CH06 Synchronization Tools

Objectives

- 임계구역 문제를 설명하고 경쟁 조건을 설명한다.
- 메모리 장벽, compare and swap 연산 및 원자적 변수를 사용하여 임계구역 문제에 대한 하드웨어 해결책을 설명한다.
- Mutex 락, 세마포, 모니터 및 조건 변수를 사용하여 임계구역 문제를 해결하는 방법을 보인다.
- 적은 , 중간 및 심한 경쟁 시나리오에서 임계구역 문제를 해결하는 도구를 평가한다.

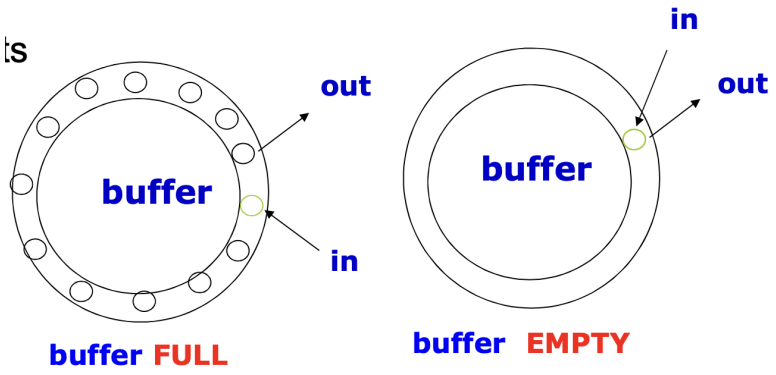
Background

- 협력 프로세스는 다른 프로세스에 영향을 미치거나 영향을 받을 수 있는 프로세스이다.
- 협력 프로세스는 다음과 같다. :
  - 논리주소 공간을 직접적으로 공유합니다,
    - 공유 메모리(코드, 데이터 등)을 Thread의 사용을 통해
  - 파일이나 메시지를 통해 공유 데이터를 허가
- 프로세스들은 동시에 실행될 수 있다.
  - 언제든지 중단(컨텍스트 전환)될 수 있으며, 부분적으로 실행이 완료된다.
- 공유 데이터에 대한 동시 액세스로 인해 데이터 불일치가 발생할 수 있다.
- 데이터 일관성을 유지하려면 협력 프로세스를 질서 있게 실행할 수 있는 메커니즘이 필요하다.

생산자 - 소비자 문제

유한 버퍼 문제 (1)

- 최대 BUFFER_SIZE-1개까지의 항목을 버퍼에 저장할 수 있다.



유한 버퍼 문제 (2)

- 유한 버퍼 문제 (1)의 단점을 없애기 위해 count라는 0으로 초기화 되어 있는 정수형 변수를 추가한다.
- 버퍼에 새 항목을 추가할 때마다 count 값을 증가시키고 버퍼에서 한 항목을 꺼낼 때마다 count 값을 감소시킨다.

생산자(Producer)

```
while (true) {
    /* produce an item in next_produced */

    while(count == BUFFER_SIZE)
        ; /* do nothing */

        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        count ++;
    }
```

소비자(Consumer)

```
while (true) {
    while(count == 0)
        ; /* do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count --;
```

