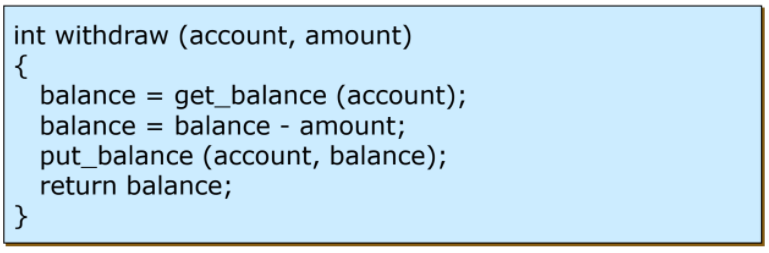
**Ch6) Synchronization**

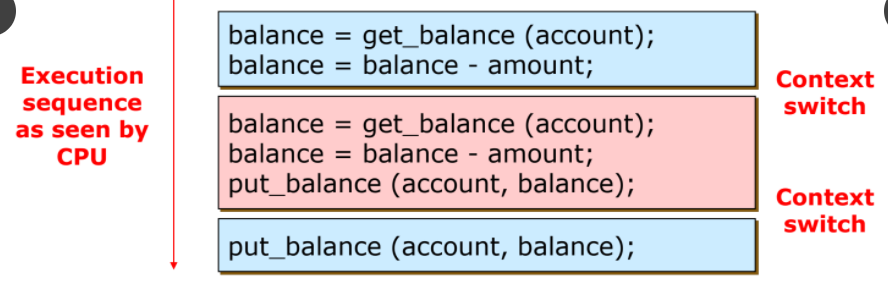
1. Threads cooperate in multithreaded programs

* 쓰레드는 멀티 쓰레드 환경에서 서로의 자원을 공유한다.
* 동기화가 필요하다.

1. 우선 동기화가 왜 필요한지에 대한 간단한 예제를 살펴보자



* 은행에서 계좌 인출에 관한 예제이다. 당신과 당신의 여자친구가 커플 계좌를 만들어서 100만원을 입금해 놓았다고 가정해보자.
* 만약 두 사람이 각각 은행에 가서 거의 동시에 10만원을 인출했다면 어떤 일이 일어날까?



* 상황은 이렇다. 남자 친구 요청에 의한 Process P1과 여자친구 요청에 의한 Process P2가 생성되었다. 첫번째 파란 박스와 같이 P1이 수행되다가 context switch가 일어나서 빨간 박스의 P2가 수행을 시작했다. 그런데 다시 context switch가 일어나서 P1마저 수행되었다.
* 계좌에 얼마가 남아있을까? -> 원래는 80만원이 되어야 맞지만 90만원이 입력되었다.
* P1 프로세스가 수행되다가 잔액이 저장되지 않은 채로 context switching이 일어났기 때문에 P2 프로세스가 수행을 시작할 때의 잔고는 90만원이 아니고 100만원이다. 따라서 동기화 문제가 발생했다.

1. Synchronization Problem

* 두 개의 concurrent threads(or process)가 synchronization없이 공유 자원을 서로 접근하려 했기 때문에 문제 발생
* 이러한 상태를 Race Condition 이라고 한다. Race condition일 때의 결과는 non - deterministic 하다고 한다.(deterministic: 항상 같은 결과, non-deterministic: 항상 같은 결과가 아님, 때에 따라 달라짐)
* Synchronization은 shared data structure에서 필수이다.

1. Critical-Section problem

* 앞서 예제에서 살펴 본 withdraw와 같이 Race condition을 유발하는 코드 level에서의 구간, 영역을 말한다.
* 각각의 process는 shared data에 접근하는 critical section의 code segment를 가지고 있다.
* 이후에 한 개의 process가 critical section에 있을 때는 다른 process가 critical section에 접근하지 못하도록 보장하는 방법으로 이 문제를 해결한다.

1. Solution to Critical-Section Problem
2. Mutual Exclusion

* 상호 배제, 나만 사용
* P1 process가 critical section을 실행하고 있을 때, 다른 어떤 process도 critical section에 진입할 수 없다.

1. Progress

* Critical section을 아무도 안 쓰고 있으면 내가 사용
* 만약 어떤 process도 critical section을 수행하고 있지 않고 critical section에 진입하고자 하는 process가 있으면, 그 process를 수행해야한다.

