**Chapter 6. 프로세스 동기화**

**(Process Synchronization)**

협력적 프로세스는 시스템 내에서 실행 중인 다른 프로세스의 실행에 영향을 주거나 영향을 받는 프로세스이다. 이경우 공유 데이터에 대한 동시 접근은 데이터의 비일관성을 낳을 수 있다.

**6.1. 배경(Background)**

프로세스가 병렬적으로 실행되었을때 이들은 서로 비동기적으로 수행하면서 데이터를 공유할 가능성이 크다.

**Example**

* counter++
  + register1 = counter
  + register1 = register1 + 1
  + counter = register1
* counter--
  + register2 = counter
  + register2 = register2 - 1
  + counter = register2
* count가 5일때 두 프로세스 실행 예제
  + 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}

**경쟁상황(RaceCondtion)**

* 이처럼 두개의 프로세스가 동시에 변수 counter를 조작하고 그 실행결과가 접근이 발생한 특정 순서에 의존하는 상황을

**경쟁 상황(Race Condition)** 이라고 한다.

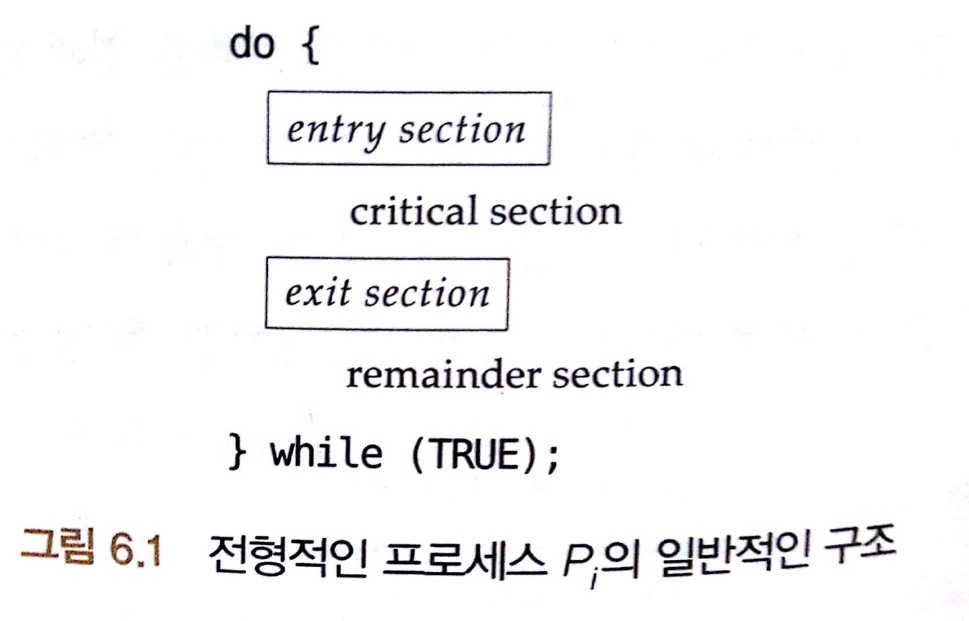
**6.2 임계구역 문제(The Critical-Section Problem)**

각 프로세스는 **임계구역(Critical Section)** 이라고 부르는 코드 부분을 포함하고 있고, 그 안에서는 다른 프로세스와 공유하는 변수를 변경하거나, 테이블을 갱신 하거나 파일을 쓰거나 하는 등의 작업을 수행한다.

**임계구역 특징**

임계구역의 가장큰 특징은 한 프로세스가 자신의 임계구역에서 수행하는 동안에는 다른 프로세스들은 그들의 임계구역에 들어갈 수 없다는 사실이다.

* 전형적인 프로세스의 일반적인 임계구역 구조

[](https://github.com/itlockit2/OperatingSystem/blob/master/1.jpg)

**임계구역 구조**

**진입 구역(Entry Section)**

* 자신의 임계구역으로 진입하려면 진입 허가를 요청해야하는데 이러한 요청을 구현하는 코드부분이다.

**퇴출 구역(Exit Section)**

* 임계구역에서 빠져나올때 다른 프로세스가 임계구역에 들어갈수 있게끔 처리하는 코드부분이다.

**나머지 구역(Remainder Section)**

* 임계구역과 관련없는 나머지 코드부분이다.

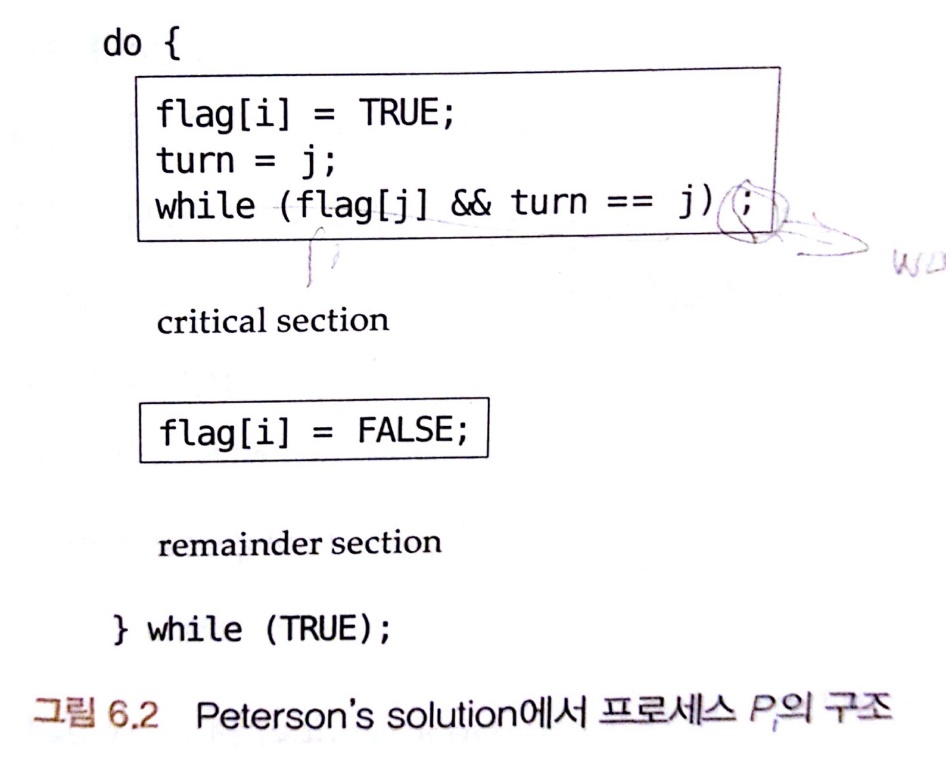
**임계구역 문제 해결안**

1. 상호배제(Mutual Exclusion) : 프로세스 P가 자기의 임계구역에서 실행된다면, 다른 프로세스들은 그들 자신의 임계구역에서 실행될 수 없다.
2. 진행(Progress) : 임계구역으로 진입하려고 하는 프로세스들이 있다면 나머지구역(Remainder Section)을 실행하고 있는 프로세스가 아닌 진입구역(Entry Section)을 실행중인 프로세스들만 경쟁할 수 있으며 이 선택은 무한정 연기될 수 없다.
3. 한정된 대기(Bounded waiting) : 프로세스가 자기의 임계구역에 진입하려는 요청을 한 후부터 그 요청은 유한한 시간 이내에 허용되어야한다.

**선점형 커널 vs 비선점형 커널**

* 선점형 커널은 프로세스가 커널 모드에서 수행되는 동안 선점되는 것을 허용한다. 따라서 경쟁조건이 생길수 있지만 비선점형 커널보다 응답이 민첩하기 때문에 더 선호하는 커널이다.
* 비선점형 커널은 모든 프로세스는 자발적으로 CPU의 제어를 양보하기 전까지 계속 수행된다. 따라서 경쟁조건을 염려할 필요가 없다.

**6.3 피터슨의 해결안(Peterson's Solution)**

[](https://github.com/itlockit2/OperatingSystem/blob/master/2.jpg)

* turn은 임계구역으로 진입할 순번을 나타낸다. 즉 turn == i이면 프로세스 Pi가 임계구역에서 실행될 수 있다.
* flag배열은 프로세스가 임계구역으로 진입할 준비가 되었다는것을 나타낸다. 예를 들어 flag[i]가 TRUE이면 Pi가 임계구역으로 진입할 준비가 되었다는뜻이다.

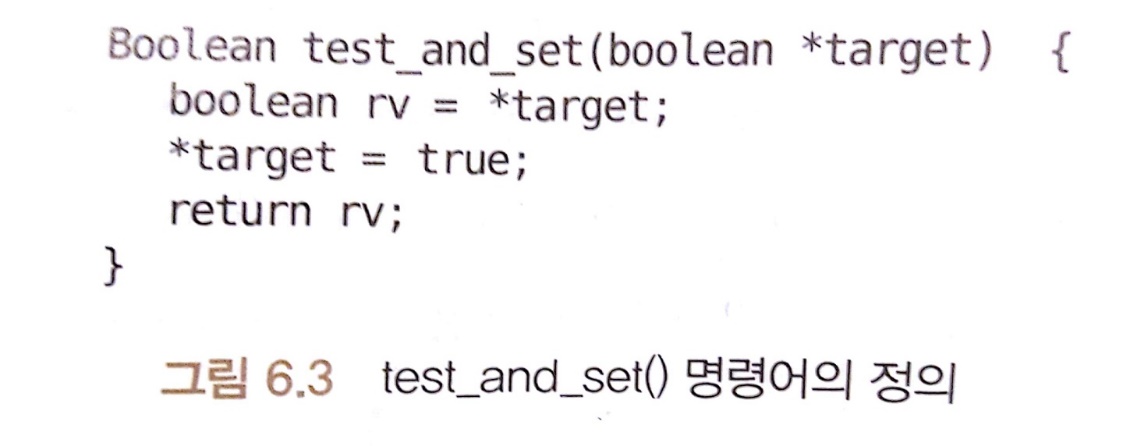
**피터슨 해결책 증명**

1. 상호 배제가 제대로 지켜지는가.
   * flag[0] == flag[1] == true라고 해도 turn값은 i , j 둘중 하나만 가질수있다. 따라서 상호배제가 지켜진다.
2. 진행에 대한 요구조건을 만족시키는가.
3. Bounded Waiting 조건을 만족시키는가.
   * Pj가 임계구역을 빠져나갈때 flag[j]의 값을 FALSE로 변경해줌으로 Pi가 진입하게 된다. 그다음 Pj는 자기자신 스스로 turn 값을 i로 바꾸어 임계구역에 들어가지 못하게 해야한다. Pi 프로세스 임계구역에서는 turn값을 바꾸지 않기 때문이다.