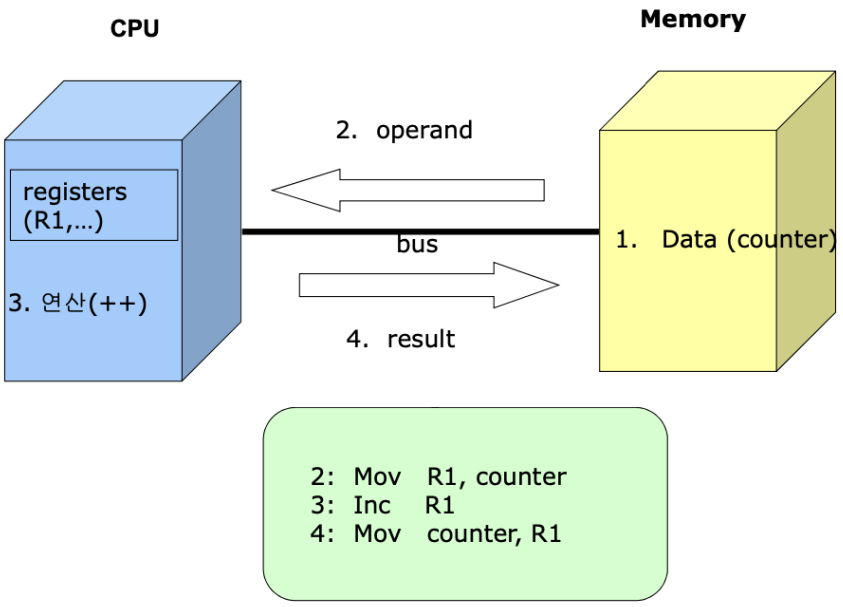
```
        /* consume the item in next_consumed */
}
```

- 생산자/소비자 코드는 독립적으로는 올바를지라도 그들을 병행적으로 수행시키면 경쟁 상태(Race Condition)가 발생할 수 있다.

Race Condition

- 여러 프로레스가 동일한 자료를 접근하여 조작하고, 그 실행 결과가 접근이 발생한 특정 순서에 의존하는 상황을 경쟁 상태라고 한다.
    - 공유 데이터의 최종 값은 어떤 프로세스가 마지막으로 완료되는지에 따라 달라진다.
    - 경주 조건을 방지하려면 동시 프로세스를 동기화해야 한다.

유한 버퍼 문제 (2) 에서의 경쟁 상황



`count ++` 의 기계 명령어

```
register1 = count
register1 = register1 + 1
count = register1
```

`count --` 의 기계 명령어

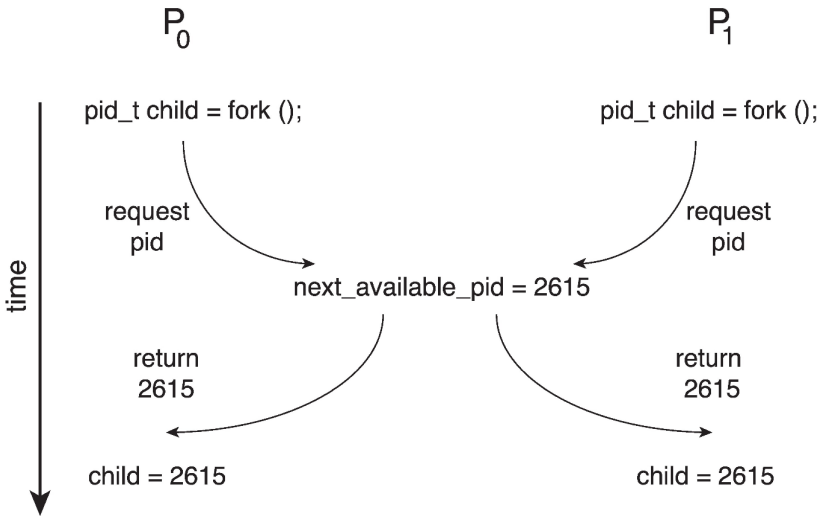
```
register2 = count
register2 = register2 - 1
count = register2
```

`count ++` 와 `count –` 문장을 병행하게 실행하는 것은 앞에서 제시한 저수준의 문장들을 임의의 순서로 뒤섞어 순차적으로 실행하는 것과 동등하다.