1. Bounded waiting

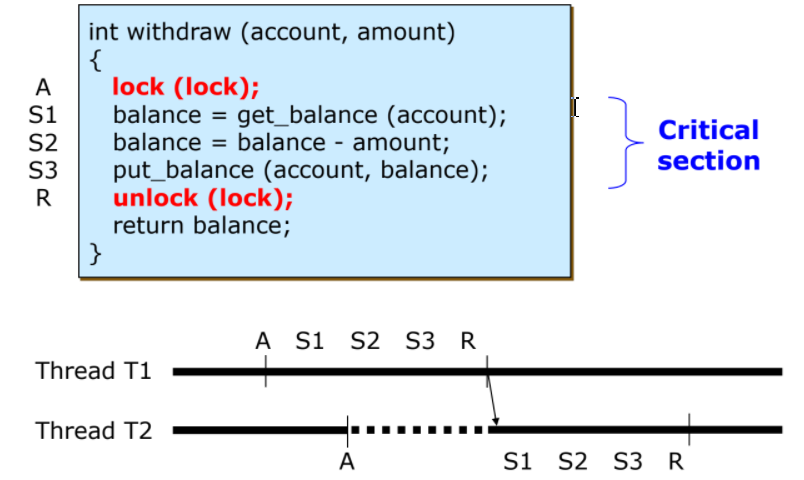
* Critical section에 들어가야할 process는 무한히 대기하고 있으면 안된다. Bounded time이 있어야 한다.

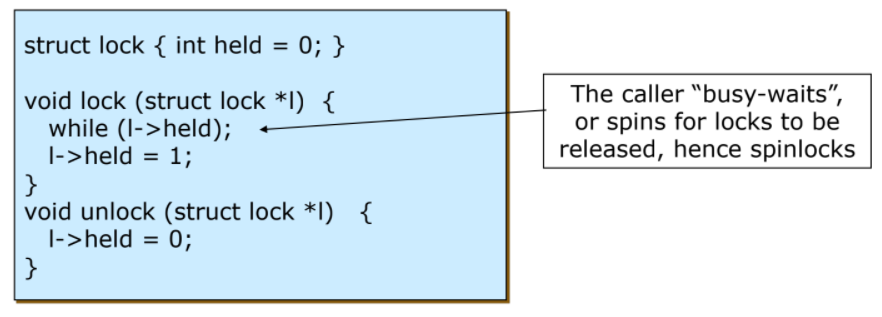
위 3가지 조건을 만족해야 critical section problem을 해결할 수 있다.

1. Mechanisms for critical sections

* 크게 어떤 방식이 있는지 살펴보자

1. Locks(잠금)
   * 가장 원초적이고 간단하고 쉬운 방법임
   * 하지만 근본적으로 critical section problem을 해결하지 못한다.
   * 아래 2, 3, 4는 lock에 대한 solution임
2. Software-only algorithm
   * Peterson algorithms for two processes
   * Bakery algorithms for n processes
   * 오버헤드가 크게 발생, 현재 운영체제 사용x
3. Hardware atomic instruction
   * Test-and-set
   * Swap
   * 하나의 명령어로 Lock, unlock을 구현
4. Disable interrupt
   * To prevent context switches
   * 문제의 원인인 Interrupt를 disable시킴
5. Semaphores(higher-level synchronization)
   * Semaphore 변수 사용
6. Monitors(higher-level synchronization)
   * High-level
   * Signal, waiting을 통한 방법
7. Locks