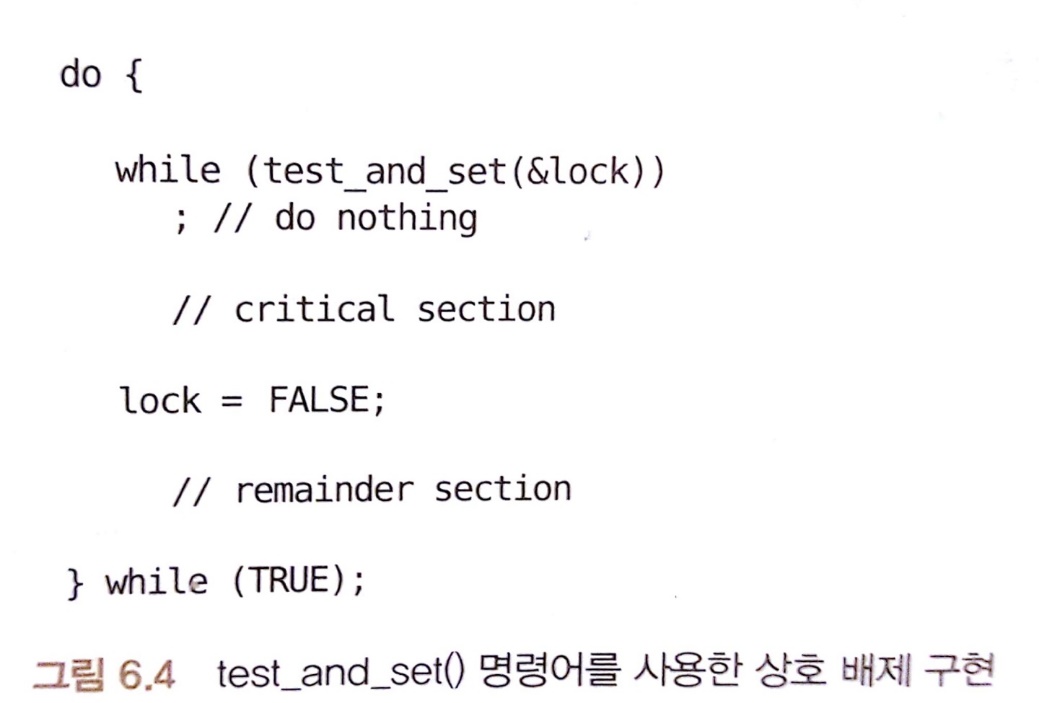
**6.4 동기화 하드웨어(Synchronization Hardware)**

Perterson's Solution는 소프트웨어 기반 해결책이다. 인터럽트 되지 않은 하드웨어 명령어를 통해서 임계구역 문제를 상대적으로 간단하게 해결할수 있다.

**Test\_and\_Set() 명령어의 정의**

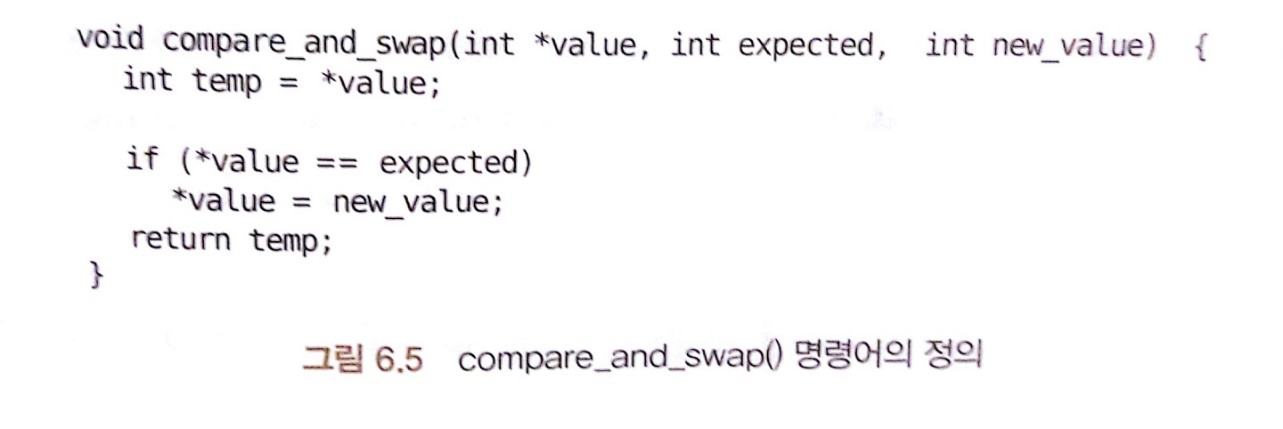
[](https://github.com/itlockit2/OperatingSystem/blob/master/3.jpg)

**Test\_and\_set() 명령어를 사용한 상호 배제 구현**

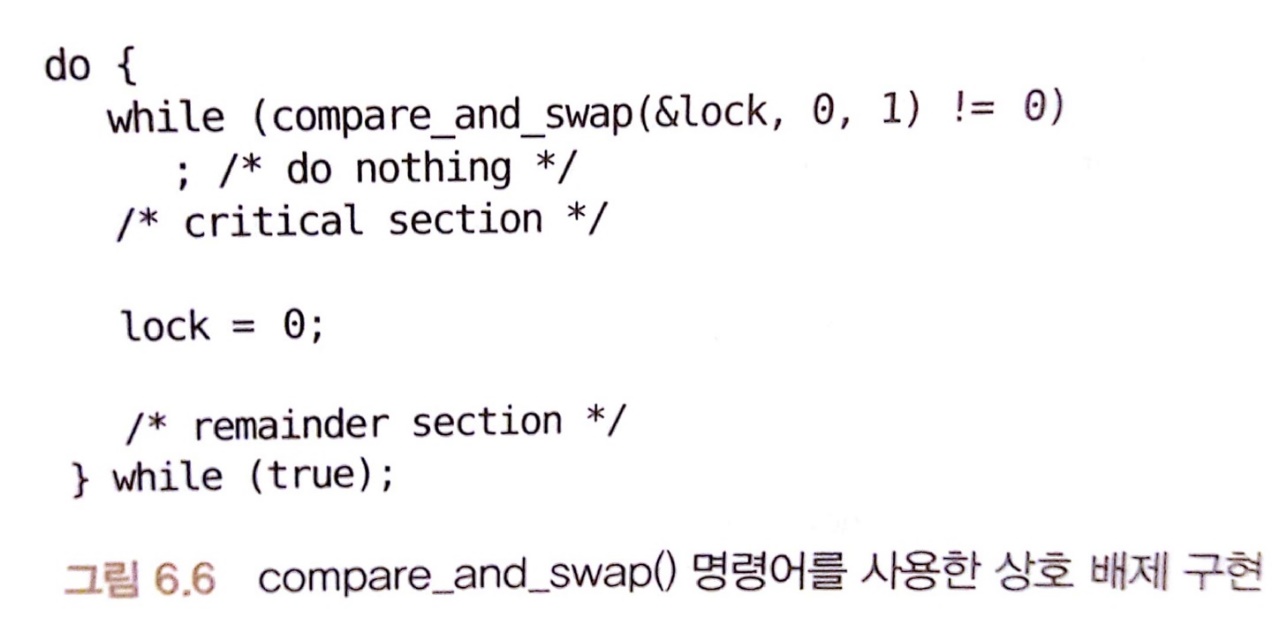
[](https://github.com/itlockit2/OperatingSystem/blob/master/4.jpg)

* 만약 lock이 false라면 false를 리턴하고 lock은 true만드는 특징을 가지고 있다.

**compare\_and\_swap() 명령어의 정의**

[](https://github.com/itlockit2/OperatingSystem/blob/master/5.jpg)

**compare\_and\_swap() 명령어를 사용한 상호 배제 구현**

[](https://github.com/itlockit2/OperatingSystem/blob/master/6.jpg)

어떠한 경우에는 compare\_and\_swap 명령어는 value의 원래 값을 반환한다.

만약 lock이 0 이라면 lock이 1로 변경되고 0을 리턴한다.

만약 lock이 1이라면 1을 리턴한다.

따라서 lock이 0일 경우에만 프로세스가 임계구역으로 진입할 수 있다.

