]; /*wait here*/

  void  pickup(int i) {
        state[i] = hungry; /* 3rd state */
        test(i); /* state of neighbors? */
        if (state[i] != eating)
              self[i].wait(); /*wait here*/
  }

  void  putdown(int i) {
        state[i] = thinking;
        // test left and right neighbors
        test((i+4) % 5); /*if L is waiting*/
        test((i+1) % 5);
  }
}
```

Can (Left eat now) && (Left was blocked)?

```
  void  test (int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)){ /OK/
          state[i] = eating;
          self[i].signal();   / *wakeup  Pi */
    }          /* used for putdown L & R */
  }

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

Program using Monitor

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

Linux의 동기화 (Synchronization in Linux)

- Linux 버전 2.6 부터 선점 가능 커널
  - 커널 모드에서 실행 중일 때에도 태스크는 선점될 수 있음
  - 이전 버전은 선점 불가능 커널
    - 커널 모드에서 실행중인 프로세스는 더 높은 우선순위의 프로세스가 실행 가능한 상태가 되더라도 선점될 수 없었음
- Linux 커널은 커널 안에서의 락킹(locking)
  - 스핀락(spinlock)
    - SMP 기계에서는 기본적인 락킹 기법
    - 단일 처리기에서는 스핀락을 사용하는 것은 부적합
      - 커널 선점 불능 및 가능으로 대체
    - 스핀락(커널 선점 불능 및 가능 또한)은 락(또는 커널 불능 기간)이 짧은 시간 동안만 유지될 때 사용
  - 세마포(semaphore)
    - 락이 오랜 시간 동안 유지되어야 한다면 세마포를 사용하는 것이 적절

Pthreads의 동기화 (Synchronization in Pthreads)

- Pthreads API는 운영체제에 독립적인 API
  - mutex 락 (mutex locks)
    - Pthread에서 사용할 수 있는 기본적인 동기화 기법으로 Mutex 락은 코드의 임계 구역을 보호하기 위해 사용
  - 세마포 (semaphore)
    - named와 unnamed라는 두 종류의 세마포 제공
  - 조건 변수 (condition variables)
  - read-write 락 (read-write locks)
  - 스핀락 (spin locks)

Pthread mutex 락

- pthread_mutex_t 데이터 형
- pthread_mutex_init(&mutex, NULL) 함수로 생성
  - 첫 번째 매개변수는 mutex를 가리키는 포인터
  - 두 번째 매개변수는 속성을 표시하며, NULL은 디폴트 속성
- mutex의 획득과 방출은 pthread_mutex_lock()과 pthread_mutex_unlock() 함수에 의해 수행
- 모든 mutex 관련 함수들은 성공적인 실행 시 0을 반환

```
#include <pthreads.h>
pthreads_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```


Pthread 세마포 (1)

- Unnamed 세마포 소개
- 세마포 생성

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);
```

- sem_init() 함수는 세마포를 생성하고 초기하고, 다음 세 개의 매개변수가 전달
  - 세마포를 가리키는 포인터
  - 공유 수준을 나타내는 플래그
    - 0 이면 세마포를 생성한 프로세스에 속한 쓰레드들만이 공유
  - 세마포의 초기값

Pthread 세마포 (2)

- Pthreads에서 wait()와 signal() 연산은 sem_wait()와 sem_post()

```
#include <smaphore.h>
sem_t sem mutex;

/* create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/*** critical section ***/

/*release the semaphore */
sem_post(&mutex);
```


원자적 트랜잭션 (Atomic Transaction)

- 임계 영역의 상호 배제는 임계 영역이 원자적으로 실행되는 것을 보장
  - 중단되지 않는 하나의 단위로 실행되는 것을 보장
- 원자적 트랜잭션 예
  - 은행의 자금 이체
    - 입금, 출금 트랜잭션
  - 데이터베이스 시스템
    - 데이터의 저장과 검색
    - 데이터 일관성 보장

시스템 모델

- 하나의 논리적인 기능을 실행하는 명령어(또는 연산)의 집합을 트랜잭션(transaction) 이라 함
  - 트랜잭션은 어떤 상황에서도 (컴퓨터 고장 시에도) 원자성(atomicity)을 보장하는 것이 중요
- 여기서는 트랜잭션을 디스크 내 파일 형태로 존재하는 데이터 항목을 접근하고 갱신하는 프로그램 단위로 간주
  - 트랜잭션은 일련의 읽기(read)와 쓰기(write)로 구성된 연산
  - 일련의 읽기/쓰기 연산은 완료(commit)되거나 철회(abort)로 끝남
    - 철회된 경우 어떤 문제로 실행이 중간에 멈춘 경우
  - 철회된 트랜잭션으로 접근된 데이터는 트랜잭션 시작 이전으로 복원해야 함
    - 이를 트랜잭션이 롤백(roll back)되었다고 함

