프로세스 동기화

왜 프로세스 동기화가 필요한가?

- 협력적 프로세스 : 논리적 주소를 공유하고 있어 다른 프로세스에게 영향을 주거나 받는 프로세스

협력적 프로세스는 **Race Condition**의 위험이 있다.

Race Condition

: 여러개의 프로세스가 동일한 데이터에 접근하고, 이 프로세스들이 실행되는 순서에 따라 실행결과가 달라지는 상황

Race Condition이 발생되지 않게 하기위해서는 협력적 프로세스들의 **질서있는 실행을 보장**하여 **데이터 무결성을 보장이 필요하다 ⇒**

데이터 무결성을 위해서는 프로세스끼리 데이터에 대한 동기화가 필요하다.

임계구역 문제

- **임계구역(Critical Section)** : 다른 프로세스와 공유하고 있는 데이터에 접근하여 조작할 수 있는 구역
- **임계구역 문제** : 임계구역으로 지정되어야 할 코드 영역이 임계구역으로 지정되지 않았을 때 발생할 수 있는 문제

임계구역 문제 해결의 조건

1. **상호배제,상호배타 Mutual exclusion**
   - 프로세스 P가 자기의 임계구역에서 실행된다면, 다른 프로세스들은 그들 자신의 임계구역에서 실행될 수 없다
2. **진행 Progress**
   - 임계구역에 프로세스가 존재하지 않는 경우, 다른 프로세스가 접근할 수 있어야한다.
   - 임계구역에 들어간 프로세스가 없는 상태에서 들어가려는 프로세스가 여러 개 있다면 어떤 프로세스가 들어갈지를 결정하는 것으로, 이것은 유한 시간 이내에 이루어져야 합니다.
3. **유한 대기 Bounded waiting** = 한정된 대기
   - 다른 프로세스의 ***기아**를 방지하기 위해, 한 번 임계 구역에 들어간 프로세스는 그 다음에 임계 구역에 들어갈 때 제한을 두어야 합니다.
     - 기아 : 프로세스가 임계 영역에 들어가기 위해 무한정으로 기다리는 현상으로 여러 프로세스가 부족한 자원을 점유하기 위해 경쟁할 때 발생합니다
   - 임계구역으로 진입하기 위해서 대기하는 모든 스레드는 유한 시간이내에 해당 임계구역으로 진입할 수 있어야 합니다.
   - 진행과 다른 점
     - 진행은 말 그래도 임계구역 해결을 위해 코드 실행이 멈추지 않는 것을 의미
     - 유한대기는 임계구역 코드 실행에 들어간 프로세스 외 나머지 프로세스들이 기아 현상이 나타나지 않도록 하기 위해 이미 임계구역을 실행한 프로세스를 제한하는 것

Critical Section & Remainder Section

각 프로세스는 자신의 임계구역으로 진입하려면 **진입 구역(entry section)**에서 진입 허가를 요청해야 하고, 임계구역 뒤에는 **퇴출 구역(exit section)**이 따라올 수 있다. 코드의 나머지 부분들을 총칭하여 **나머지 구역(reaminder section)**이라고 한다.

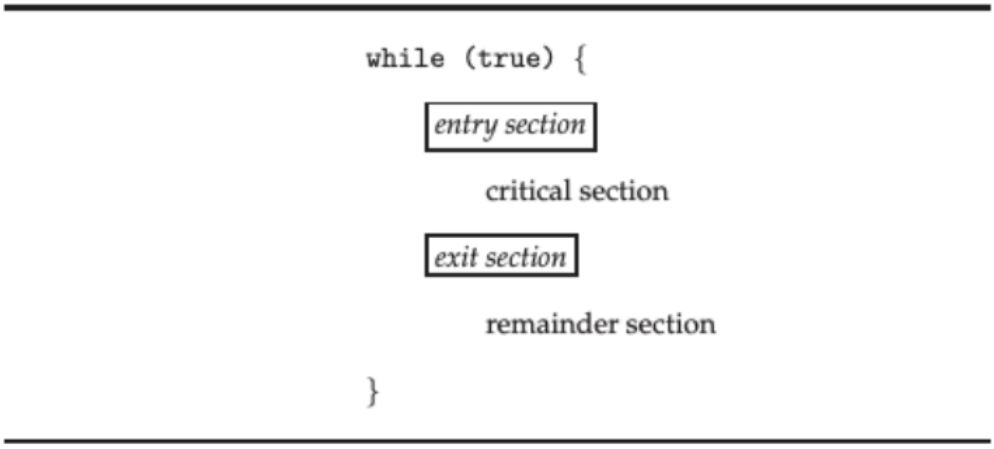

Figure 6.1 General structure of a typical process.

Peterson 해결안

Peterson 해결안은 Critical Section과 Remainder Section을 번갈아 가며 실행하는 두 개의 프로세스로 한정된다.