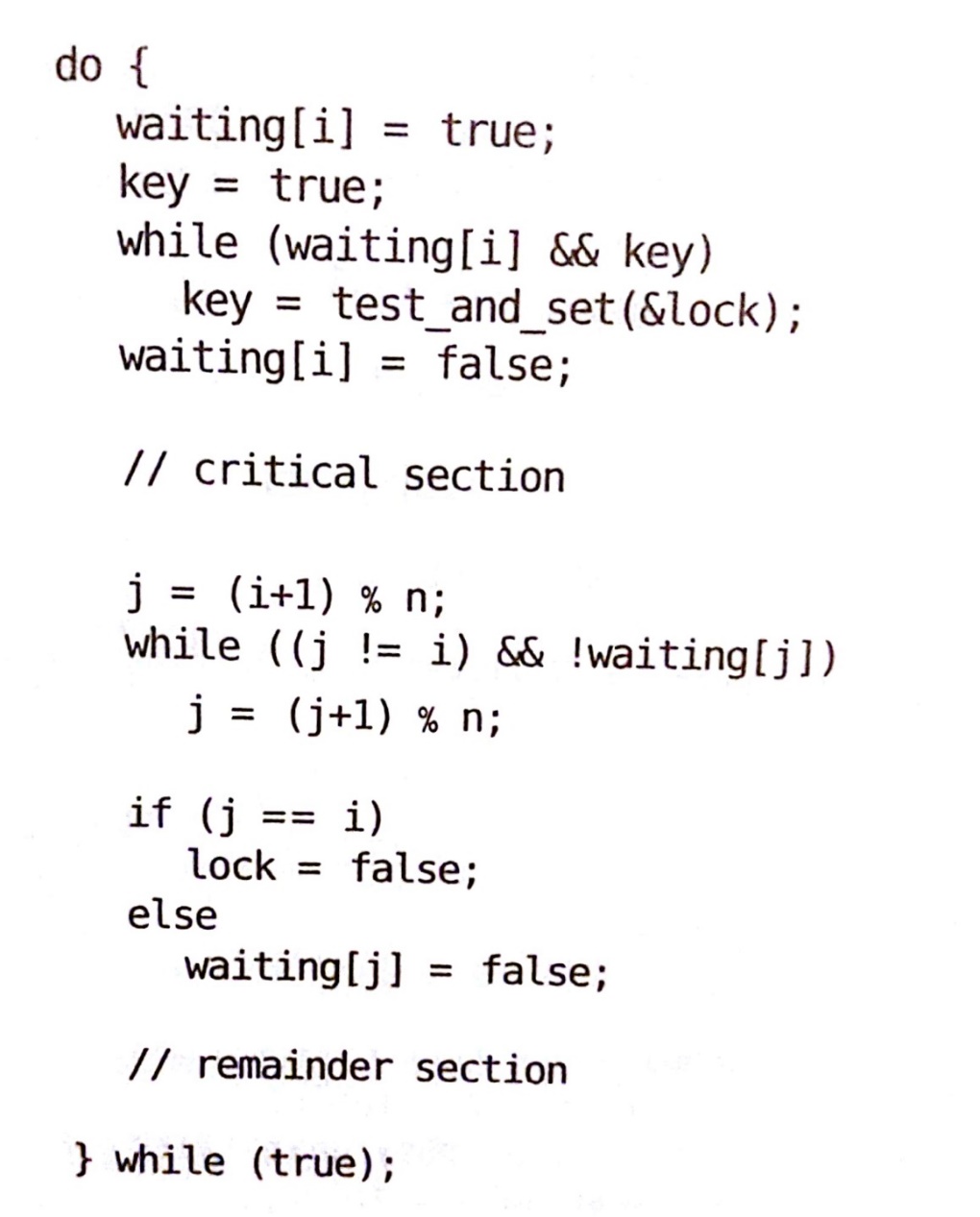
* 문제점
  + 위 알고리즘은 상호 배제 조건은 만족시키지만 한정된 대기 조건을 만족 시키지 못한다.

Perterson's Solution 같은경우를 보면 turn이라는 변수를 통해 제어를 하여

두 프로세스가 번갈아 실행되게 하지만 위 알고리즘은 프로세스들의 while문의 조건식이 같기 때문에

Pj가 들어갈 수 있으면 Pi도 들어갈수 있다. 따라서 하나의 프로세스가 독점 될 가능성이 존재한다.

**Test\_and\_set() 명령어를 사용한 한정된 대기 조건을 만족시키는 상호배제**

[](https://github.com/itlockit2/OperatingSystem/blob/master/7.jpg)

boolean waiting[n] 이라는 변수를 통해 한정된 대기 조건을 만족시킨다.

Pi가 임계구역에 진입하기 위해서는 waiting[i]가 FALSE 이거나 key 가 FALSE 이여야한다.

waiting 변수는 다른 프로세스가 임계구역을 떠날때만 변경되며

key값은 test\_and\_set() 명령어를 통해서만 변경된다.

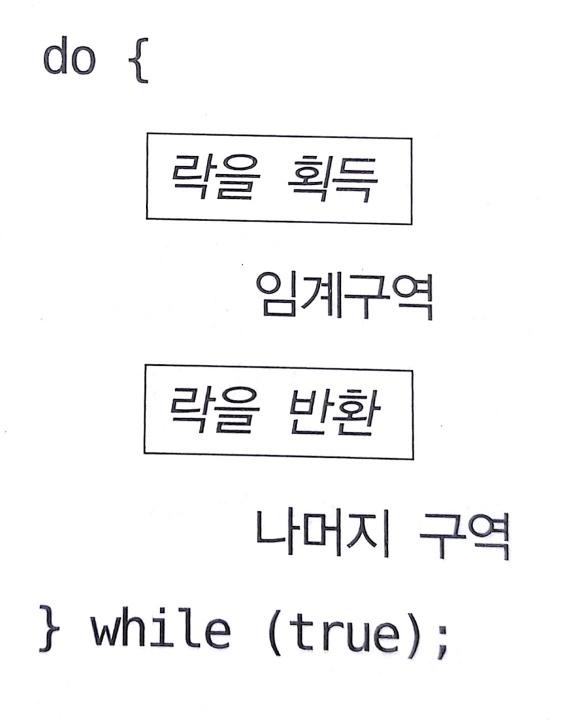
한 프로세스가 임계구역을 떠날때 waiting 배열을 순환하면서 다음 프로세스를 임계구역에 들어 갈 수있게 해주므로

**한정된 대기시간**을 만족시켜준다.

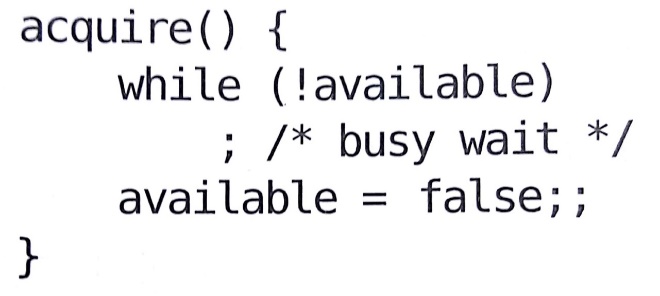
**6.5 Mutex Locks**

6.4에서 제시한 임계구역 문제는 하드웨어 기반 해결책으로 소프트웨어 도구를 이용하여 해결하는 방안이 있는데 그것이 바로 MUTEX락 이다.

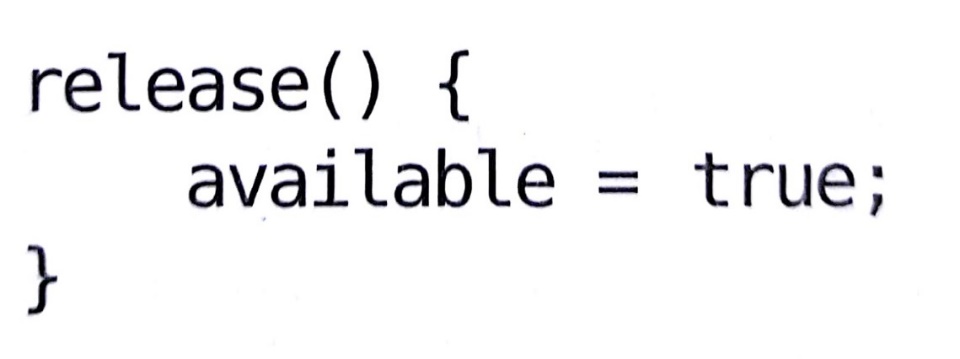
**MUTEX락을 이용한 임계구역 문제 해결 방안**

[](https://github.com/itlockit2/OperatingSystem/blob/master/8.jpg)

**acquire() 함수의 정의**

[](https://github.com/itlockit2/OperatingSystem/blob/master/9.jpg)

**release() 함수의 정의**

[](https://github.com/itlockit2/OperatingSystem/blob/master/10.jpg)