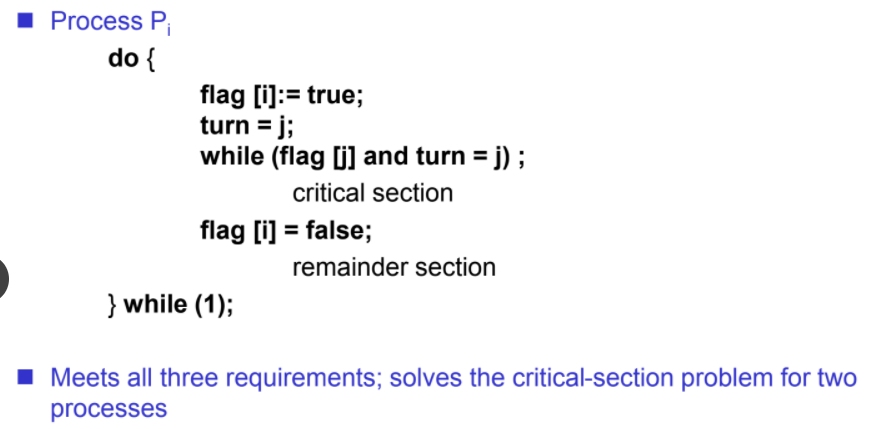




* Lock은 객체이고 lock(), unlock() 함수를 제공한다. Lock()은 lock이 free가 될때까지 붙잡고 wait시키는 함수이다. Unlock()은 lock()상태에서 waiting하고 있는 thread를 unlock하는 함수이다.
* 처음 lock은 초기화되고 Critical section에 진입 직전 lock()함수를 호출하고 빠져나올 때 unlock()함수를 호출한다.
* Lock()과 unlock()사이에 thread는 lock을 유지한다.
* While부분에서 cpu가 무한루프를 돌기위해 사용되기 때문에 busy-waiting, spinlock 이라고 불린다.
* \*\* Does it work? -> while부분에서 timer interrupt가 일어난다면? 또 다른 무한루프 발생

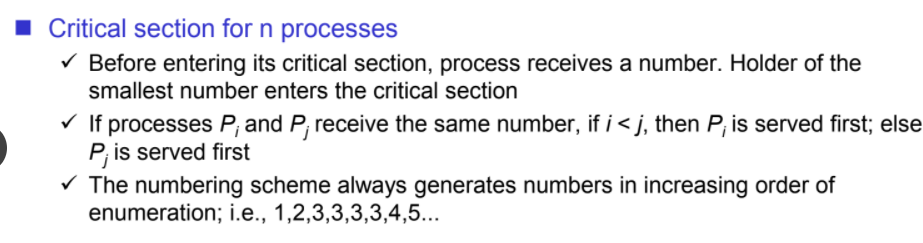
1. Software only algorithms
2. Peterson algorithms

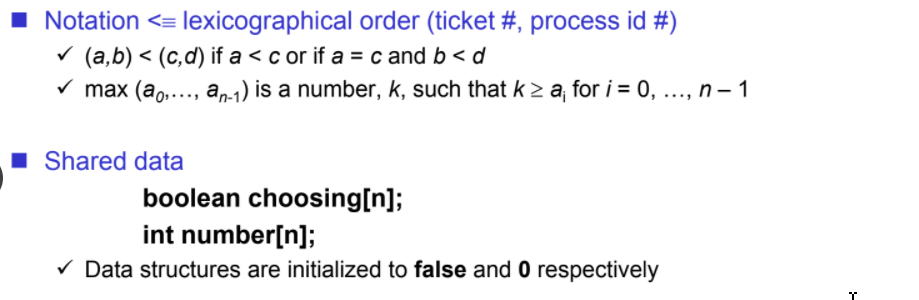
* 교재에 앞서 나온 algorithms 1과 algorithms 2는 mutual exclusion은 만족시키지만 progress는 만족시키지 못한다. 두 가지를 보안해서 나온 것이 이 피터슨 알고리즘이다. 피터슨 알고리즘은 3가지 조건을 모두 만족시킨다.(mutual exclusion, progress, bounded waiting)

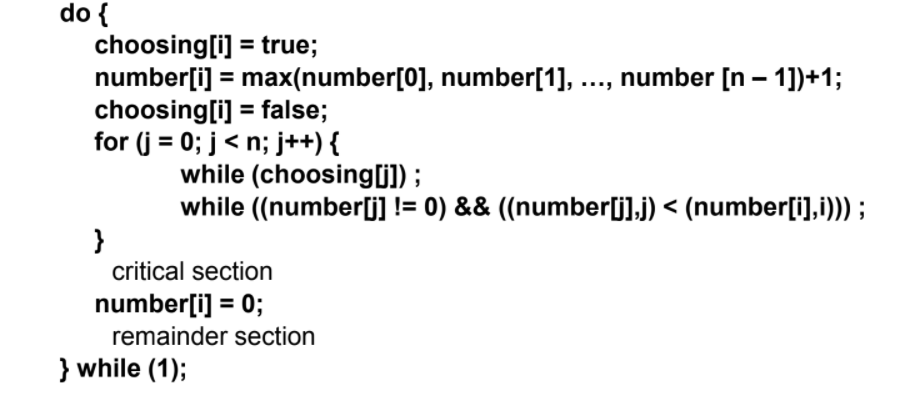


1. Bakery algorithms

* 앞서 살펴본 알고리즘은 두 개의 process를 위한 알고리즘이다. 베이커리 알고리즘은 n개의 process를 해결하기 위한 알고리즘이다.