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 - 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}
```

따라서 소비자와 생산자가 병행하게 실행하게 되면 count 값은 4가 될 수도, 5가 될 수도, 6이 될 수도 있다.

Race Condition 예시 (2)



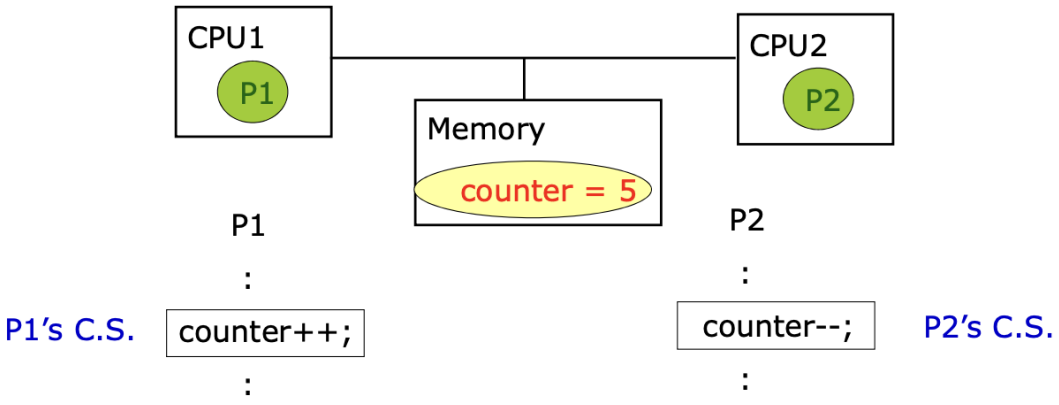
- 이 상황에서 P0 및 P1 두 프로세스는 fork() 시스템 콜을 사용하여 자식 프로세스를 생성한다. fork()는 새로 생성된 프로세스의 프로세스 식별자를 부모 프로세스로 반환한다는 점을 상기하라. 이 예에서, 커널 변수 next_available_pid에 경쟁 조건이 있으며 이 변수는 다음 사용 가능한 프로세스 식별자의 값을 나타낸다.
- 상호 배제가 제공되지 않으면 동일한 프로세스 식별자 번호가 두 개의 다른 프로세스에 배정될 수 있다.

Synchronization

- Race Condition
  - 다수의 스레드 또는 프로세스가 동일한 데이터를 동시에 읽고 쓰는 상황과 실행 결과가 액세스가 발생하는 특정 순서에 따라 달라지는 상황
- Critical Section
  - 공유 자원에 대한 액세스를 요구하고 다른 프로세스가 대응하는 중요한 섹션에 있는 동안 실행될 수 없는 프로세스 또는 스레드 내의 코드 섹션
- Mutual Exclusion
  - 하나의 프로세스가 임계 섹션에 있을 때, 다른 프로세스가 대응하는 임계 섹션에 있을 수 없다는 요구 사항.

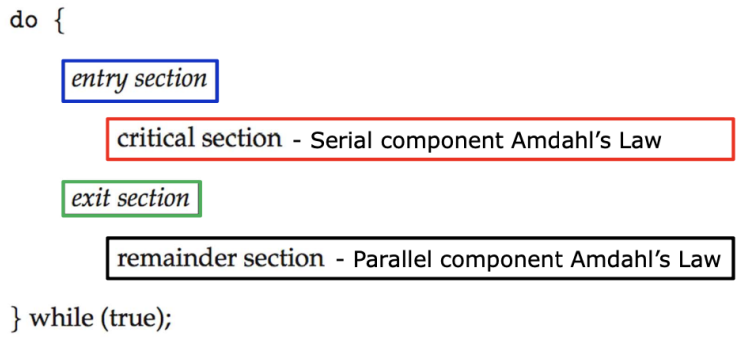
Critical Section

> 💬 한 프로세스가 자신의 임계구역에서 수행하는 동안에는 다른 프로세스들은 그들의 임계구역에 들어갈 수 없다.



Critical Section Problem

- 프로세스들이 데이터를 협력적으로 공유하기 위해서 자신들의 화ㄹ동을 동기화할 때 사용할 수 있는 프로토콜을 설계하는 것



임계구역 문제 해결안 : 3가지 요구 조건

1. 상호 배제 (mutual exclusion)
    - 프로세스 Pi가 자기의 임계구역에서 실행된다면, 다른 프로세스들은 그들 자신의 임계구역에서 실행될 수 없다.
2. 진행 (Progress)
    - 자기의 임계구역에서 실행되는 프로세스가 없고 그들 자신의 임계구역으로 진입하려고 하는 프로세스들이 있다면, 나머지 구역에서 실행 중이지 않은 프로세스들만 다음에 누가 그 임계구역으로 진입할 수 있는지를 결정하는 데 참여할 수 있으며, 그 선택은 무한정 연기될 수 없다.
3. 한정된 대기(bounded waiting)
    - 프로세스가 자기의 임계구역에 진입하려는 요청을 한 후부터 그 요청이 허용될 때까지 다른 프로세스들이 그들 자신의 임계구역에 진입하도록 허용되는 횟수에 한계가 있어야 한다.

OS에서의 임계구역 핸들링

- 운영체제 내에서 임계구역을 다루기 위해서 선점형 커널과 비선점형 커널의 두 가지 일반적인 접근법이 사용된다.
    - 선점형 커널 : 프로세스가 커널 모드에서 수행되는 동안 선점되는 것을 허용
    - 비선점형 커널 : 커널 모드에서 수행되는 프로세스의 선점을 허용하지 않고 커널 모드 프로세스는 커널을 빠져나갈 때까지 또는 봉쇄될 때까지 또는 자발적으로 CPU의 제어를 양보할 때까지 계속 수행된다.

선점형 커널의 장점

- 커널 모드 프로세스가 대기 중인 프로세스에 처리기를 양도하기 전에 오랫동안 실행할 위험이 적기 때문에 선점형 커널은 더 응답이 민첩할 수 있다.

- 실시간 프로세스가 현재 커널에서 실행 중인 프로세스를 선점할 수 있기 때문에 실시간 프로그래밍에 더 적당하다.

Peterson’s Solution

- 현대 컴퓨터에서 올바르게 실행됨을 보장할 수는 없는 방법이지만, 임계구역 문제를 해결하기 위한 좋은 알고리즘적인 설명을 제공하고 상호 배제, 진행, 한정된 대기의 요구조건을 중점으로 다루는 소프트웨어를 설계하는 데 필요한 복잡성을 잘 설명하기 때문에 피터슨의 해결책을 제시한다.
- 피터슨의 해결안은 두 프로세스가 공유하는 두개의 데이터를 필요로 한다.