**MUTEX 락의 단점**

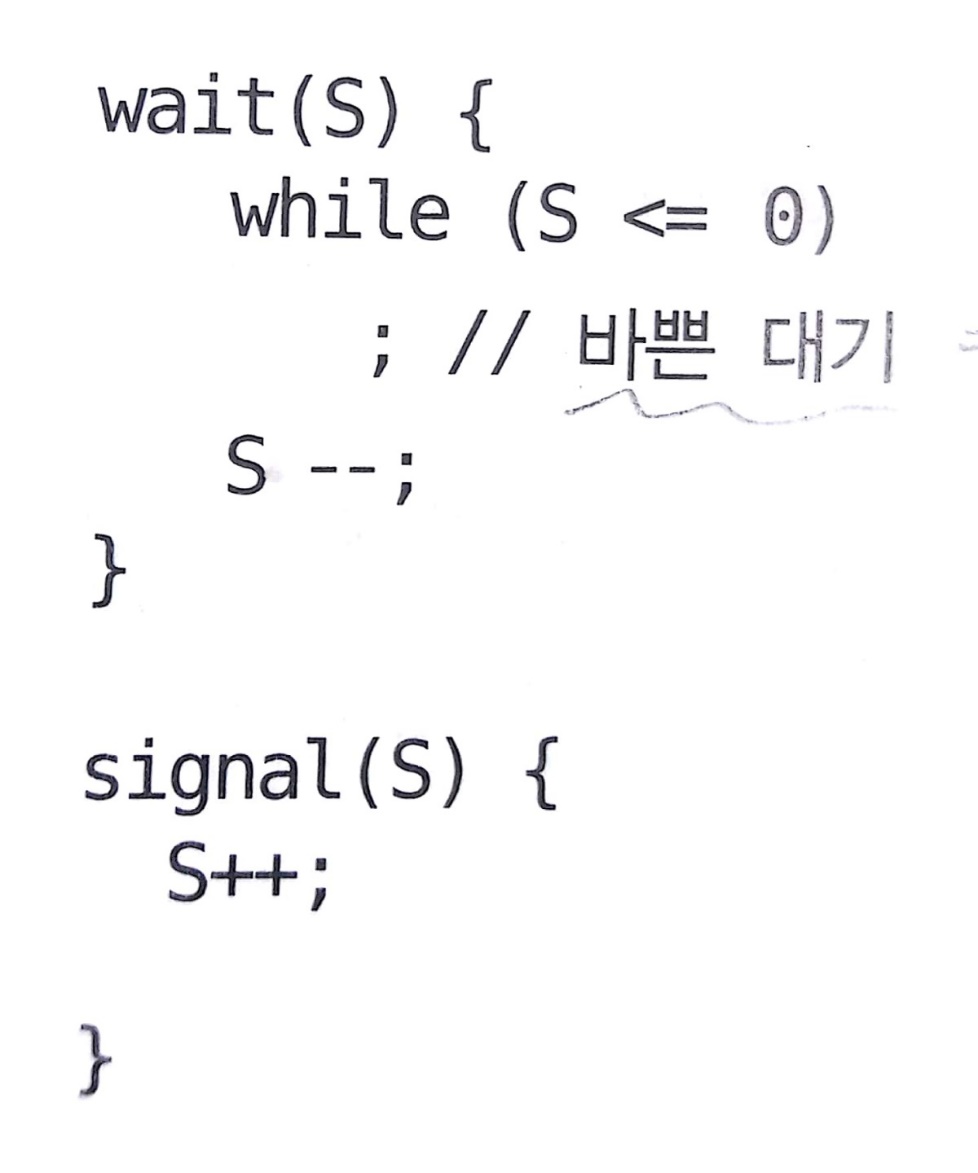
MUTEX락 뿐만 아니라 test\_and\_set()과 compare\_and\_swap() 방법또한 임계구역에 들어가기 위해서 반복문을 계속 실행해야 하는 **바쁜대기(Busy waiting)** 를 해야한다는 것이다. 이러한 Lock 방식을 **SpinLock** 이라고도 부른다.

**6.6 세마포(Semaphores)**

앞에서 언급한 mutex와 유사하게 동작하지만 프로세스들이 자신들의 행동을 더 정교하게 동기화 할 수 있는 방법이다.

세마포 S는 정수 변수로서, 초기화를 제외하고는 wait()와 signal()로만 접근이 가능하다.

**wait()와 signal()연산 정의**

[](https://github.com/itlockit2/OperatingSystem/blob/master/11.jpg)

**6.6.1. 세마포 사용법**

* 카운팅 세마포(Counting Semaphore) : 카운팅 세마포에서 S의 값은 제한 없는 영역을 갖는다.
* 이진 세마포(Binary Semaphore) : 이진 세마포에서 S의 값은 0과 1만 가능하다. 만약 두 프로세스 P1,P2에서 P2는 P1이 끝나야만 실행되야 한다면 세마포를 이용하여 구현할 수 있다.

// 프로세스 P1의 구조

P1;

signal(synch);

// 프로세스 P2의 구조

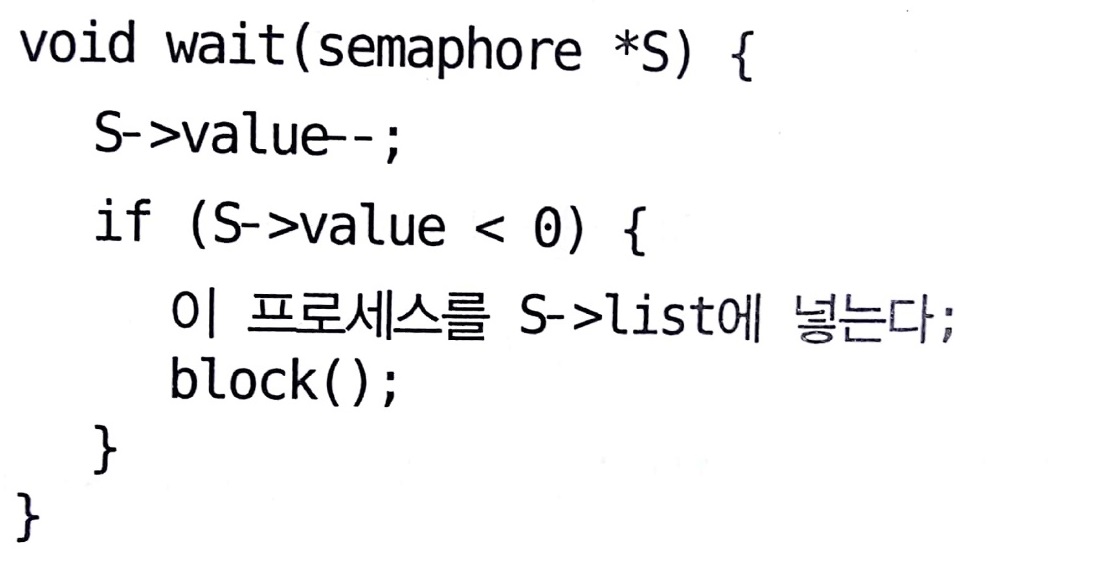
wait(synch);

S2;

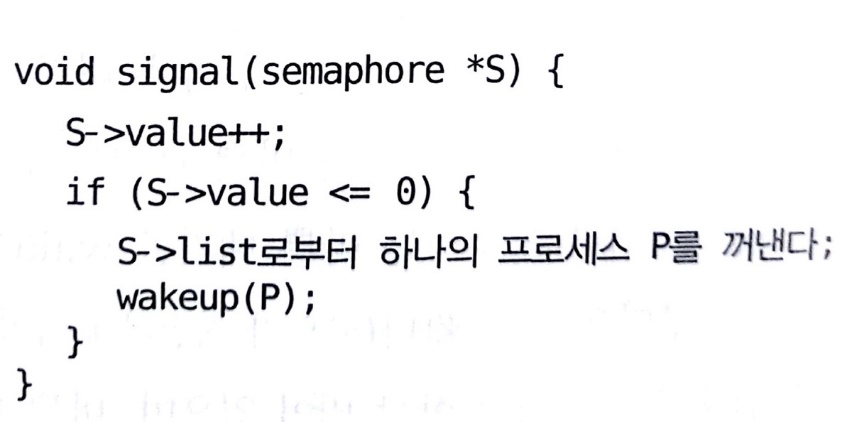
**6.6.2. 구현(Implementation)**

6.5절에서 MUTEX락의 단점을 바쁜 대기라고 언급했었다. 따라서 세마포의 이것을 염두하고 구현해야한다.

**wait()연산 정의**

[](https://github.com/itlockit2/OperatingSystem/blob/master/12.jpg)

**signal()연산 정의**

[](https://github.com/itlockit2/OperatingSystem/blob/master/13.jpg)

wait()연산에 있는 block()메소드는 자기를 호출한 프로세스를 중지시킨다. signal()연산에 있는 wakeup(P) 연산은 봉쇄된 프로세스를 재개시킨다.

**6.6.3. 교착상태와 기아(Deadlock and Starvation)**

만약 P0가 wait(s)를 실행시키고 P1이 wait(Q)를 실행한다고 가정하자.

이러면 P0는 signal(S)를 실행될때 까지 대기를 해야하고 P1은 signal(Q)이 실행될때 까지 대기를 해야한다.

하지만 두 프로세스가 wait 상태이므로 signal() 연산은 실행될 수가없다.

따라서 P0와 P1은 **교착상태(Deadlock)** 가 된다.

**무한 봉쇄(indefinite blocking) 또는 기아(starvation)** 은 세마포에서 프로세스들을 후입선출(LIFO)순서로 제거할 경우 발생할 수 있는 문제점이다.

**6.6.4. 우선순위 역전(Priority Inversion)**

예를 들어 우선순위가 L < M < H 순서인 3개의 프로세스가 존재한다고 가정하자.

프로세스 H는 자원 R을 필요로 하고 이 R자원은 현재 프로세스 L에 의해 접근이 되고 있다.

따라서 프로세스 H는 L이 자원사용을 마칠때 까지 기다려야한다.

그러나 L보다 M이 우선순위가 높으므로 M은 계속 L을 선점하여 간섭하게 된다.

이렇게 H는 우선순위가 제일 높은 프로세스 인데도 우선순위가 낮은 프로세스들에 의해 간섭 받는 현상을 **우선순위 역전(Priority Inversion)** 이라고 한다.

**우선순위 상속 프로토콜(Priority-inheritance protocol)**

우선순위 역전 문제를 해결하기 위한 방법으로 더 높은 우선순위 프로세스가 필요로 하는 자원을 접근 하는 모든 프로세스들은 문제가 된 자원의 사용이 끝날때까지 더 높은 우선순위를 상속받게 되고 자원 사용이 끝나면 원래 우선순위로 돌아간다.

**Example**

아까와 동일한 상황에서 L이 임시적으로 H의 우선순위를 상속받아. 프로세스 M이 L을 선점하는 것을 방지한다. 프로세스 L이 자원 R의 사용을 마치면 상속받은 우선순위를 방출하고 원래의 우선순위로 되돌아간다.

**6.7. 고전적인 동기화 문제들(Classic Problems of Synchronization)**

**6.7.1. 유한 버퍼 문제(The Bounded-Buffer Problem)**

유한 버퍼문제를 해결하기 위해 다음과 같은 자료구조를 인용한다.