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

- turn → 임계구역에 접근할 프로세스의 번호
- flag → 해당 인덱스의 프로세스가 준비가 되었는지

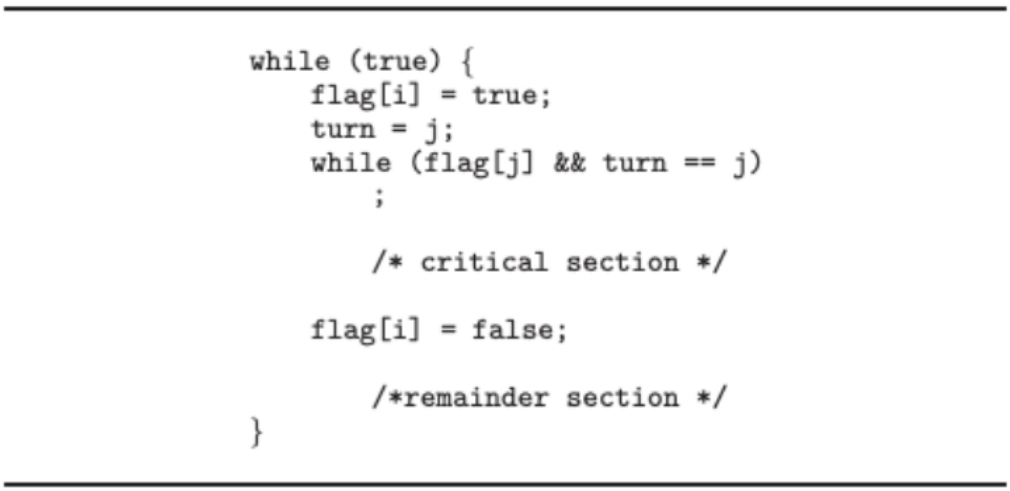

Figure 6.3 The structure of process P_i in Peterson's solution.

Critical Section으로 진입하기 위해서는 Pi는 먼저 flag[i]를 참으로 만들고, turn을 j로 지정한다. 이렇게 함으로써 Pi는 Pj가 Critical Section으로 진입하기를 원한다면 진입 가능하다는 것을 보장한다. 만일 두 프로세스가 동시에 진입하기를 원한다면 진입 turn은 거의 동시에 i와 j로 지정될 것이다. 이 때의 경우 turn의 궁극적인 값이 둘 중 누가 먼저 Critical Section으로 진입할 것인가를 결정한다.

flag[j]가 false(Pj가 remainder section 수행)가 되거나 turn이 i(Pj는 준비완료됬고, Pi가 Entry Section에서 대기하고 있음)일 경우 Pi는 Critical Section에 들어갈 수 있다.

Peterson의 해결안이 Ciritical Section 문제를 해결하기 위해선 위에서 명시한 것 처럼 3가지 요구조건을 만족해야 한다. 같이 살펴보자.

1. 상호 배재(Mutual Exclusion)
    - flag[2]와 turn 변수에 의해서 하나의 프로세스만 Critical Section에서 연산을 수행할 수 있음으로 Mutual Exclusion는 지켜진다.
2. 진행(Progress)
    - 각 프로세스가 자신이 Critical Section을 수행할 동안 while문에서 다른 프로세스를 유한하게 대기하도록 만드는 방법을 통하여 Progress를 지킬 수 있다.
3. 한정된 대기(bounded waiting)
    - 각 프로세스들은 Critical Section에 진입하려는 요청을 한 후부터 다른 프로세스가 Critical Section을 수행하는 동안 유한하게 대기함으로 bounded waiting 또한 지켜진다.

3가지 하드웨어 명령어

Critical Section 문제의 소프트 웨어 기반 해결책은 최신 컴퓨터 아키텍처에서 작동하지 않을 수 있다.

읽고 쓰는 작업이 서로 다른 인스트럭션으로 이루어져 있기 때문이다. 인스트럭션이 수행되는 도중에는 인터럽트가 올 수 없기 때문에, 만약 읽기와 쓰기가 하나의 인스트럭션에서 이루어져 **원자적인** 명령어를 통해 Critical Section문제를 해결할 수 있다.

- **메모리 장벽** : 컴퓨터 아키텍처는 메모리의 모든 변경 사항을 다른 모든 프로세서로 전파하는 명령어를 제공하여 다른 프로세서에서 **실행 중인 스레드에 메모리 변경 사항이 보이는 것을 보장**한다. 이러한 명령어를 **메모리 장벽(Memory Barriers) 또는 메모리 펜스(Memory Fences)**라고 한다.
    - 메모리 장벽은 매우 낮은 수준의 연산으로 간주하며 일반적으로 Mutual Exclusion을 보장하는 특수 코드를 작성할 때 커널 개발자만 사용한다.