기억장치 종류

- 휘발성 저장장치 (volatile storage)
  - 시스템이 고장나면 사라짐
  - 메인 메모리, 캐시
- 비휘발성 저장장치 (nonvolatile storage)
  - 시스템이 고장나도 보존됨
  - 디스크, 테이프
- 안전 저장장치 (stable storage)
  - 정보가 결코 손실되지 않음
  - 이론적으로 불가능하지만 이에 근접한 장치
  - 독립적인 장치에 중복해서 기록 (RAID)
- 시스템 고장으로 데이터 손실 가능성이 있는 비휘발성 저장장치 환경 하에서 트랜잭션의 원자성을 보장하는 것이 목표

로그 기반 복구 (Log-Based Recovery)

- 트랜잭션에 의해 접근된 모든 데이터의 변경 내역을 안전 저장 장치에 기록
- 가장 많이 사용되는 기법은 로그 우선 쓰기(write-ahead logging)
  - 시스템은 안전 저장장치에 로그(log)라는 자료구조를 유지
  - 각 로그 레코드는 트랜잭션의 하나의 쓰기 연산으로 아래 필드로 구성
    - 트랜잭션 이름
    - 데이터 항목 이름
    - 이전 값
    - 새 값
  - 트랜잭션 T_i 가 시작 전 <T_i starts> 레코드가 로그에 기록
  - 트랜잭션 T_i 가 쓰기를 할 때 마다 새로운 레코드가 로그에 기록
  - 트랜잭션 T_i 가 완료되면 <T_i commits> 레코드가 로그에 기록

로그-기반 복구 알고리즘

- 시스템은 로그를 사용하여 휘발성 저장장치의 어떠한 고장도 처리
  - undo(T_i) 는 T_i 가 갱신한 모든 데이터 항목의 값들을 이전 값으로 복구
  - redo(T_i) 는 T_i 가 갱신한 모든 데이터 항목에 새로운 값을 기록
- undo(T_i)와 redo(T_i)는 idempotent 해야 함
  - 이 연산을 여러 번 실행해도 한 번 실행 한 것과 동일한 결과를 만들어 내야 함
- 시스템 고장이 발생하면 로그를 참조하여 모든 갱신된 데이터를 복구
  - 로그에 <T_i starts> 가 있지만 <T_i commits>이 없으면 undo(T_i)
  - 로그에 <T_i starts>와 <T_i commits> 모두 있으면 redo(T_i)

검사점 (Checkpoints)

- 시스템 고장 시 로그 전체를 검색하면 작업시간이 많이 소모
- 검사점(checkpoint)는 로그의 길이를 줄이고 복구 시간을 단축

- 시스템은 주기적으로 다음의 검사점을 실행
  1. 현재까지 휘발성 저장장치에 있는 모든 로그 레코드들을 안전 저장장치에 출력
  2. 휘발성 저장장치에서 변경된 모든 데이터 항목들을 안전 저장장치로 출력
  3. <checkpoint>라는 로그 레코드를 안전 저장장치로 출력
- 시스템 고장 시 가장 최근의 검사점 직전에 실행된 T_i를 조사하고, T_i 이 후의 모든 트랜잭션들에 대해서만 복구를 수행

동시 실행 원자적 트랜잭션 (Concurrent Atomic Transactions)

- 다수의 트랜잭션들이 동시에 활성화되는 경우 고려

- 각 트랜잭션이 원자적이기 때문에 동시 실행 트랜잭션의 결과는 임의의 순서로 하나 씩 순차적으로 실행한 것과 같은 결과를 보여야 함
  - 이러한 성질을 직렬 가능성(serializability)라고 함

- 모든 트랜잭션들이 임계 영역 내에서 실행되면 가능
  - 원자성은 보장되지만 너무 비효율적

- 동시성-제어(concurrency-control) 알고리즘이 직렬 가능성을 보장

직렬 가능성 (Serializability)

- 두 개의 데이터 항목 A, B를 각각 읽기/쓰기를 하는 트랜잭션 T_0 와 T_1가 순서대로 원자적으로 실행한다고 가정
  - 이러한 실행 순서를 스케줄(schedule)이라고 함
  - 각 트랜잭션들이 원자적으로 실행되는 스케줄을 직렬 스케줄(serial schedule)이라고 함
  - N개의 트랜잭션이 있다면 N!의 유효한 직렬 스케줄이 존재