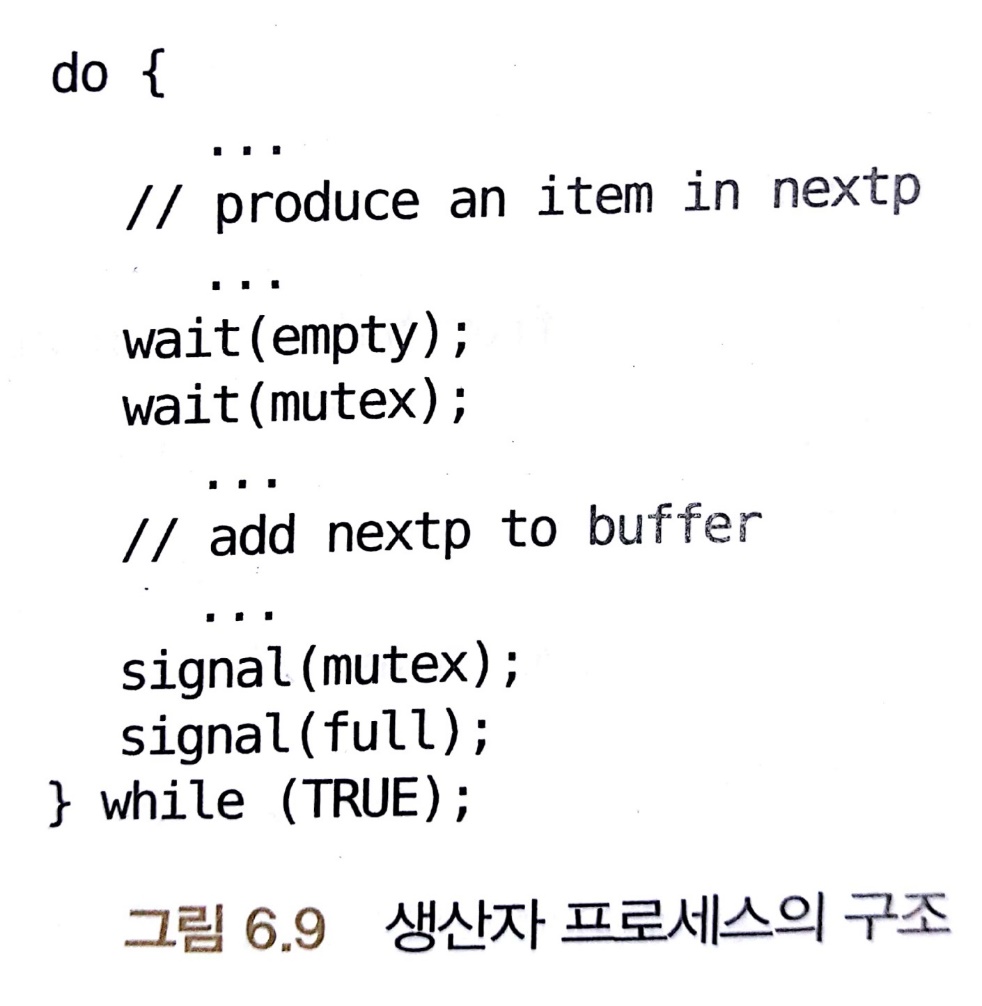
int n;

semaphore mutex = 1;

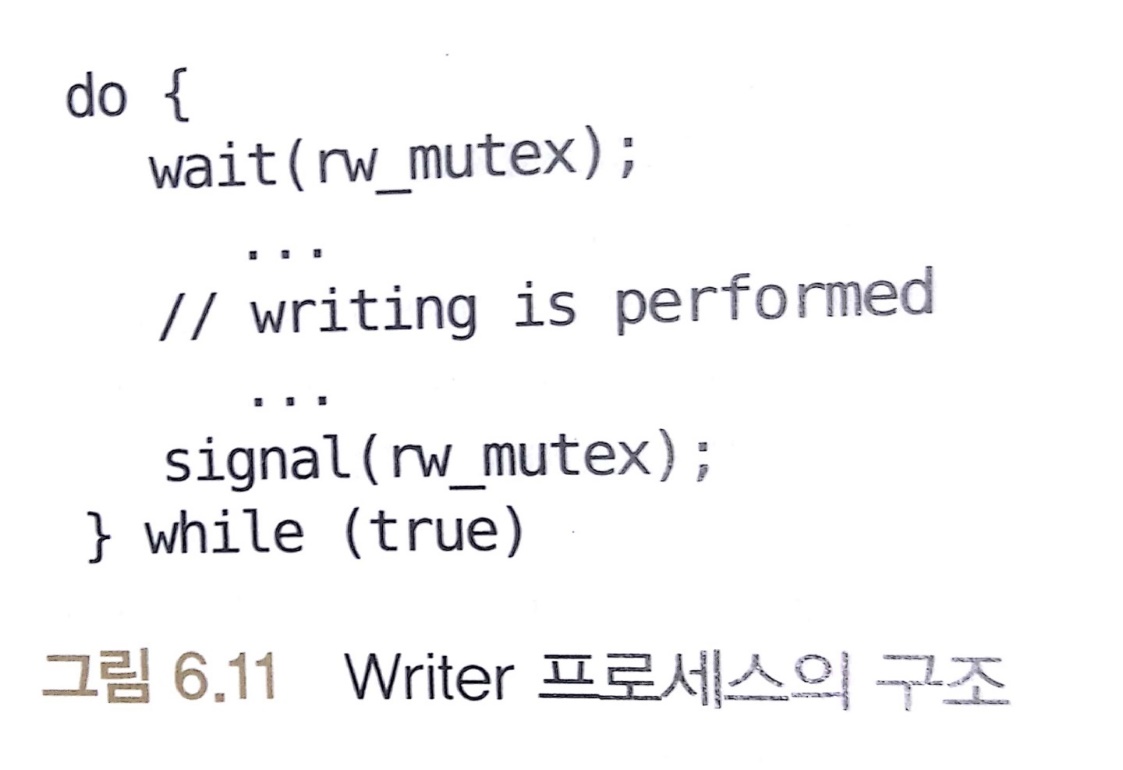
semaphore empty = n;

semaphore full = 0;

**생산자 프로세스의 구조**

[](https://github.com/itlockit2/OperatingSystem/blob/master/14.jpg)

**소비자 프로세스의 구조**

[](https://github.com/itlockit2/OperatingSystem/blob/master/15.jpg)

**6.7.2. Readers-Writers 문제**

하나의 데이터베이스가 다수의 병행 프로세스들 간에 공유된다고 가정하자. 이들 프로세스 들 중의 일부는 데이터베이스의 내용을 읽기만 하고 어떤 프로세스들은 데이터베이스를 갱신(쓰기)를 할 수 있다. 이때 Reader의 경우 동시접근을 해도 문제가 발생하지 않지만 Writer의 경우에는 많은 문제를 야기할수 있다.

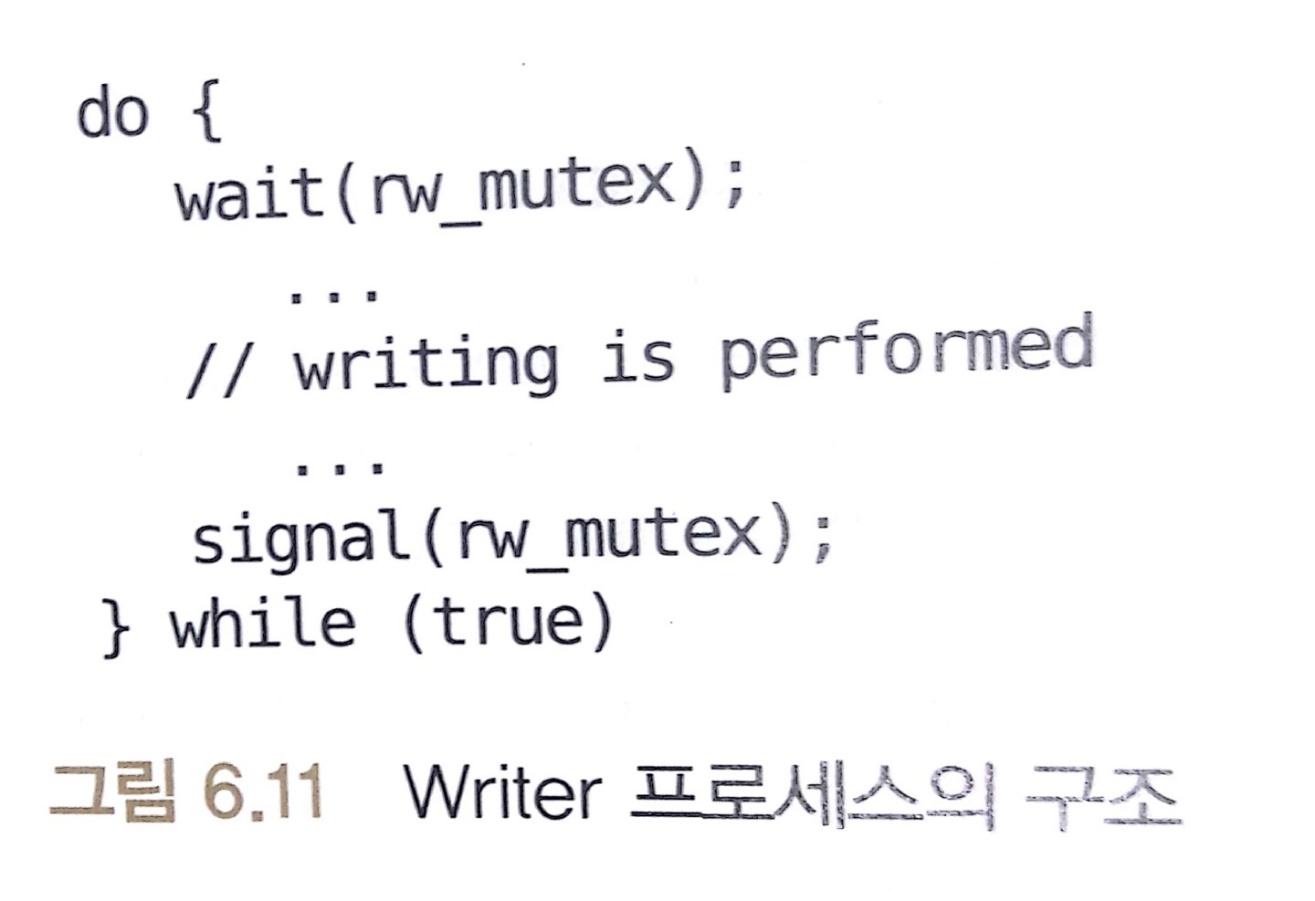
이문제를 해결하기 위해 다음과 같은 자료구조를 인용한다.