```
/* 각각의 프로세스가 임계구역을 사용하고자 하는 표시이다.
예를 들어 flag[0]==true이면 0번째 프로세스가 임계구역의 자료를 사용하고
boolean flag[2];
/*프로세스의 차례를 의미, 예를 들어 turn=0이라면 0번째 프로세스의 차례*/
int turn;
```

프로세스 i의 구조(Pi)

```
do{
    /* entry section */
    flag[i] = true; // i번째 프로세스가 임계구역을 사용하고 싶음.
    turn = j;       // 현재 j번째 프로세스가 임계구역을 사용할 차례

    // 현재 j번째 프로세스가 임계구역을 사용 중
    // 반복문 탈출 조건
    // 1. flag[j]=false or
    // 2. turn=i
    while(flag[j]==true && turn==j)
    {

    }

    /* critical section */

    /* exit section */
    flag[i] = false;    // i번째 프로세스가 임계구역을 다 사용했기 때
```

```
    /* remainder section */
}while(true)
```

→ 요구사항 [1] 상호배제(mutual - exclusion) 만족

→ 요구사항 [2] 진행(progress) 만족

→ 요구사항 [3] 한정된 대기(bounded waiting) 만족

Synchronization Hardware

- 많은 시스템에서 중요한 섹션 코드에 대한 하드웨어 지원을 제공한다.
- Uniprocessors : 인터럽트를 비활성화할 수 있음
  - 현재 실행 중인 코드가 선점 없이 실행된다.
  - 일반적으로 멀티프로세서 시스템에서는 너무 비효율적이다.
    - 이를 사용하는 운영 체제는 광범위하게 확장할 수 없다.
- 현대 컴퓨터는 특별한 원자적(atomic) 하드웨어 지침을 제공한다.
  - Atomic = Interrupt 불가
  - 메모리 워드 및 설정값의 내용을 테스트하는 Test-and-set 명령어
  - 두 개의 메모리 단어의 내용을 바꾸는 Compare-and-Swap 명령어

Locks를 이용한 임계구역 문제의 해결

- 모든 해결책은 locking 아이디어로부터 파생된다.