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

비직렬 스케줄 (nonserial schedule)

- 비직렬 스케줄(nonserial schedule)은 중첩 실행
  - 반드시 잘못된 결과를 초래하지는 않음
- 두 개의 트랜잭션 T_i 와 T_j에 속한 연산 O_i와 O_j를 차례로 실행하는 스케줄 S를 가정
  - O_i와 O_j가동일한 데이터 항목을 액세스하고 그 중 적어도 하나가 쓰기 연산이면 충돌(conflict) 발생
- O_i와 O_j가 서로 인접해 실행되는 연산이고 충돌하지 않으면 순서를 바꾼(swap) 스케줄 S' 가능
- 서로 순서를 바꾼 S와 S' 이 비충돌 연산인 경우 S는 충돌 직렬가능(conflict serializable) 하다고

스케줄2: 동시 실행 직렬가는 스케줄

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

락킹 프로토콜 (Locking Protocol)

- 직렬 가능성을 보장하는 한가지 방법으로 각 데이터 항목마다 락(lcok)을 설정
  - 락킹(locking) 프로토콜에 따라 락을 획득하고 반납
- 락킹 모드
  - 공유(shared)
    - 트랜잭션 T_i 가 데이터 항목 Q에 대한 공유 모드 락(S)을 가지고 있으면 T_i 는 Q를 읽을 수는 있지만 쓸 수는 없음
  - 독점(exclusive)
    - 트랜잭션 T_i 가 데이터 항목 Q에 대한 독점 모드 락(X)을 가지고 있으면 T_i 는 Q를 읽기/쓰기 모두 할 수 없음
- Q를 접근하는 모든 트랜잭션은 적절한 락을 획득해야 함
- 다른 트랜잭션이 락을 가지고 있으면 새로운 요청은 기다려야 함
  - 앞의 readers-writers 알고리즘과 유사

두 단계 락킹 프로토콜 (two-phase locking protocol)

- 충돌 직렬가능성을 보장
- 각 트랜잭션은 다음 두 단계로 락과 언락(unlock)을 요청
  - 확장 단계 (growing phase)
    - 트랜잭션은 락을 새로 획득할 수 있지만 반납은 안됨
  - 수축 단계 (shrinking phase)
    - 트랜잭션은 락을 반납할 수 있지만 새로운 락을 얻어서는 안됨
- 교착상태(deadlock) 문제로 부터 자유롭지 못함

타임스탬프 기반 프로토콜 (Timestamp-Based Protocols)

- 직렬 가능한 순서를 미리 선택
  - 타임스탬프 순서 기법(timestamp ordering scheme)이 널리 사용
- 트랜잭션 T_i마다 트랜잭션 실행 전에 고유한 고정 타임 스탬프 $TS(T_i)$ 를 부여
  - T_i가 T_j 전에 시스템에 들어오면 $TS(T_i) < TS(T_j)$
  - TS는 시스템 클럭이나 트랜잭션 진입 시 마다 증가되는 논리적인 카운터 등을 사용하여 생성
- 타임 스탬프는 직렬 가능성 순서를 결정
  - $TS(T_i) < TS(T_j)$ 이면 시스템은 생성된 스케줄이 T_i를 처리한 후 T_j 를 처리하는 직렬 스케줄과 동등하도록 보장해야 함

타임스탬프 기반 프로토콜 구현 (1)

- 각 데이터 항목 Q마다 두 개의 타임 스탬프 부여
  - W-timestamp(Q)
    - write(Q)를 성공적으로 실행한 트랜잭션의 타임스탬프 중 가장 큰 값
  - R-timestamp(Q)
    - read(Q) 성공적으로 실행한 트랜잭션의 타임스탬프 중 가장 큰 값
  - 타임스탬프들은 read(Q) 나 write(Q)가 실행될 때마다 갱신
- 타임스탬프 순서 프로토콜은 모든 충돌하는 읽기나 쓰기 연산들이 타임스탬프 순서대로 실행하는 것을 보장
- 트랜잭션 T_i가 read(Q)를 요청하면
  - $TS(T_i) < W\text{-timestamp}(Q)$이면 , T_i가 이미 겹쳐쓰기된 (overwritten) Q값을 읽기 시도
    - read 연산은 거부되고 T_i 는 롤백
  - $TS(T_i) \geq W\text{-timestamp}(Q)$ 이면,
    - read 연산은 실행되고 R-timestamp(Q) 는 R-timestamp(Q)와 $TS(T_i)$ 중 큰 값으로 지정