semaphore rw\_mutex = 1;

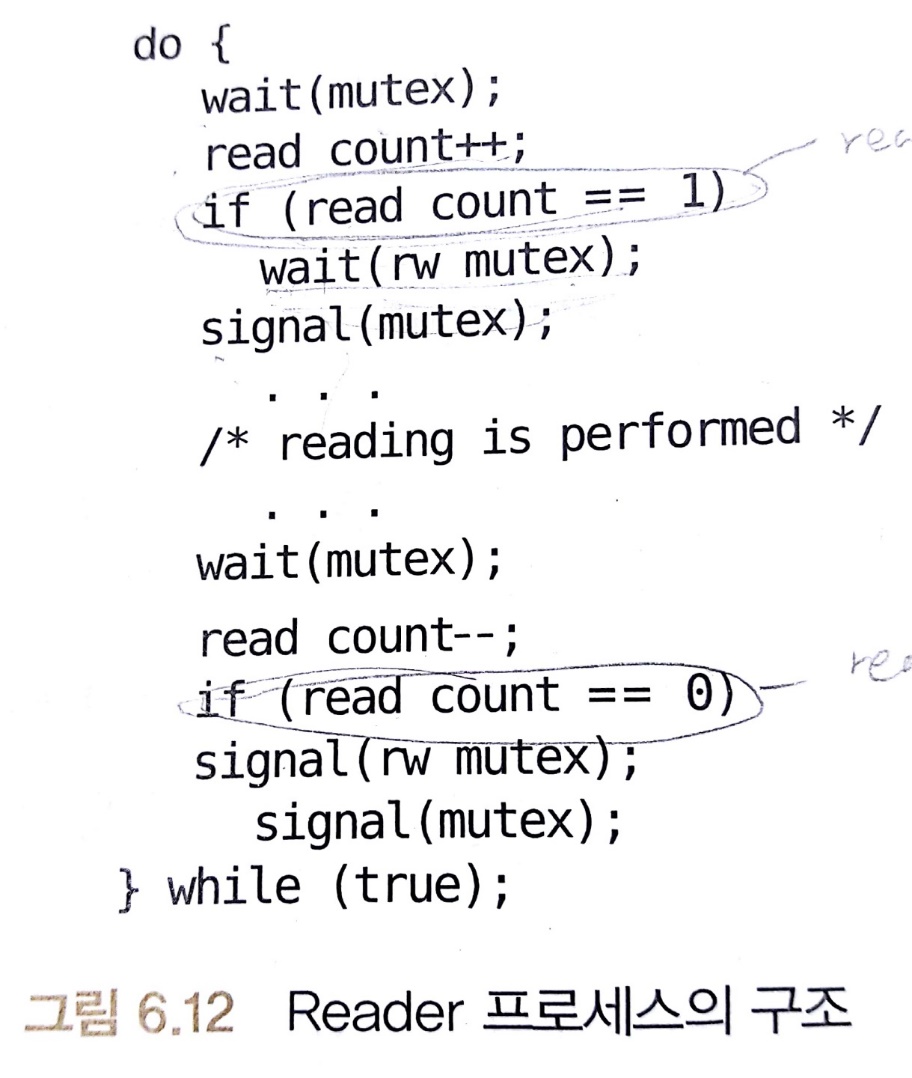
semaphore mutex = 1;

int read\_count = 0; // 현재 몇개의 프로세스들이 객체를 읽고 있는지 알려주는 변수

**Writer 프로세스의 구조**

[](https://github.com/itlockit2/OperatingSystem/blob/master/15.jpg)

**Reader 프로세스의 구조**

[](https://github.com/itlockit2/OperatingSystem/blob/master/16.jpg)

* writer가 임계구역에 있고 n개의 reader들이 기다리고 있으면 한개의 reader만이 rw\_mutex와 관련되어 대기하고 있고 n-1개의 reader 들은 mutex와 관련하여 대기하고 있다.
* 따라서 signal(rw\_mutex)를 수행하면 대기 중인 여러 Reader들이 실행될지 또 다른 Writer가 실행될지는 스케쥴러가 정한다.

**6.7.3. 식사하는 철학자 문제(The Dining-Philosophers Problem)**

5명의 철학자가 원형테이블에서 식사를 하고 있고 테이블에는 5개의 젓가락이 놓여있다.

때때로 철학자들은 자신에게 가장 가까이 있는 2개의 젓가락(왼쪽과 오른쪽)을 집으려고 시도한다.

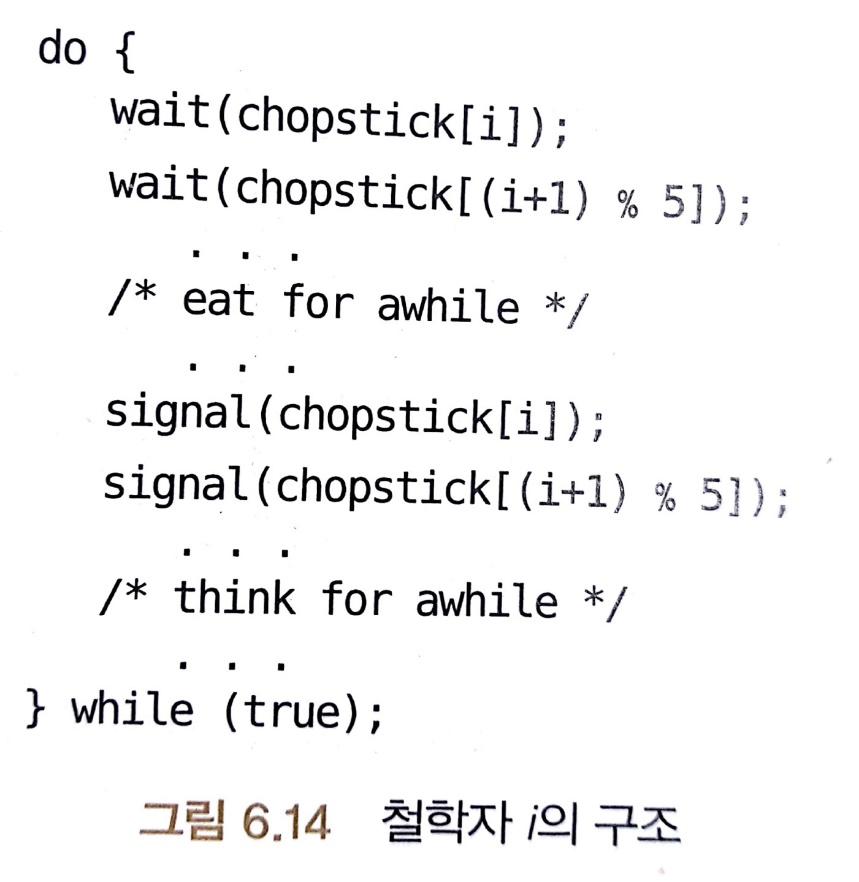
만약 철학자가 동시에 젓가락 2개를 집으면 젓가락을 놓지 않고 식사를 한다.

식사를 마치면 젓가락 2개를 모두 놓고 다시 생각하기 시작한다.

**식사하는 철학자의 상황**

[](https://github.com/itlockit2/OperatingSystem/blob/master/17.jpg)

**철학자 i의 구조**

[](https://github.com/itlockit2/OperatingSystem/blob/master/18.jpg)

**문제점**

이 해결안은 인접한 두 철학자가 동시에 식사하지 않는다는 것을 보장하지만, 교착 상태를 야기할 가능성이 있다.

예를 들어 5명의 철학자가 모두 동시에 배가 고프게 되어 각각 자신의 왼쪽 젓가락을 잡는다고 가정해보자.

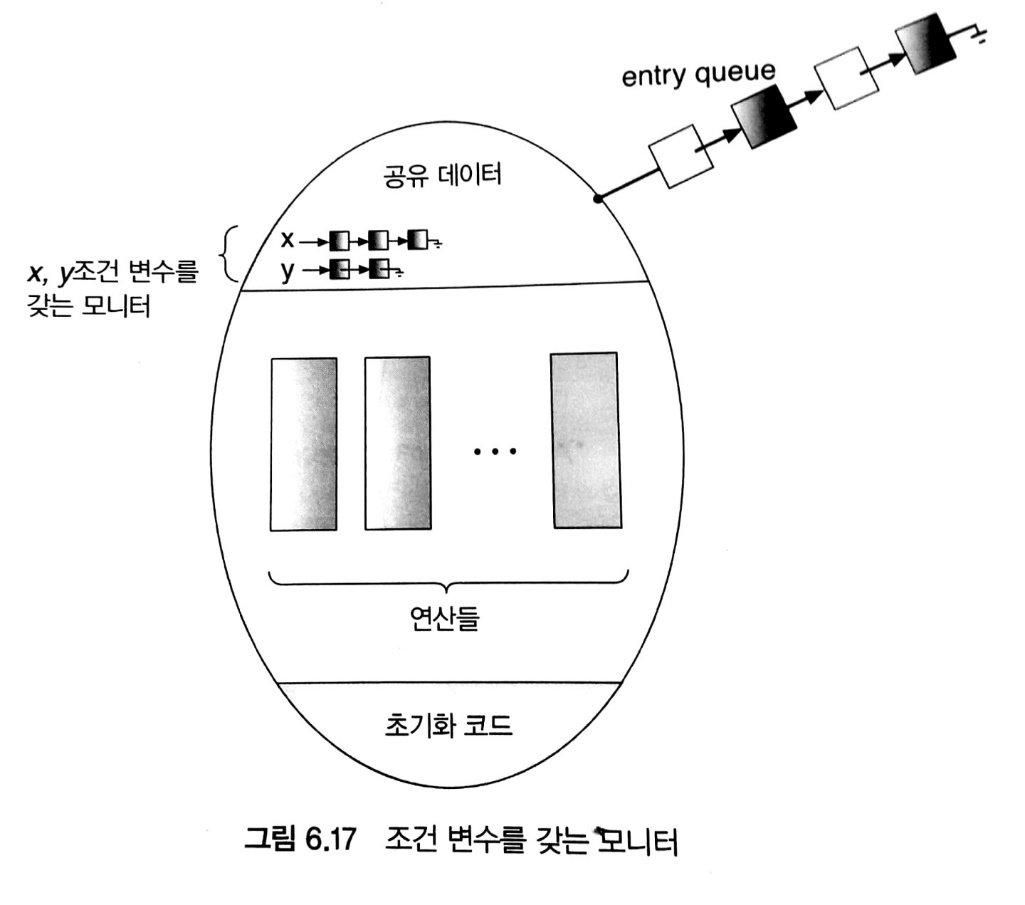
chopstick의 모든 원소들은 이제 0이되고 각 철학자가 오른쪽 젓가락을 집으려고 하면 영원히 기다려야 한다.

