운영체제: 강의노트 07

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

Part II. Process Management

7 프로세스 동기화

7.1 배경

- 예1) 생산자 소비자 문제에서 생산자와 소비자 프로세스는 공유메모리를 이용하여 공통된 변수와 버퍼를 공유한다. 여기서는 카운터를 이용하여 버퍼의 모든 공간을 사용하는 해결책을 생각하여 보자.
  - 이들 프로세스는 독립적으로 수행되면 올바르게 동작하나 공유 변수의 값이 결과에 영향을 주므로 병행으로 수행되면 실행되는 순서에 따라 올바르게 동작하지 않을 수 있다.
  - 고급 프로그래밍 언어의 한 문장은 컴파일러에 의해 여러 개의 기계 명령어로 번역될 수 있다. 인터럽트는 한 기계 명령어의 명령어 주기가 끝난 시점에서 처리된다.
  - 공유메모리에 있는 변수 *counter*의 값이 5일 때, 생산자와 소비자가 각각 "counter++;"와 "counter--;"를 병행으로 수행하면 결과는 5가 되어야 하지만 그렇지 않을 수 있다.
    - "counter++"를 하위 기계어로 표현하면 다음과 같으며,

      $$\begin{aligned} &\textit{register}_1 = \text{counter};\\ &\textit{register}_1 = \textit{register}_1 + 1;\\ &\text{counter} = \textit{register}_1; \end{aligned}$$

      "counter--"는 다음과 같다.

      $$\begin{aligned} &\textit{register}_2 = \text{counter};\\ &\textit{register}_2 = \textit{register}_2 - 1;\\ &\text{counter} = \textit{register}_2; \end{aligned}$$

      여기서 $\textit{register}_1$과 $\textit{register}_2$는 CPU 내부 레지스터이며, 같은 레지스터일 수 있다.
  - 두 문장이 병행으로 수행되면 그림 7.1과 같이 수행될 수 있다.

- 이 처럼 병행으로 수행되는 여러 프로세스가 공통된 데이터를 조작할 때 결과가 접근 순서에 의해 결정되면 **경합 상태(race condition)**가 존재한다고 한다.

T_0 : 생산자	$\textit{register}_1 = \text{counter};$	$\{\textit{register}_1 = 5\}$
T_1 : 생산자	$\textit{register}_1 = \textit{register}_1 + 1;$	$\{\textit{register}_1 = 6\}$
T_2 : 소비자	$\textit{register}_2 = \text{counter};$	$\{\textit{register}_2 = 5\}$
T_3 : 소비자	$\textit{register}_2 = \textit{register}_2 - 1;$	$\{\textit{register}_2 = 4\}$
T_4 : 생산자	$\text{counter} = \textit{register}_1;$	$\{\text{counter} = 6\}$
T_5 : 소비자	$\text{counter} = \textit{register}_2;$	$\{\text{counter} = 4\}$

<그림 7.1> "counter++"와 "counter--"의 병행 수행 결과

- 예2) 변수의 값이 항상 같아야 하는 두 개의 변수 a와 b를 여러 프로세스가 공유한다고 가정하자. 이 때 다음과 같은 두 개의 프로세스가 있다고 하자.
  - P_1

    a = a + 1;
    b = b + 1;
  - P_2

    b = b * 2;
    a = a * 2;

  각 프로세스가 독립적으로 수행되면 일관성은 유지된다. 그러나 병행으로 수행되면 다음과 같이 일관성이 깨질 수 있다.

  a = a + 1;
  b = b * 2;
  b = b + 1;
  a = a * 2;

  예 1과 달리 한 고급 프로그래밍 문장이 원자적으로 수행되어도 원하지 않는 결과가 발생할 수 있다.

- 여러 프로세스가 병행으로 수행되어 발생하는 문제를 해결하는 것을 프로세스 동기화라 한다.

7.2 임계 구역 문제

- **임계 구역(critical section)**: 프로세스의 코드의 일부분으로서, 다른 프로세스와 공동으로 사용하는 변수, 테이블, 파일 등을 변경하는 부분이다.

