운영체제: 강의노트 08

A. Silberschatz, P.B. Galvin, G. Gagne
Operating System Concepts,
Sixth Edition, John Wiley & Sons, 2003.

Part II. Process Management

7 프로세스 동기화

7.5 고전 동기화 문제

7.5.1 한계 버퍼 생산자와 소비자 문제

- 생산자는 버퍼가 차 있으면 대기하여야 하고, 소비자는 버퍼가 비어 있으면 대기하여야 한다. 또한 버퍼에 대한 접근은 상호배제가 되어야 한다.
- 세마포어를 이용한 생산자 소비자 문제
- 생산자 프로세스: 그림 7.1

```
do {
        . . .
        nextp에 다음 item을 생산
        . . .
        wait(empty);
        wait(mutex);
        . . .
        nextp를 버퍼에 추가
        . . .
        signal(mutex);
        signal(full);
} while(1);
```

<그림 7.1> 생산자 프로세스

- 소비자 프로세스: 그림 7.2

```
do {
        wait(full);
        wait(mutex);
        . . .
        버퍼에 있는 item을 nextc로 옮김
        . . .
        signal(mutex);
        signal(empty);
        . . .
        nextc를 소비
        . . .
} while(1);
```

<그림 7.2> 소비자 프로세스

- 버퍼의 크기: n
- *mutex* 세마포어: 이진 세마포어로 1로 초기화되며, 버퍼 접근에 대한 상호배제를 제공한다.
- *empty* 세마포어: 계수 세마포어로 n으로 초기화되며, 비어 있는 버퍼의 수를 나타낸다.
- *full* 세마포어: 계수 세마포어로 0으로 초기화되며, 채워져 있는 버퍼의 수를 나타낸다.
- *mutex*는 상호배제를 위해 사용되었고, *empty*와 *full*은 동기화를 위해 사용되었다.

7.5.2 읽기 쓰기 문제

- 여러 프로세스가 데이터 객체(파일, 공유 메모리에 있는 구조)를 공유한다. 어떤 프로세스들은 이 객체의 내용을 읽고 싶고, 어떤 프로세스들은 이 객체의 내용을 갱신하고 싶다. 전자와 같은 프로세스를 읽기 프로세스(reader)라 하고, 후자와 같은 프로세스를 쓰기 프로세스(writer)라 한다.
- 두 읽기 프로세스가 병행으로 수행되어도 부정적인 결과가 발생하지 않는다. 그러나 하나의 쓰기 프로세스와 다른 프로세스가 병행으로 수행되면 부정적인 결과가 발생할 수 있다.

버전 1. 읽기 우선 읽기 쓰기 문제

- 이 버전에서 읽기 프로세스는 쓰기 프로세스가 공유 객체를 접근하고 있지 않으면 대기하지 않는다.
- 이 버전의 문제점: 쓰기 프로세스의 굶주림
- 쓰기 프로세스: 그림 7.3

```
wait(wsem);
. . .
쓰기 수행
. . .
signal(wsem);
```

<그림 7.3> 읽기 우선 쓰기 프로세스

- 읽기 프로세스: 그림 7.4

```
wait(mutex);
readcount++;
if (readcount == 1) wait(wsem);
signal(mutex);
. . .
읽기 수행
. . .
wait(mutex);
readcount−−;
if (readcount == 0) signal(wsem);
signal(mutex);
```

<그림 7.4> 읽기 우선 읽기 프로세스

```
wait(w);
writecount++;
if (writecount==1) wait(rsem);
signal(w);
wait(wsem);
...
쓰기 수행
...
signal(wsem); wait(w);
writecount--;
if (writecount==0) signal(rsem);
signal(w);
```

<그림 7.5> 쓰기 우선 읽기 프로세스

- *wsem* 세마포어: 이진 세마포어로 1로 초기화되며, 쓰기 접근에 대한 상호배제를 제공한다.
- *mutex* 세마포어: 이진 세마포어로 1로 초기화되며, *readcount* 변수 접근에 대한 상호배제를 제공한다.
- *readcount* 변수: 정수 변수로 0으로 초기화되며, 읽기 연산을 수행하고 있는 프로세스의 수를 나타낸다.
- 읽기 연산을 수행하고 있는 프로세스가 없을 때 읽기 연산을 수행하는 프로세스는 *wsem* 세마포어를 검사하여 쓰기 프로세스가 쓰기 연산을 수행하고 있는지 확인한다. 만약 없으면 쓰기 프로세스가 쓰기 연산을 하지 못하도록 한다.
- 읽기 연산을 맨 마지막에 수행하는 프로세스는 쓰기 프로세스가 쓰기 연산을 할 수 있도록 *wsem* 세마포어를 증가시켜준다.