**6.8. 모니터(Monitors)**

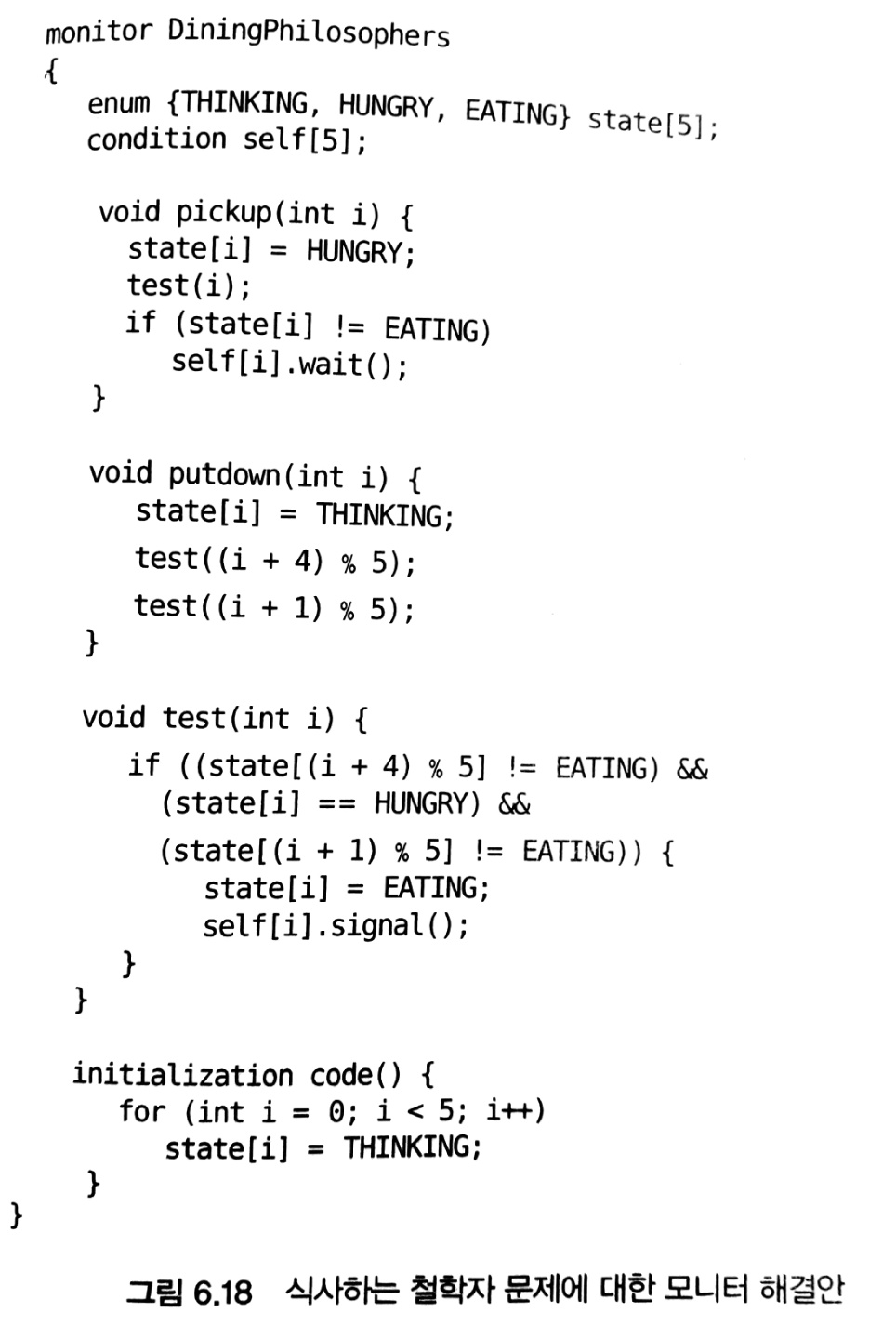
세마포를 이용하여 임계구역 문제를 해결 할 수 있지만 프로그래머가 세마포를 잘못 사용하면 다양한 유형의 오류가 너무나도 쉽게 발생할 수 있다. 따라서 고급 언어 구조물 중 하나인 **모니터(Monitor)형** 에 대해 설명한다.

**6.8.1 모니터 사용법(Usage)**

**추상화 된 데이터 형(abstract data type, ADT)** 은 데이터와 이 데이터를 조작하는 함수들의 집합을 하나의 단위로 묶어 보호한다.

[](https://github.com/itlockit2/OperatingSystem/blob/master/19.jpg)

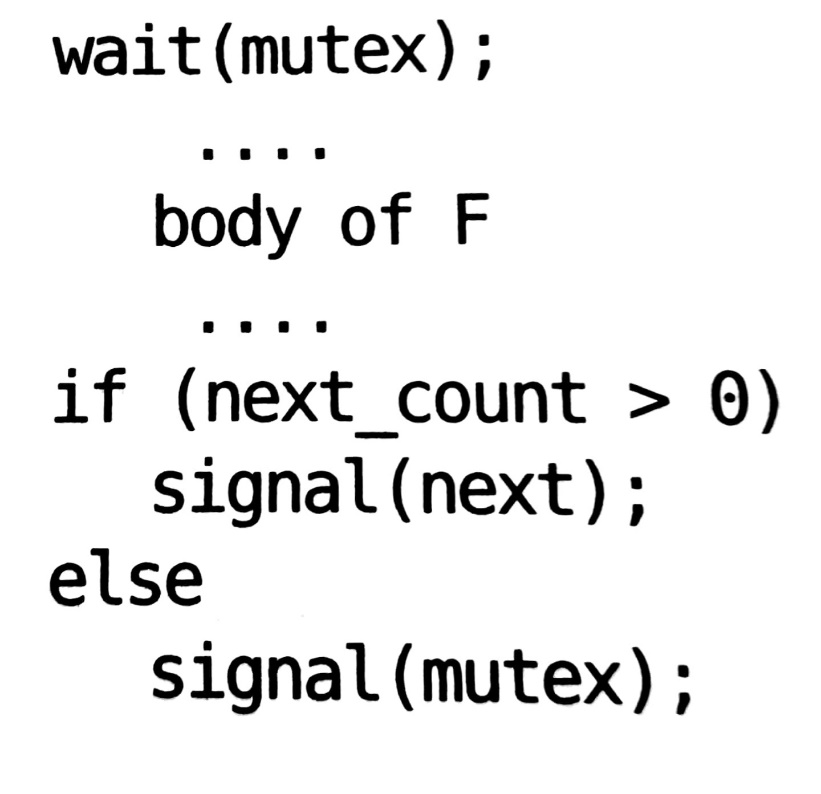
**6.8.2. 모니터를 사용한 식사하는 철학자 해결안**

[](https://github.com/itlockit2/OperatingSystem/blob/master/20.jpg)

* 철학자 i는 그의 양쪽 두 이웃이 식사하지 않을 때만 변수 state[i] = eating으로 설정할수 있다.
* 철학자 i가 젓가락을 내려 놓을때 양옆의 철학자들이 eating 상태가 될수 있는지 test한다.

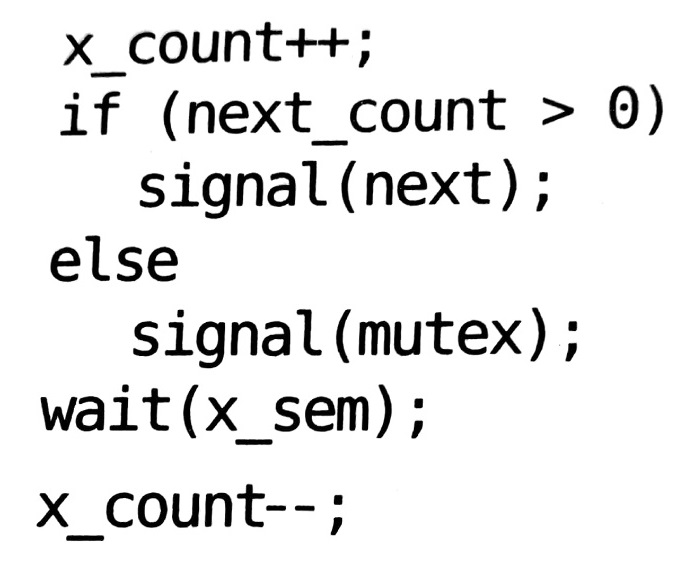
**6.8.3. 세마포를 이용한 모니터의 구현**

* 각 모니터 마나 mutex라는 세마포가 정의되고 초기값은 1이다.
* 프로세스는 모니터로 들어가기 전에 wait(mutex)를 실행하고 모니터를 나온 후에 signal(mutex)를 실행해야한다.