- 임계 구역의 실행은 상호배타적(mutually exclusive)으로 실행되어야 한다. 즉, 한 프로세스가 임계 구역을 실행하고 있으면 다른 프로세스는 임계 구역에 진입할 수 없어야 한다.

- 따라서 각 프로세스는 임계구역에 진입하기 전에 허가를 받아야 한다. 이 허가를 요청하는 코드를 **진입 구역(entry section)**이라 한다. 허가를 받아 임계구역을 실행한 다음에는 다른 프로세스들이 진입할 수 있도록 해주어야 한다. 이것을 하는 코드를 **출구 구역(exit section)**이라 한다.

- 진입 구역, 임계 구역, 출구 구역이 아닌 코드 부분을 **잔류 구역(remainder section)**이라 한다.

- 임계 구역이 포함된 프로세스의 일반 구조

  do {

entry section

  critical section

exit section

  remainder section

  } while(1);

  여기서 do while 구조로 표현한 것은 특별한 이유가 있는 것은 아니다.

- 임계 구역 문제를 해결하는 메커니즘은 다음 세 가지 요건을 충족해야 한다.
  - **상호 배제(mutual exclusive)**: 한 프로세스가 임계 구역에서 실행하고 있으면 어떤 프로세스도 임계 구역에 진입할 수 없어야 한다.
  - **진행(progress)**: 임계 구역을 실행하고 있는 프로세스가 없을 때, 몇 개의 프로세스가 임계 구역에 진입하고자 하면 이들의 진입 순서는 이들에 의해서만 결정되어야 한다. 또한 이 선택은 무한정 연기되어서는 안 된다.
  - **한계 대기(bounded waiting)**: 한 프로세스가 자신의 임계 구역에 진입하고자 요청을 한 후부터 이 요청이 허용될 때까지 다른 프로세스가 그들의 임계 구역에 진입할 수 있는 회수가 제한되어야 한다.

7.3 소프트웨어 접근 방법

7.3.1 두 개의 프로세스를 위한 해결책

- 두 개의 프로세스 중 하나를 P_0라 하고 다른 하나는 P_1이라 하자. 또한 설명의 편리성을 위해 하나를 P_i라 하면 다른 하나는 P_j가 된다.
- **알고리즘 1**. Dekker의 알고리즘
  - 공유 변수: int *turn*;
  - 초기값: *turn*=1; /* or 0 */
  - *turn*이 i이면 P_i가 임계 지역에 진입할 수 있고, j이면 P_j가 진입할 수 있다.

  프로세스 P_i의 구조는 다음과 같다.

  do {

while(*turn* != i);

  critical section

turn=j;

  remainder section

  } while(1);

  - 상호 배제는 만족하지만 진행 요구 조건은 충족하지 못한다. 즉, 엄격하게 교대로 진입하므로 다른 프로세스가 진입하고자 하더라도 그 프로세스의 차례가 아니면 무조건 기다려야 한다. 특히 한 프로세스가 예기치 않게 종료되면 종료되는 위치와 상관없이 다른 프로세스는 영원히 진입할 수 없다.

- **알고리즘 2**. Dekker의 알고리즘
  - 공유 변수: boolean *flag*[2];
  - 초기값: *flag*[0]=*flag*[1]=false;
  - *flag*[i]가 true이면 P_i가 진입할 준비가 되었음을 나타낸다.

  프로세스 P_i의 구조는 다음과 같다.

  do {

flag[i]=true; while(*flag*[j]);

  critical section

flag[i]=false;

  remainder section

  } while(1);

  - 상호 배제는 만족하지만 진행 요구 조건은 충족하지 못한다. 특히 다음과 같이 실행되었다면 P_0와 P_1은 모두 영원히 기다리게 된다.