- **하드웨어 명령어 : 원자적인 연산**
    - test_and_set() : 파라미터로 온 변수의 내용을 검사해서 변경
    - compare_and_set() : 파라미터로 온 두가지의 변수를 비교하여 변경
        - test_and_set() 명령어와 마찬가지로 두 개의 워드에 대해 원자적인 연산을 하지만 두 워드의 내용 교환에 기반을 둔 기법이다.
        - 명령어를 직접하용하진 않고 **원자적 변수**를 활용한다.
            - 원자적 변수는 정수 및 부울과 같은 기본 데이터 유형에 대한 원자적 연산을 제공하는데, **CAS**는 이런 원자적 연산을 구현할 때 내부적으로 활용된다.

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

   return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

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

      /* critical section */

   lock = false;

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

Figure 6.6 Mutual-exclusion implementation with `test_and_set()`.

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

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7 The definition of the atomic `compare_and_swap()` instruction.

Mutex Locks

mutex lock은 mutual exclusion lock의 축약 형태로서 Critical Section을 해결하기위한 하드웨어 기반의 해결책보다 상위 수준의 해결책이다.

Mutex Lock의 기본 개념은 프로세스는 Critical Section에 **들어가기 전에** 반드시 **lock을 획득**해야 하고 Critical Section을 **빠져나올 때 lock을 반환**해야 한다.

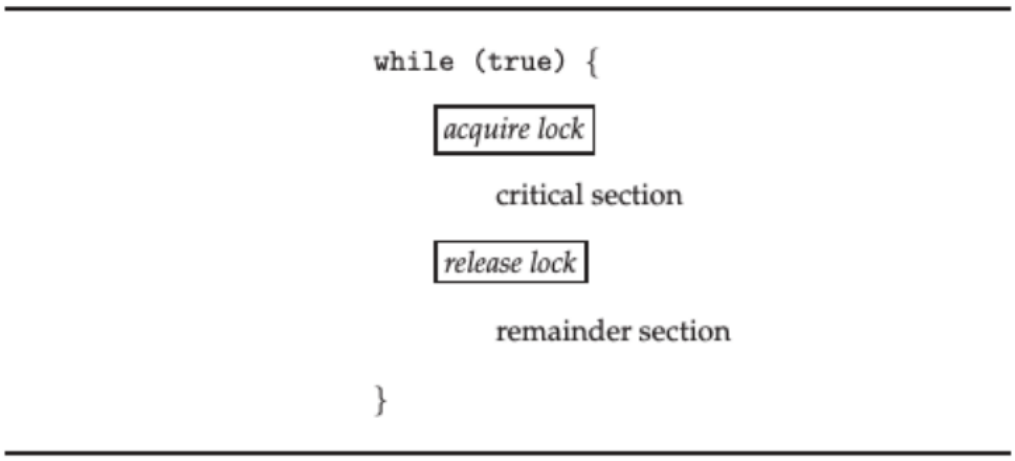

Figure 6.10 Solution to the critical-section problem using mutex locks.

Mutex Lock에 필요한 3가지

1. acquire() : Lock을 획득하는 함수

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

단점은 **바쁜 대기(busy waiting)**를 해야 한다는 것이다. 이러한 busy waiting은 대기하는 다른 프로세스가 계속해서 acquire 함수를 호출해서 다른 프로세스가 생산적으로 사용할 수 있는 CPU 주기를 낭비한다.

프로세스가 락을 기다려야 하고 문맥 교환에 상당한 시간이 소요될 때 문맥 교환이 필요하지 않다는 장점이 있다.

최신 다중 코어 컴퓨팅 시스템에서 스핀락은 많은 운영체제에서 널리 사용된다. **일반적으로 락이 유지되는 기간이 문맥 교환을 두 번(1.스레드를 대기상태로 2.대기중인 스레드를 복원) 하는 시간보다 짧은 경우 Mutext Lock을 사용한다.**

2. release() : Lock을 반환하는 함수

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

3. available : Lock을 사용할 수 있는 지 여부를 표시하는 변수

세마포

세마포 S는 정수 변수로 사용되는데, 두개의 **원자적** 연산 **wait() 와 signal()**로 접근할 수 있다.

- **wait()** : 자원을 사용하려는 프로세스가 wait 연산을 수행한다. **세마포의 값은 감소한다.**

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

- **signal()** : 프로세스가 자원을 방출할때 signal() 연산을 수행하고 **세마포의 값은 증가한다**. 세마포의 값이 **0이 되면 모든 자원이 사용중**임을 나타낸다. 이후 자원을 사용하려는 프로세스는 세모포값이 0보다 커질 때까지 봉쇄된다.

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

세마포 구현

busy waiting을 피하기 위해 세마포 S를 대기하면서 일시 중지된 프로세스는 다른 프로세스가 signal() 연산을 실행하면 재시작되어야 한다. 프로세스는 s**leep() 연산에 의해서 일시 중지**되고 **wakeup() 연산에 의하여 재시작된다.**

1. **세마포 구조**

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

2. wait()

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S -> list;
    sleep();
  }
}
```

3. signal()

```
signal(semaphore *S) {
  S->value ++;
  if (S->value <= 0) {
    remove a process P from S -> list;
    wakeup(P);
  }
}
```