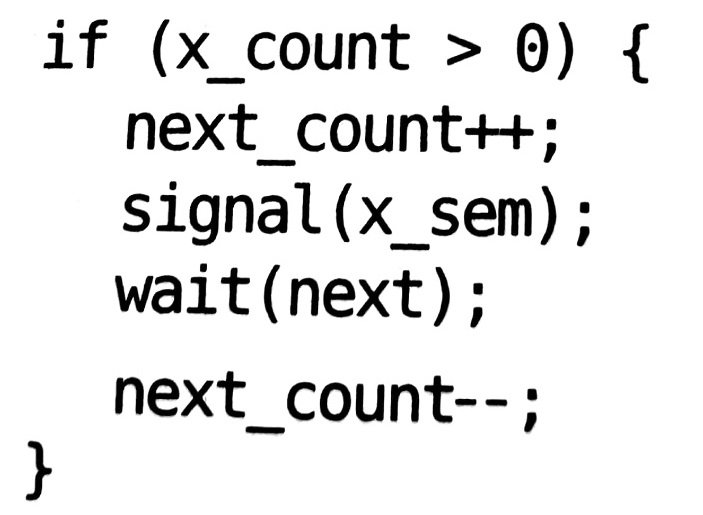
[](https://github.com/itlockit2/OperatingSystem/blob/master/21.jpg)

* next\_count는 next에서 일시중단 되는 프로세스의 개수를 의미한다.

**x.wait() 연산 구현**

[](https://github.com/itlockit2/OperatingSystem/blob/master/22.jpg)

**x.signal() 연산 구현**

[](https://github.com/itlockit2/OperatingSystem/blob/master/23.jpg)

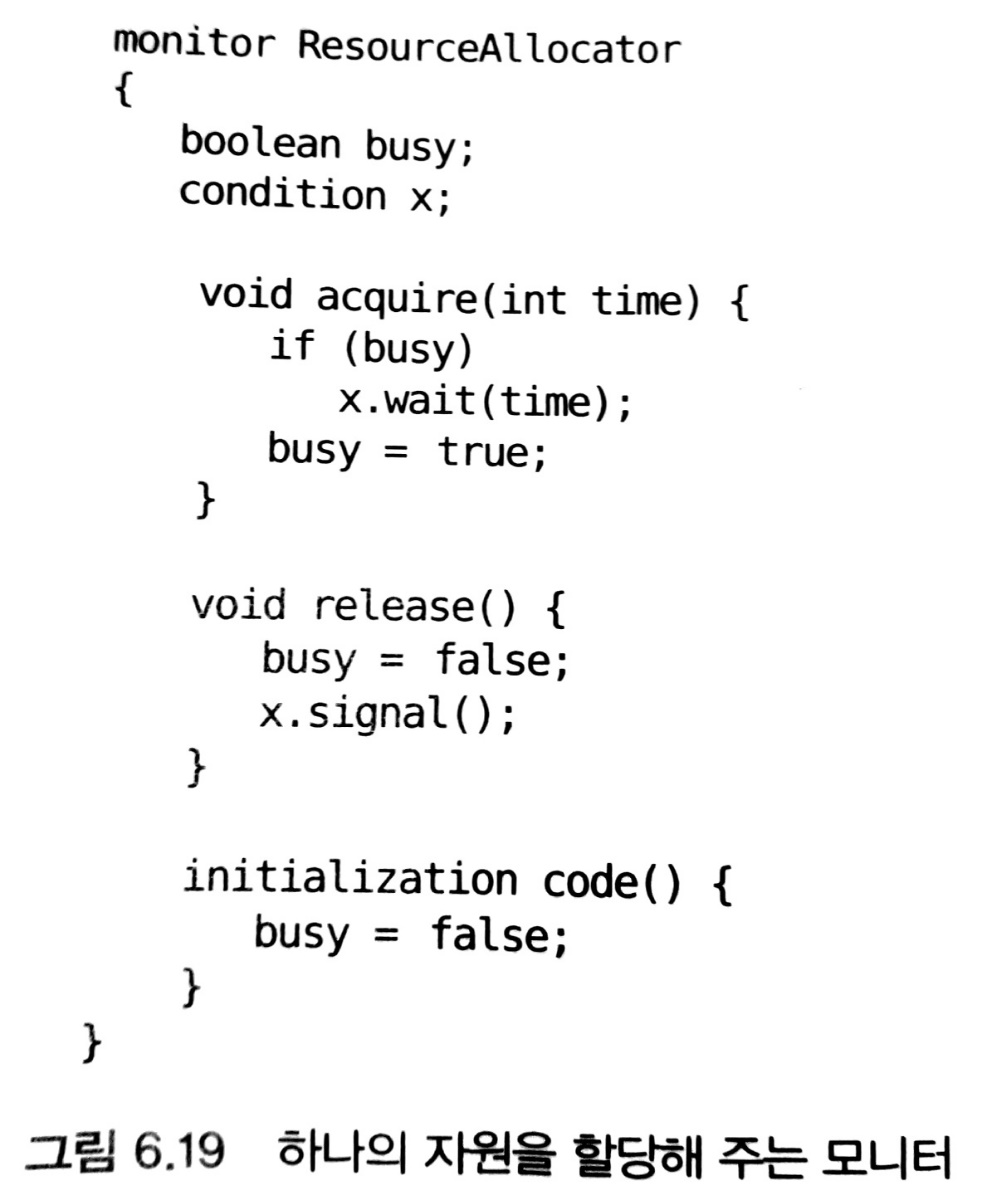
**6.8.4. 모니터 내에서 프로세스 수행재개**

조건 변수 x에 여러 프로세스들이 일시중단 되어 있고 어떤 프로세스가 x.signal() 연산을 수행했다면, 일시중단 되었던 프로세스들 중 어느 프로세스가 수행 재개 될 것인가?? 이를 위해서 **Conditional-wait** 구조물을 사용할 수 있다.

* conditional-wait 형태 \* x.wait(c);

여기서 c의 값은 **우선순위 번호(Priority number)** 라고 불리며 일시중단 되는 프로세스의 이름과 함께 저장된다.

x.signal()이 수행되면 가장 작은 우선순위 번호를 가진 프로세스가 다음번에 수행재개 된다.

[](https://github.com/itlockit2/OperatingSystem/blob/master/24.jpg)

**6.9. 동기화 사례(Synchronization Examples)**

**6.9.1. Windows의 동기화(Synchronization in Windows)**

* dispatcher 객체 : 스레드는 dispatcher 객체를 사용하여 mutex락, 세마포, event 타이머 등등을 이용하여 동기화 한다.
* Event : 조건변수와 같은 역할로 기다리는 조건이 만족되면 기다리고 있는 스레드에게 통지해줄 수 있다.
* signaled 상태 : dispatcher 객체를 사용 가능하고 그 객체를 얻을 때 스레드가 봉쇄되지 않음을 뜻한다.
* nonsignaled 상태 : dispatcher 객체를 사용 가능하지 않고 그 객체를 얻으려고 시도하면 그 스레드가 봉쇄됨
* critical-section 객체 : 커널의 개입 없이 획득하거나 방출할 수 있는 사용자 모드 mutex이다.

**6.9.2. 리눅스의 동기화(Synchronization in Linux)**

Linux는 커널 선점을 제어하는데 특이한 방식을 사용한다.

* preempt\_disable()과 preempt\_enable() \* 커널에서 실행 중인 테스크가 락을 소유하고 있을 경우에는 커널은 선점 가능하지 않다.

**6.9.3. Solaris의 동기화**

* 적응적 Mutex : 모든 임계 데이터 항목에 대한 접근을 보호한다.
* tunrstile : 락 때문에 봉쇄된 스레드들을 수용하는 큐 구조이다. turnstile는 우선순위 역전 현상을 방지하기 위해 우선순위 상속 프로토콜을 구성한다.

**6.9.4. Pthreads 동기화**

* 기명(named)과 무기명(un-named) 세마포를 제공한다. 기명 세마포는 파일 시스템 안에 실제 이름을 가지고 있어서 여러 관계없는 프로세스들이 공유할 수 있지만 무기명 세마포는 같은 프로세스에 속한 스레드에 의해서만 사용 가능하다.

**6.10. 대체 방안들**

**6.10.1. 트랜잭션 메모리(Transactional Memory)**

트랜잭션 메모리의 개념은 데이터베이스 이론분야에서 출발한 아이디어이다.

한 트랜잭션의 모든 연산이 완수되면 메모리 트랜잭션은 **확정(commit)** 된다. 만약 그렇지 않다면

그 시점까지 완수된 모든 연산들은 취소되고 트랜잭션 시작 이전의 상태로 **되돌려야(roll-back)** 한다.

* 소프트웨어 트랜잭션 메모리(STM) : 특별한 하드웨어 필요없이 소프트웨어만으로 구현 됨. 트랜잭션 블록안에 검사 코드를 삽입함으로써 동작.
* 하드웨어 트랜잭션 메모리(HTM) : 개별 처리기 캐시에 존재하는 공유 데이터의 충돌을 해결하고 관리하기 위하여 하드웨어 캐시 계층 구조와 캐시 일관성 프로토콜을 사용한다.

**6.10.2. OpenMP**

OpenMP는 **#pragma ompciritical이라는 디렉티브** 를 제공한다. 이 디렉티브는 디렉티브 이후에 나오는 코드 구역을 임계구역으로 지정하여 한 번에 하나의 스레드만이 실행할 수 있게 한다.

**6.10.3. 함수형 프로그래밍 언어**

함수형 언어는 변수가 정의되어 값을 배정받으면 그 값은 변경될 수 없다. 따라서 함수형 언어는 변경 가능 상태를 허용하지 않기 때문에 경쟁 조건이나 교착상태와 같은 쟁점에 대해 신경쓸 필요가 없다.