    T_0 : P_0 *flag*[0] = true;
    T_1 : P_1 *flag*[1] = true;

  - 프로세스가 임계 구역 내에서 또는 자신의 *flag* 값을 true로 설정한 다음에 예기치 않게 종료되면 다른 프로세스는 영원히 진입할 수 없다.
  - 이 알고리즘의 진입 구역을 다음과 같이 바꾸면 어떻게 될까?

    while(*flag*[j]);
    flag[i]=true;

    이 경우에는 상호 배제가 만족되지 않는다.
  - 이 알고리즘의 진입 구역을 다음과 같이 바꾸면 어떻게 될까?

    flag[i]=true;
    while(*flag*[j]){
    flag[i]=false;
    /* delay */;
    flag[i]=true;
    }

    이 경우에는 다음과 같이 수행될 수 있다.

    T_0 : P_0 *flag*[0]=true;
    T_1 : P_1 *flag*[1]=true;
    T_2 : P_0 checks *flag*[1]
    T_3 : P_1 checks *flag*[0]
    T_4 : P_0 *flag*[0]=false;
    T_5 : P_1 *flag*[1]=false;
    T_6 : P_0 *flag*[0]=true;
    T_7 : P_1 *flag*[1]=true;

    이것이 영구적으로 반복될 수 있다. 그러나 두 프로세스의 실행 속도가 조금만 어긋나도 정상적으로 수행될 수 있기 때문에 교착상태는 아니다. 이런 현상을 **livelock**이라 한다.

- **알고리즘 3**. Dekker의 알고리즘
  - 공유 변수

    boolean *flag*[2];
    int *turn*;

  - 초기값

    flag[0]=*flag*[1]=false;
    turn=0; /* or 1 */

프로세스 P_i의 구조는 다음과 같다.

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

둘 중 하나만 진입하고자 하면 다른 프로세스의 *flag* 값이 false이므로 진입할 수 있다. 두 프로세스가 모두 진입하고자 하면 *turn* 값이 진입하는 프로세스를 결정된다. 이 알고리즘은 세 가지 요구 조건을 모두 충족한다.

- **알고리즘 4**. Peterson의 알고리즘
  - 공유 변수

    boolean *flag*[2];
    int *turn*;

  - 초기값

    flag[0]=*flag*[1]=false;
    turn=0; /* or 1 */

프로세스 P_i의 구조는 다음과 같다.

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

임계 구역에 진입하기 전에 P_i는 *flag*[i]를 true로 설정하여 자신이 진입하고자 함을 나타내고, 그 다음 *turn*의 값을 j로 설정하여 다른 프로세스가 진입하고자 하면 이를 허용한다. P_i는 *flag*[j]가 false이거나 *turn*이 i일 경우에만 진입할 수 있다.

  - 상호 배제: 두 프로세스가 모두 진입하고자 하더라도 진입 구역의 두 번째 문장의 실행 순서에 따라 *turn*의 값은 i 또는 j 중 하나의 값을 가지게 되므로 두 프로세스 중 하나만 while 문을 통과할 수 있다. 또한 다른 프로세스는 임계 구역에 진입한 프로세스가 자신의 *flag* 값을 false로 설정하기 전까지 임계 구역에 진입할 수 없다.
  - 진행과 한계 대기: P_i가 진입을 시도하지 않으면 *flag*[i] 값이 false이므로 P_j는 임계 구역에 진입할 수 있다. 둘 다 동시에 진입하고자 하면 상호 배제가 충족되므로 어느 하나만 진입할 수 있으며 진입된 프로세서가 임계구역을 빠져나오면 *flag* 값을 false로 바꾸므로 다른 프로세서는 진입할 수 있다. 따라서 최대 다른 프로세스가 한번 진입한 후에는 진입할 수 있다.