버전 2. 쓰기 우선 읽기 쓰기 문제

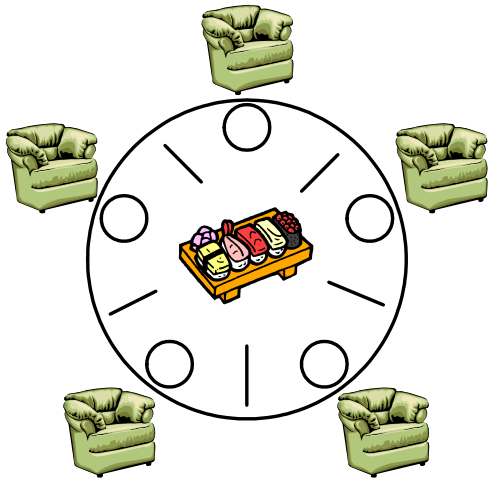
- 이 버전에서 쓰기 프로세스가 준비되었으면 더 이상 새로운 읽기 프로세스가 읽기 연산을 수행할 수 없다.
- 이 버전의 문제점: 읽기 프로세스의 굶주림
- 쓰기 프로세스: 그림 7.5
- 읽기 프로세스: 그림 7.6
- *wsem* 세마포어: 이진 세마포어로 1로 초기화되며, 쓰기 접근에 대한 상호배제를 제공한다.
- *rsem* 세마포어: 이진 세마포어로 1로 초기화되며, 쓰기 프로세스가 준비되었을 때 새로운 읽기 프로세스의 허용을 막기 위해 사용된다.
- *w* 세마포어: 이진 세마포어로 1로 초기화되며, *writecount* 변수에 대한 상호배제 접근을 제공한다.
- *r* 세마포어: 이진 세마포어로 1로 초기화되며, *readcount* 변수에 대한 상호배제 접근을 제공한다.

```
wait(rw);
wait(rsem);
wait(r);
readcount++;
if (readcount == 1) wait(wsem);
signal(r);
signal(rsem);
signal(rw);
...
읽기 수행
...
wait(r);
readcount--;
if (readcount==0) signal(wsem);
signal(r);
```

<그림 7.6> 쓰기 우선 읽기 프로세스

- *rw* 세마포어: 이진 세마포어로 1로 초기화되며, 모든 읽기 프로세스가 *rsem*에 대기하지 않도록 한다. 이것은 마지막 쓰기 프로세스가 완료되어 하나의 읽기 프로세스가 시작된 후에 다시 쓰기 프로세스가 준비되면 기존에 대기하고 있던 읽기 프로세스들보다 쓰기 프로세스에게 우선순위를 주기 위해 필요하다.
- *writecount* 변수: 정수 변수로 0으로 초기화되며, 쓰기 연산을 준비하고 있는 프로세스의 수를 나타낸다.
- *readcount* 변수: 정수 변수로 0으로 초기화되며, 읽기 연산을 수행하고 있는 프로세스의 수를 나타낸다.
- 프로세스 큐의 상태: 표 7.1

7.6 철학자들의 만찬 문제



<그림 7.7> 철학자들의 만찬 문제

- 1965년에 Dijkstra가 제안

<표 7.1> 쓰기 우선 읽기 쓰기 문제의 프로세스 큐들의 상태

읽기 프로세스만 존재	• *wsem*=0 (읽기 프로세스에 의해) • 모든 큐가 empty
쓰기 프로세스만 존재	• *wsem*=0 (쓰기 프로세스에 의해) • *rsem*=0 (쓰기 프로세스에 의해) • *wsem* 큐에 나머지 쓰기 프로세스들 대기
읽기와 쓰기 프로세스가 모두 존재 읽기 프로세스가 먼저 진입	• *wsem*=0 (읽기 프로세스에 의해) • *rsem*=0 (쓰기 프로세스에 의해) • *wsem* 큐에 모든 쓰기 프로세스들 대기 • *rsem* 큐에 한 읽기 프로세스가 대기 • *rw* 큐에 나머지 읽기 프로세스들 대기
읽기와 쓰기 프로세스가 모두 존재 쓰기 프로세스가 먼저 진입	• *wsem*=0 (쓰기 프로세스에 의해) • *rsem*=0 (쓰기 프로세스에 의해) • *wsem* 큐에 나머지 쓰기 프로세스들 대기 • *rsem* 큐에 한 읽기 프로세스가 대기 • *rw* 큐에 나머지 읽기 프로세스들 대기