```
do {
    acquire lock
            //critical section
    release lock
            // remainder section
} while (TRUE);
```

test_and_set 명령어

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
```

```
        *target = true;
        return rv;
}
```

- 원자적으로 실행된다.
- 전달된 파라미터의 원래 값을 반환
- 전달된 파라미터의 새 값을 true로 설정

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

        /* critical section */
    lock = false;
        /* remainder section */
    } while(true);
```

compare_and_swap 명령어

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

- 원자적으로 실행
- 전달된 매개 변수 값의 원래 값을 반환
- 변수 값을 전달된 매개 변수 new_value의 값으로 설정하되 *value == expected가 true인 경우에만 설정. 즉, 교환은 이 조건에서만 이루어진다.

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

        /* critical section */

        lock = 0;

        /* remainder section */
}
```

acquire lock

release lock

Mutex Lock

- 임계구역을 보호하고, 경쟁 조건을 방지하기 위해 mutex 락을 사용한다.
- mutex라는 용어는 mutual exclusion의 축약으로, 상호 배제 lock으로 임계구역을 보호하고 경쟁 상황을 방지한다.
- 즉, 프로세스는 임계구역에 들어가기 전에 반드시 락을 얻어야 임계구역에 들어갈 수 있고, 나갈 땐 락을 반납하고 나가야 한다. acquire() 함수가 Lock을 획득하고 Release() 함수가 Lock을 반환한다.
- Mutex 락은 available이라는 boolean 변수를 가지는데 이 변수 값이 Lock의 가용 여부를 표시한다. Lock이 가용하면 acquire() 호출은 성공하고 Lock은 곧 사용 불가 상태가 된다. 사용 불가 상태의 Lock을 획득하려고 시도하는 프로세스는 Lock이 반환될 때까지 봉쇄된다.

```
acquire() {
    while(!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

```
while (true) {
    acquire lock
            //critical section
    release lock
```

```
            // remainder section
}
```

- 뮤텍스 락은 스핀락(spinlock)이라고도 한다. 락을 사용할 수 있을 때까지 프로세스가 "회전"하기 때문이다.

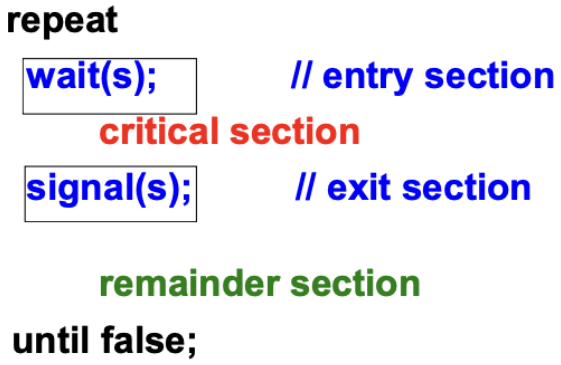
Semaphores

- mutex와 유사하게 동작하지만 프로세스들이 자신들의 행동을 더 정교하게 동기화할 수 있는 방법을 제공하는 강력한 도구
- 세마포 S는 정수 변수로서, 초기화를 제외하고는 단지 두 개의 표준 원자적 연산 wait() [=p()]과 signal()[=v()]로만 접근할 수 있다.

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

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

- 공유 데이터
  - semaphore s ; //n개의 프로세스 간에 공유된 세마포어(1로 초기화)



Usage of Semaphore - 두 프로세스 간 동기화

- 문제

- P1과 P2(동시에 돌아가는 두 프로세스)
- P2의 S2는 P1의 S1 이후에만 실행되어야 한다.

Two Types of Semaphores

- Binary Semaphore
  - 정수 값의 범위는 0에서 1 사이
  - mutex lock처럼 동작
- Counting Semaphore
  - 정수 값은 제한되지 않은 도메인에서 범위를 지정할 수 있다.