7.3.2 다중 프로세스를 위한 해결책

- 이 알고리즘은 다른 말로 제과점 알고리즘(bakery algorithm)이라 한다.
- 기본 생각: 가게에 들어오는 손님들은 번호를 하나 받으며, 가장 낮은 번호를 받은 손님 순으로 서비스를 받는다. 그런데 모든 손님이 다른 번호를 받도록 보장할 수 없다. 만약 두 손님이 같은 번호를 받으면 이름 순으로 서비스를 받는다.
  - 모두 다른 번호를 받도록 하기 위해서는 번호를 받는 것 자체가 임계 구역이 된다.
  - 컴퓨터에서 프로세스는 모두 다른 이름(프로세스 번호)을 가진다.
- 공유 변수

  boolean *choosing*[n];
  int *num*[n];

  *choosing*은 모두 false로 초기화되며, *num*은 모두 0으로 초기화된다.
- 표기법
  - $(a, b) < (c, d)$: $a < c$이거나 $a == c$이고 $b < d$이면 참이다.
  - max(a_0,a_1,. . .,a_{n-1}): $a_i (i = 0, \ldots, n-1)$ 중 가장 큰 값
- 프로세스 P_i의 구조는 다음과 같다.

```
do {
    choosing[i]=true;
    num[i]=max(num[0], num[1],
            . . ., num[n-1])+1;
    choosing[i]=false;
    for(j=0; j<n; j++){
        while(choosing[j]);
        while((num[j]!=0)&&
            ((num[j],j)<(num[i],i)));
    }
```

critical section

num[i]=0;

remainder section

} while(1);

- 알고리즘의 정확성
  - 상호배제
    - P_i가 번호를 선택한 후에 P_k가 선택하였다면 (*num*[i],i)<(*num*[k],k)가 성립한다. 따라서 P_k는 P_i가 임계 구역에 빠져나오면서 num[i]=0을 할 때까지 진입할 수 없다.
    - 비슷한 시기에 번호를 받아 같은 번호를 받더라도 두 프로세스의 이름이 같을 수 없으므로 이 중 하나가 먼저 임계 구역에 진입하며, 다른 하나는 그 프로세스가 임계 구역에 빠져나올 때까지는 진입할 수 없다.
    - *choosing* 변수가 필요한 이유: 같은 번호를 받은 두 프로세스가 둘 다 임계 구역에 진입하는 경우를 제거하기 위해 필요하다. 예를 들어 같은 번호를 받게 되는 두 프로세스 P_i와 P_k 중 P_i는 번호를 받는 도중에 프로세스 스위치가 일어났고, 다른 프로세스는 계속 진행을 하여 for 문을 수행하고 있다. 이 경우 while(*choosing*[j]); 문이 없으면 P_k는 임계 구역에 아무런 제재 없이 진입하게 된다. P_i가 다시 실행되면 P_k와 번호가 같고 이름이 더 작으므로 이 프로세스도 진입하게 된다.
  - 진행: 번호를 선택하고 있지 않는 프로세스는 *num*[i]=0이므로 다른 프로세스의 진입을 막지 못한다.
  - 한계 대기: 진입하고자는 두 프로세스 P_i와 P_k 중 P_k가 먼저 번호를 받아 진입을 하였어도 P_k가 기다라고 있는 P_i보다 먼저 다시 진입을 할 수 없다. 즉, P_i는 최대 각 프로세스가 각 한 번씩 진입한 후에는 진입할 수 있다.

7.4 동기화 하드웨어

- 단일 프로세서 시스템의 경우 공유된 변수를 변경하는 동안에 인터럽트를 발생할 수 없도록 하면 위와 같은 문제를 보다 쉽게 해결할 수 있다.
- 인터럽트 억제 방법을 이용한 임계 구역 문제의 해결책

do {

disable interrupt

critical section

enable interrupt

remainder section

} while(1);

다중 프로그래밍에 큰 영향을 주므로 효율적인 해결책은 아니다.