- 그 이후 제안된 모든 동기화 도구는 이 문제를 얼마나 효율적으로 해결할 수 있는지를 보임으로써 그것의 우수성을 증명하였다.
- **철학자들의 만찬 문제(dining-philosopher problem)**: 다섯명의 철학자가 있다. 이들은 생각에 잠기거나 먹는 것 외에는 아무것도 하지 않는다. 철학자들은 원형 테이블에 앉아 있으며, 테이블 중앙에 밥이 있다. 그런데 젓가락이 다섯개 밖에 없다. 배가 고프면 철학자는 가장 가까이 있는 두 개의 젓가락을 집는다. 철학자는 한 번에 하나의 젓가락만 집을 수 있으며, 옆 사람이 집고 있는 젓가락은 집을 수 없으며, 두 개의 젓가락을 모두 집어야 식사를 할 수 있다. 식사를 한 후에는 다시 두 개의 젓가락을 모두 내려 놓고 생각에 잠긴다. 그림 7.7 참조.
- 단순 해결책: 그림 7.8

```
do {
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        . . .
        식사
        . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
} while(1);
```

<그림 7.8> 철학자들의 만찬 문제에 대한 단순 해결책

  - 공유 데이터

    semaphore *chopstick*[5];

    이 세마포어들은 모두 1로 초기화한다.
  - 문제점: 상호배제는 충족(이웃하는 철학자는 병행으로 식사할 수 없음)되지만 교착상태(모두 젓가락을 하나씩만 집을 수 있다)가 발생할 수 있다.
- 해결책 1. 하나의 젓가락을 집은 다음에 다른 하나를 집을 수 없으면 집은 젓가락을 내려놓고 잠시 기다린 후에 다시 시도한다.
  - 문제점: livelock이 발생할 수 있다.
  - 해결책: 임의의 시간을 기다린다. (Ethernet 방식)
- 해결책 2. 세마포어를 하나 더 사용하여 전체 과정을 임계 구역으로 설정한다. 그림 7.9

```
do {
        wait(mutex);
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        . . .
        식사
        . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        signal(mutex);
} while(1);
```

<그림 7.9> 철학자들의 만찬 문제에 대한 해결책 02

  - 문제점: 한번에 한 철학자만 식사가 가능
- 해결책 3. 세마포어를 하나 더 사용하여 젓가락을 집는 부분만 임계 구역으로 설정한다. 그림 7.10
  - 문제점: 교착상태가 발생할 수 있다.

7.7 임계 영역

- 세마포어를 사용하면 상호배제를 쉽게 충족시킬 수 있다.

```
do {
        wait(mutex);
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        signal(mutex);
        . . .
        식사
        . . .
        wait(mutex);
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        signal(mutex);
} while(1);
```

<그림 7.10> 철학자들의 만찬 문제에 대한 해결책 03

- 그러나 프로그램을 실수로 잘못 구성하거나 악의적인 프로그래머에 의해 악용될 수 있다.
- 세마포어를 이용한 일반적인 상호배제 구조는 다음과 같다.

  wait(*mutex*);

  . . .
  critical section
  . . .

  signal(*mutex*);

- **wait**와 **signal**의 순서를 바꾸면 상호배제가 충족되지 않는다.

  signal(*mutex*);

  . . .
  critical section
  . . .

  wait(*mutex*);

- 다음과 같은 구조를 사용하면 교착상태가 발생할 수 있다.

  wait(*mutex*);

  . . .
  critical section
  . . .

  wait(*mutex*);

- 이런 문제를 제거하기 위해 세마포어보다는 상위 개념의 동기화 해결 도구를 제공한다.
- 대표적인 상위 개념 동기화 도구
  - **임계 영역**(**critical region**)
  - **모니터**(**monitor**)
- 임계 영역 도구에서는 공유 변수는 다음과 같이 선언한다.

  v: **shared** T;

- 이 변수에 대한 접근은 **region** 문장을 내에서 수행해야 한다. **region** 문장의 형태는 다음과 같다.

  region v **when** (B) S;

  이 문장의 의미: B 조건이 참이어야 S 문장을 수행할 수 있으며, 이미 한 프로세스가 S를 수행하고 있으면 다른 프로세스는 기다린다.