타임스탬프 기반 프로토콜 구현 (2)

- 트랜잭션 T_i가 write(Q)를 요청하면
  - $TS(T_i) < R\text{-timestamp}(Q)$이면 , T_i가 생성한 값이 과거에 필요했고 T_i가 이 값을 생성하지 않았을 것이라는 것을 의미
    - write 연산은 거부되고 T_i 는 롤백
  - $TS(T_i) < W\text{-timestamp}(Q)$ 이면, T_i는 이미 소용없게 된 값을 Q에 쓰려고 시도
    - write 연산은 거부되고 T_i 는 롤백
  - 이 외의 경우 write 연산이 실행
- 롤백된 트랜잭션 T_i는 새 타임스탬프가 배정되고 처음부터 다시 시작
- 이 알고리즘은 교착상태를 유발하지 않으면서 직렬가능성을 보장

스케줄3: 타임스탬프 프로토콜로 가능한 스케줄

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

실습과제

- 다음은 6.6.1절의 유한버퍼 문제를 Pthread로 구현한 p.305(p.266) 문제 6.37의 생산자-소비자 문제 프로젝트의 C 프로그램이다. 프로그램을 실행하고 교과서의 내용을 참조하여 프로그램 내용과 결과를 분석하라.
  - 다른 터미널를 생성하고 실행 중 다음을 실행하여 출력 분석
    $ ps –am –L

```
/* buffer.h*/

typedef int buffer_item;
#define BUFFER_SIZE 5

int insert_item(buffer_item item);
int remove_item(buffer_item *item);
```

```
/* buffer.c */

#include "buffer.h"
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define TRUE 1

buffer_item buffer[BUFFER_SIZE];
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

int insertPointer = 0, removePointer = 0;

void *producer(void *param);
void *consumer(void *param);
```



```
int insert_item(buffer_item item)
{
   //Acquire Empty Semaphore
     sem_wait(&empty);

     //Acquire mutex lock to protect buffer
     pthread_mutex_lock(&mutex);
     buffer[insertPointer++] = item;
     insertPointer = insertPointer % 5;

     //Release mutex lock and full semaphore
     pthread_mutex_unlock(&mutex);
     sem_post(&full);

     return 0;
}
```

```
int remove_item(buffer_item *item)
{
     //Acquire Full Semaphore
     sem_wait(&full);

     //Acquire mutex lock to protect buffer
     pthread_mutex_lock(&mutex);
     *item = buffer[removePointer];
     buffer[removePointer++] = -1;
     removePointer = removePointer % 5;

     //Release mutex lock and empty semaphore
     pthread_mutex_unlock(&mutex);
     sem_post(&empty);

     return 0;
}
```

```
int main(int argc, char *argv[])
{
    int sleepTime, producerThreads,
    consumerThreads;
    int i, j;

    if(argc != 4)
    {
        fprintf(stderr, "Useage: <sleep time>
    <producer threads> <consumer
    threads>₩n");
        return -1;
    }

    sleepTime = atoi(argv[1]);
    producerThreads = atoi(argv[2]);
    consumerThreads = atoi(argv[3]);

    //Initialize the the locks
    printf("%d₩n",pthread_mutex_init(&mutex,
    NULL));
    printf("%d₩n",sem_init(&empty, 0, 5));
    printf("%d₩n",sem_init(&full, 0, 0));
    srand(time(0));

    //Create the producer and consumer threads
    for(i = 0; i < producerThreads; i++)
    {
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, producer,
    NULL);
    }

    for(j = 0; j < consumerThreads; j++)
    {
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, consumer,
    NULL);
    }

    //Sleep for user specified time
    sleep(sleepTime);
    return 0;
}
```



```
void *producer(void *param)
{
    buffer_item random;
    int r;

    while(TRUE)
    {
        r = rand() % 5;
        sleep(r);
        random = rand();

        if(insert_item(random))
            fprintf(stderr, "Error");

        printf("Producer produced %d ₩n",
    random);

    }

}
```

```
void *consumer(void *param)
{
    buffer_item random;
    int r;

    while(TRUE)
    {
        r = rand() % 5;
        sleep(r);

        if(remove_item(&random))
            fprintf(stderr, "Error Consuming");
        else
            printf("Consumer consumed %d
    ₩n", random);
    }
}
```

특별 실습과제

- 참고 웹 사이트의 POSIX Thread Programming (https://computing.llnl.gov/tutorials/pthreads/)의 7장 Mutex Variables에서 소개된 예제를 분석하고 실행하여라