- 다중 프로세서 시스템에서는 인터럽트가 발생할 수 없도록 하더라도 여전히 두 프로세서에서 두 프로세스가 동시에 실행될 수 있어 임계 구역 문제를 해결할 수 없다.
- 이 때문에 인터럽트를 억제하는 방법 대신 대부분의 시스템은 이런 문제를 해결할 때 사용할 수 있는 특수한 하드웨어 명령어를 제공한다. 이 명령어는 원자적으로(이 명령어를 수행하는 동안에는 인터럽트되지 않음) 한 워드를 검사하고 수정할 수 있도록 해주거나 두 워드의 값을 교환할 수 있도록 해준다.
- **TestAndSet** 명령어

```
boolean TestAndSet(boolean &target){
        boolean rv = target;
        target = true;
        return rv;
}
```

이 명령어는 원자적으로 실행되므로 두 프로세스가 이 명령어를 병행으로 수행하더라도(두 프로세서에서 동시에 수행되더라도) 임의 순서로 순차적으로 수행된다. 시스템이 **TestAndSet** 명령을 제공하면 *lock*이라는 불(boolean)형 변수를 선언하고, 이것을 false로 초기화하여 상호 배제를 다음과 같이 쉽게 구현할 수 있다.

do {

while(**TestAndSet**(*lock*));

critical section

lock=false;

remainder section

} while(1);

- **Swap** 명령어

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

시스템이 **Swap** 명령을 제공하면 *lock*이라는 광역 불형 변수(공유 변수)와 프로세스마다 *key*라는 지역 변수를 선언하여 상호 배제를 다음과 같이 구현할 수 있다. 이 때 *lock*은 false로 초기화한다.

do {

key=true;
while(*key*==true) **Swap**(*lock*,*key*);

critical section

lock=false;

remainder section

} while(1);

그러나 **TestAndSet**과 **Swap**을 이용하는 두 알고리즘은 모두 상호배제만 충족할 뿐 한계 대기는

충족되지 않는다. 이 두 명령이 실행되는 순서에 따라 한 프로세스는 영구적으로 임계 구역에 진입을 못할 수 있다.

- **TestAndSet**을 이용한 임계 구역 문제의 모든 요구사항을 충족하는 알고리즘

```
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(1);
```

  - 상호배제: *lock*은 false로 초기화된다. 따라서 **TestAndSet**을 제일 먼저 실행한 프로세스는 자신의 지역 변수 *key* 값이 false가 되어 진입한다. 다른 프로세스들은 진입한 프로세스가 임계 구역을 끝내고 출구 지역에서 **lock**을 false로 하거나 *waiting*[j]를 false로 변경해 줄 때까지 진입할 수 없으므로 상호배제가 충족된다.
  - 진행: 임계 구역을 끝낸 프로세스는 출구 지역에서 다른 프로세스가 진입할 수 있도록 *lock*을 false로 하거나 *waiting*[j]를 false로 변경해주므로 다른 프로세스는 결국에는 진입할 수 있다.
  - 한계 대기: 임계 구역을 끝낸 프로세스는 출구 지역에서 $i+1$부터 $i-1$까지 순환 순서로 다른 프로세스의 *waiting*을 조사한다. 이 때 첫번째로 *waiting* 값이 true인 것을 false로 변경해준다. 또한 기다리고 있는 프로세스가 없으면 *lock*을 false로 변경해준다. 따라서 최대 다른 프로세스가 한 번씩 진입한 후에는 임계구역에 진입할 수 있다.

- 동기화 하드웨어나 소프트웨어 접근 방식의 공통된 문제점: 모두 어느 한 프로세스가 임계 구역에 있으면 진입하고자 하는 다른 프로세스는 "busy waiting"을 해야 한다.

7.5 세마포어

- **세마포어(semaphore)** S는 정수 변수로서 초기화를 제외하고는 두 가지 연산 **wait**와 **signal**을 통해서만 접근할 수 있다. 이 두 연산은 원자적이다.

- **wait** 연산

```
wait(S){
    while(S <= 0);
    S--;
}
```

- **signal** 연산

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

7.5.1 사용법

- 세마포어를 이용한 n 프로세스 임계 구역 문제에 대한 해결책

```
do {
        wait(mutex);

        critical section

        signal(mutex)

        remainder section
} while(1);
```

  모든 프로세스는 *mutex*라는 세마포어를 공유하며, *mutex*는 1로 초기화된다.