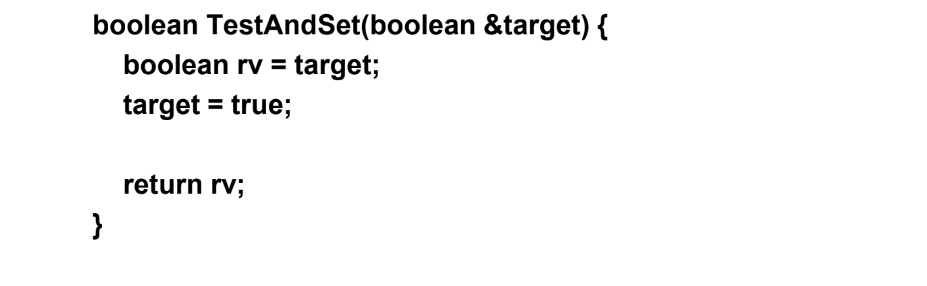


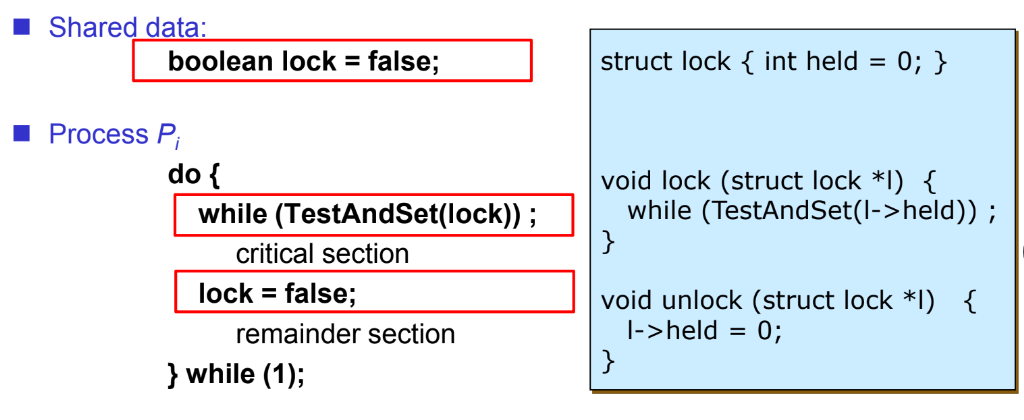




* Critical section에 진입하기 전에 각각의 process는 번호표를 부여 받는다. 그리고 가장 작은 번호표를 부여 받은 process가 critical section에 진입하게 된다.
* 만약 두개의 process가 부여 받은 번호가 같다면 PID갚으로 순서를 지정한다.
* Boolean choosing[n]은 번호표를 부여 받을 준비가 되어 있다는 것을 나타낸다.(peterson’s algorithms의 flag와 유사), int number[n]은 부여 받은 번호.
* 초기값은 choosing, number 각각 false와 0이다.
* 3가지 조건을 모두 만족시킨다.(mutual exclusion, progress, bounded waiting)