- 다음 두 문장이 병행으로 수행되면

  region v **when** (true) S1;
  region v **when** (true) S2;

  그 결과는 S1을 수행하고 S2를 수행하거나 S2를 수행하고 S1을 수행한 결과와 같다. 즉, 원자적으로 수행된다.

- 임계 영역을 이용한 한계 버퍼 생산자 소비자 문제
  - 공유 데이터

    ```
    typedef {
            item pool[n];
            int count, in, out;
    } bufferT;
    buffer: shared bufferT;
    ```

  - 생산자 프로세스

    ```
    region buffer when (count<n) {
            pool[in]=nextp;
            in = (in+1) % n;
            counter++;
    }
    ```

  - 소비자 프로세스

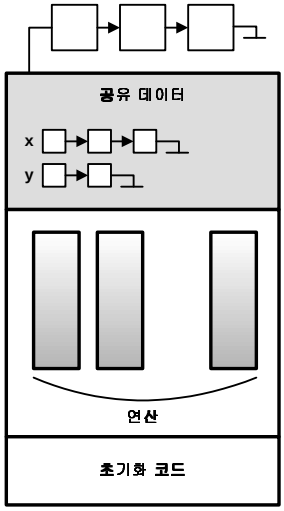
    ```
    region buffer when (count>0) {
            nextc=pool[out];
            out = (out+1) % n;
            counter--;
    }
    ```

7.8 모니터

- 모니터 내에 선언된 변수는 모니터 내에 정의된 함수를 통해서만 접근할 수 있으며, 모니터 내에 정의된 함수는 모니터 내에 선언된 변수만 사용할 수 있다.
- 한번에 한 프로세스만이 모니터 내부 함수를 실행할 수 있다.
- 이 기능만으로 복잡한 동기화 문제를 해결하기에 부족하다. 모니터 내에 정의된 함수를 사용할 수 없을 때 대기할 수 있도록 해주어야 한다. 따라서 모니터는 이를 위해 추가로 조건(condition) 타입의 변수를 제공한다.

  condition x, y;

- 조건 변수는 다음 두 연산에 의해서만 조작될 수 있다.



<그림 7.11> 조건 변수를 가진 모니터

- **wait** 연산

  x.**wait**();

  이 연산을 수행하는 프로세스는 대기하게 된다.

- **signal** 연산

  x.**signal**();

  이 연산을 수행하면 대기 중인 하나의 프로세스가 재개한다.

- 모니터를 이용한 교착상태가 없는 철학자들의 만찬 문제에 대한 해결책: 그림 7.12
  - 각 철학자는 다음 절차에 따라 식사를 한다.

    dp.pickup(i);
    . . .
    식사
    . . .
    dp.putdown(i);
  - 조건변수 *self*를 이용하여 자신이 배고프지만 젓가락을 집을 수 없으면 스스로 기다린다.
- 세마포어를 이용한 모니터의 구현
  - 각 모니터마다 1로 초기화된 *mutex* 세마포어를 연관한다. 모든 프로세스는 모니터를 사용하기 전에 **wait**(*mutex*)를 사용해야 하며, 모니터의 사용을 끝내기 전에 **signal**(*mutex*)를 사용해야 한다.
  - 조건 변수에 대한 **signal**을 보낸 프로세스는 재개된 프로세스가 모니터 사용을 완료하던가 다른 세마포어에 의해 대기할 때까지 대기해야 한다. 만약 이렇게 하지 않으면 두 프로세스가 모니터 내에 병행으로 존재할 수 있다. 이를 위해 0으로 초기화된 세마포어 *next*를 사용하며, *next*에 대기하는 프로세스의 수를 유지하기 위해 0으로 초기화된 *next_count* 변수를 사용한다.

```
monitor dp {
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void picking(int i) {
        state[i] = hungry;
        test(i);
        if(state[i]!=eating) self[i].wait();
    }
    void putdown(int i) {
        state[i] = thinking;
        test((i+4)%5);
        test((i+1)%5);
    }
    void test(int i) {
        if ((state[(i+4)%5] != eating) &&
            (state[i] == hungry) &&
            (state[(i+1) % 5] != eating)){
                state[i] = eating;
                self[i].signal();
        }
    }
    void init() {
        for(int i=0; i<5; i++) state[i]=thinking;
    }
}
```