  - **wait** 연산을 제일 먼저 실행하는 프로세스는 *mutex* 값이 1이므로 이 값을 0으로 바꾸고 임계 구역에 진입할 수 있다. 이 프로세스가 출구 지역에서 *mutex* 값을 1로 다시 바꾸어주면 다른 프로세스가 진입을 할 수 있다.
  - 이 방법도 앞서 본 **TestAndSet**과 **Swap** 명령어를 이용한 방법과 마찬가지로 한계 대기는 충족하지 못한다.

- 병행으로 수행되는 두 개의 프로세스 P_1과 P_2는 각각 S_1과 S_2 프로그래밍 문장을 가지고 있다고 하자. 또한 S_1이 실행된 다음에 S_2가 실행되어야 한다. 이런 동기화 문제는 세마포어를 이용하여 쉽게 해결할 수 있다. 두 프로세스는 0으로 초기화된 *synch*라는 세마포어를 공유한다.

  - P_1

    S_1;
    signal(*synch*);

  - P_2

    wait(*synch*);
    S_2;

7.5.2 구현

- busy waiting을 하는 세마포어를 **spinlock**이라 한다. spinlock은 문맥 전환이 필요없어 오래 동안 busy waiting을 하지 않는다면 효과적인 방법이다.

- busy waiting을 제거하기 위해서는 **wait** 연산을 수행하였을 때 세마포어 값이 양수가 아니면 스스로 블록하도록 해야 한다.

- busy waiting이 없는 세마포어 구조

```
typedef struct {
    int value;
    struct process *L';
} semaphore;
```

- busy waiting이 없는 **wait** 연산

```
void wait(semaphore S){
    S.value--;
    if(S.value < 0){
        add this process to S.L;
        block();
    }
}
```

여기서 큐는 FIFO 방식이다.

- busy waiting이 없는 **signal** 연산

```
void signal(semaphore S){
    S.value++;
    if(S.value <= 0){
        remove a process P from S.L;
        wakeup(S.L);
    }
}
```

- busy waiting이 있는 세마포어의 경우에는 세마포어 값이 음수가 되지 않는다. 그러나 busy waiting이 없는 세마포어의 경우에는 세마포어 값이 음수가 될 수 있으며, 음수이면 이것은 세마포어를 기다리는 프로세스의 수를 나타낸다.

- 세마포어가 올바르게 동작하기 위해서는 **wait**와 **signal** 연산은 반드시 원자적으로 수행되어야 한다. 이를 위해 단일 프로세서 시스템에서는 인터럽트 억제 방법을 사용할 수 있다. 그러나 다중 프로세서 시스템에서는 이 방법이 가능하지 않다. 따라서 특수한 하드웨어 명령을 제공하지 않으면 7.2절에 제시된 소프트웨어 해결법을 사용해야 한다.

7.5.3 이진 세마포어

- 지금까지 사용한 세마포어는 가질 수 있는 값의 범위가 정해져 있지 않은 카운팅 세마포어이다.

- 세마포어가 가질 수 있는 값의 범위가 0 또는 1로 제한되어 있으면 **이진 세마포어**라 한다.

- busy waiting이 없는 이진 **wait** 연산

```
void wait(semaphore S){
    if(S.value == 1)
        S.value = 0;
    else {
        add this process to S.L;
        block();
    }
}
```

- busy waiting이 없는 이진 **signal** 연산

```
void signal(semaphore S){
    if(S.L is empty)
        S.value = 1;
    else {
        remove a process P from S.L;
        wakeup(S.L);
    }
}
```

- 카운팅 세마포어는 두 개의 이진 세마포어 S_1과 S_2를 이용하여 구현할 수 있다. 이 때 추가로 카운팅 세마포어의 초기값으로 초기화된 정수 변수 C를 사용한다.

  - **wait** 연산

```
void wait(semaphore S){
    wait(S1);
    C--;
    if(C < 0){
        signal(S1);
        wait(S2);
    }
    signal(S1);
}
```

  - **signal** 연산

```
void signal(semaphore S){
    wait(S1);
    C++;
    if(C <= 0) signal(S2);
    else signal(S1);
}
```