1. Hardware atomic instruction
2. Test-and-set



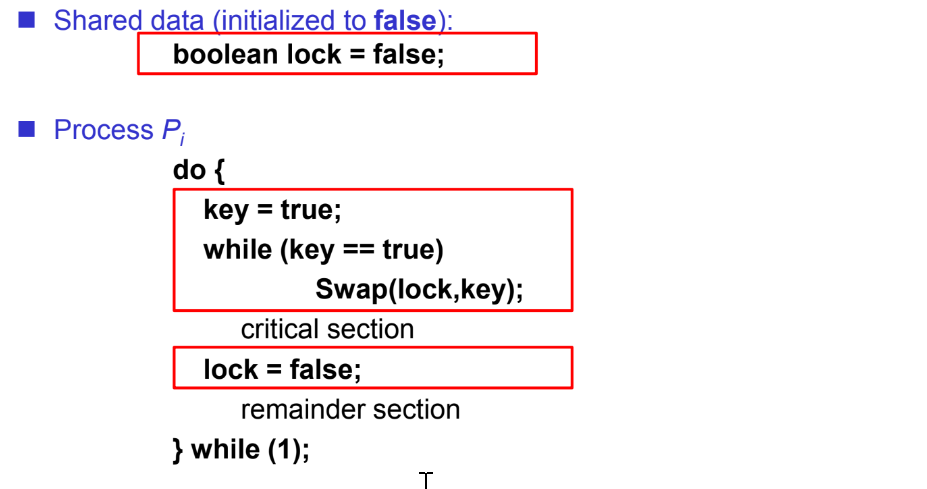


* Test-and-set은 bounded waiting을 만족시키지 못한다.

1. Swap

텍스트이(가) 표시된 사진

자동 생성된 설명



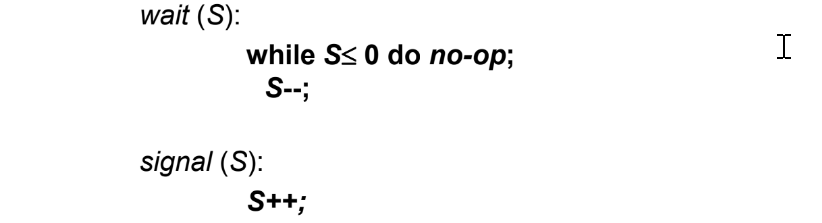
1. Disable interrupt

* 언뜻 보면 context switch가 critical section문제를 일으키는 근본적인 원인이기 때문에 disable interrupt가 합당한 방법이라고 생각할 수 있지만 그렇지 않다. 이 방법은 커널에서만 처리가 가능하고, 각 process마다 인터럽트를 걸지 말라고 일일이 얘기를 해줘야 한다. 따라서 오버헤드가 매우 크고 비효율적인 방법이라 현대 운영체제에서 사용하지 않는 방법이다.