<그림 7.12> 모니터를 이용한 철학자들의 만찬 문제에 대한 해결책

  - 모니터 내부에 선언된 각 함수는 세마포어로 구현하면 실제 다음과 같다.

    wait(*mutex*);
    . . .
    함수의 실제 본체
    . . .
    if (*next_count*>0) **signal**(*next*);
    else **signal**(*mutex*);
  - 각 조건 변수 x마다 이진 세마포어 *x_sem*과 정수 변수 *x_count*를 연관한다. 이 둘은 모두 0으로 초기화된다.
  - x.**wait**

    x_count++;
    if (*next_count*>0) **signal**(*next*);
    else **signal**(*mutex*);
    wait(*x_sem*);
    x_count--;
  - x.**signal**

    if (x_count>0) {
        next_count++;
        signal(*x_sem*);
        wait(*next*);
        next_count--;
    }
- 프로세스가 재개되는 순서
  - 가장 단순한 방법: FCFS

<표 7.2> 모니터를 이용한 철학자들의 만찬 문제의 해결책

	state[0]	state[1]	state[4]	sem[0]	cnt[0]	sem[1]	cnt[1]	sem[4]	cnt[4]	mutex	next	n_cnt
초기	thinking	thinking	thinking	0	0	0	0	0	0	1	0	0
P_0 pickup 시작	hungry	thinking	thinking	0	0	0	0	0	0	0	0	0
P_0 pickup 완료	eating	thinking	thinking	0	0	0	0	0	0	1	0	0
P_1 pickup 시작	eating	hungry	thinking	0	0	0	0	0	0	0	0	0
P_1 pickup 대기	eating	hungry	thinking	0	0	대기	1	0	0	1	0	0
P_4 pickup 시작	eating	hungry	hungry	0	0	대기	1	0	0	0	0	0
P_4 pickup 대기	eating	hungry	hungry	0	0	대기	1	대기	1	1	0	0
P_0 putdown 시작	thinking	hungry	hungry	0	0	대기	1	대기	1	0	0	0
P_0 putdown 대기	thinking	hungry	eating	0	0	대기	1	신호	1	1	대기	1
P_4 pickup 완료	thinking	hungry	eating	0	0	대기	1	0	0	0	신호	0
P_0 putdown 재개	thinking	hungry	eating	0	0	신호	1	0	0	0	대기	1
P_0 putdown 대기	thinking	eating	eating	0	0	신호	1	0	0	0	대기	1
P_1 pickup 완료	thinking	eating	eating	0	0	0	0	0	0	0	신호	0
P_0 putdown 완료	thinking	eating	eating	0	0	0	0	0	0	1	0	0

P_i: 철학자 i, sem[i]: self[i]_sem, cnt[i]: self[i]_count, n_cnt: next_count

- 우선순위 방법: 조건부 **wait** 사용

  x.**wait**(c);

  여기서 c는 우선순위 값이다.

```
monitor R {
    boolean busy;
    condition x;
    void acquire(int time) {
        if(busy) x.wait(time);
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
    void init() {
        busy = false;
    }
}
```

<그림 7.13> 모니터를 이용한 단일 자원 할당 문제에 대한 해결책

- 조건부 **wait**를 사용하는 예: 그림 7.13 참조.
  - 각 프로세스는 다음 절차에 따라 자원을 할당 받는다.

    R.acquire(t);
    ...
    자원 사용
    ...
    R.release();

    여기서 t는 자원을 사용할 최대 시간이다. 여러 프로세스가 대기 중일 때 **signal**이 발생하면 가장 작은 시간의 프로세스를 먼저 재개된다.
  - 문제점
    - 프로세스는 허가를 받지 않고 자원을 접근할 수 있다.
    - 프로세스는 자원을 할당 받은 다음에 해제하지 않을 수 있다.
    - 프로세스는 할당 받지도 않은 자원에 대한 해제를 시도할 수 있다.
    - 프로세스는 할당 받은 자원을 해제하지 않고 다시 같은 자원의 할당을 요청할 수 있다.
  - 해결책
    - 방법 1. 자원 사용 코드를 모니터 내부에 포함한다. 모니터 내에서 외부 데이터를 사용하며, 스케줄링이 모니터에 의해 이루어지므로 적절한 방법이 아님.
    - 방법 2. 모든 코드를 검사한다.