1. Higher-level synchronization

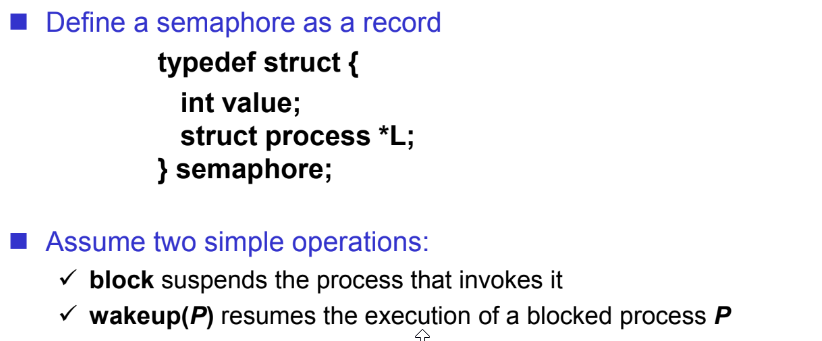
* User-mode에서 사용하는 synchronization

1. Semaphore



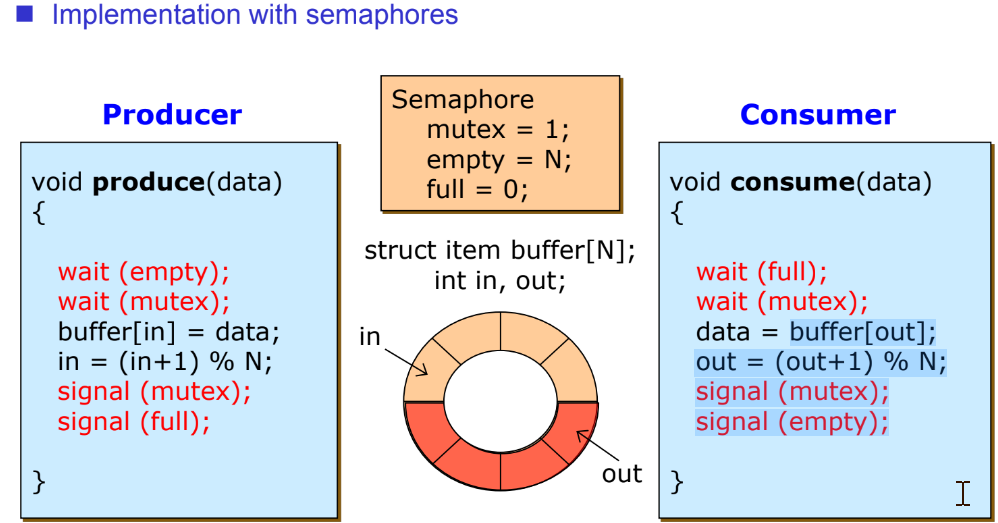
텍스트이(가) 표시된 사진

자동 생성된 설명

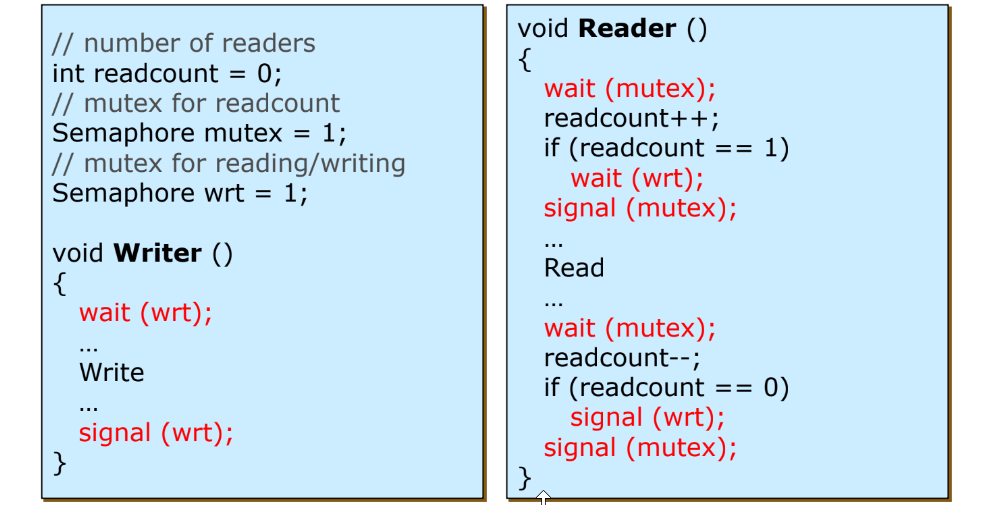


* 이러한 semaphore를 통해 해결할 수 있는 3가지 classical problems을 살펴보자

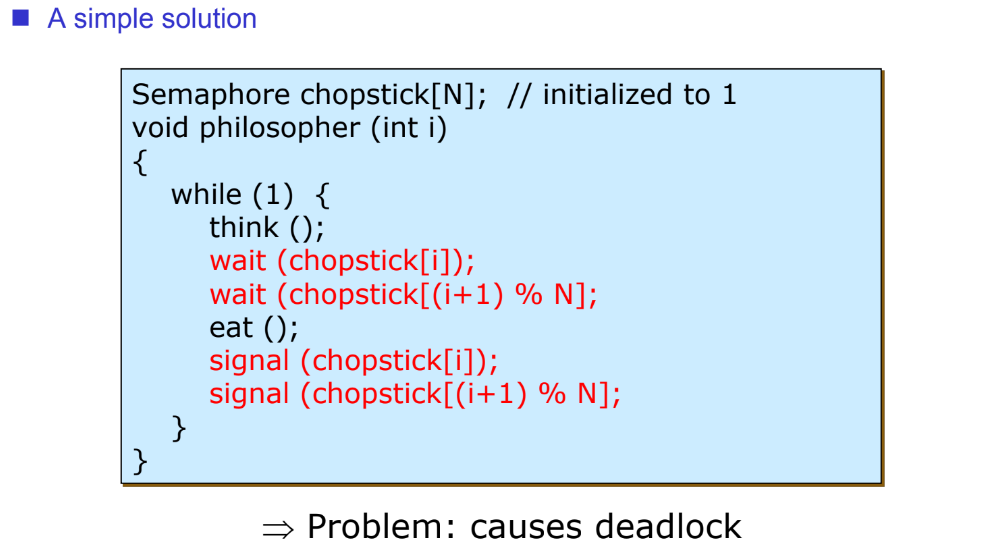
1. Bounded-Buffer Problem(Producer, Consumer Problem)

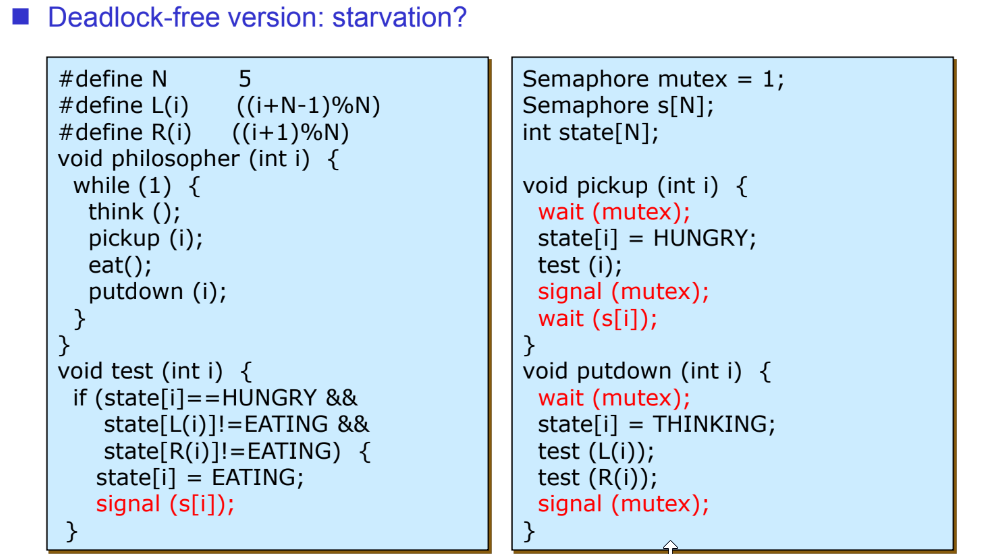


1. Readers and Writers Problem
   * + Writer process가 들어와 있을 때는 또 다른 writer process나 reader process가 들어와도 실행이 안되지만 reader process가 들어와 있을 때는 writer process가 들어오면 실행이 안되지만 또 다른 reader process는 수행이 가능하다.



1. Dining -Philosophers Problem





* Semaphore변수를 mutex와 철학자배열 s[N] 두가지를 설정한다.

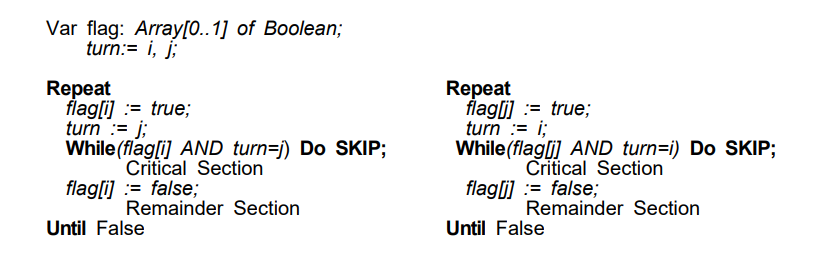
1. Monitor

\*\*\* 예상 문제

1. What is the critical-section problem? describe the detailed condition and solution for critical section

* Critical section이란 Race condition을 유발하는 코드 level에서의 구간을 이야기 한다. 그리고 두 개 이상의 process나 thread가 동기화 없이 shared data에 접근하려고 할 때 생기는 이러한 상태를 race condition이라고 한다. 이러한 critical section 문제를 해결하기 위해선 다음과 같은 3가지 solution(조건)이 필요하다.

1. Mutual exclusion: 한 개의 process가 critical section을 수행하고 있다면 다른 process는 critical section에 진입하지 못하도록 하는 것이다.(나만 사용)
2. Progress: Critical section을 수행중인 process가 없고 진입하고자 하는 Process가 있다면 그 process를 수행시켜야 한다는 것이다(아무도 안 쓰면 내가 사용). 이 조건은 CPU utilization과 관계가 있다.
3. Bounded waiting: critical section에 들어가고자 하는 process가 무한히 기다리면 안된다. 이 조건은 starvation과 관계가 있다.
4. Describe whether the following software solution to guarantee mutual exclusion is correct or not. If not, please explain why it is incorrect?



* 위 알고리즘은 Peterson 알고리즘이다. Peterson’s 알고리즘은 3가지 조건인 mutual exclusion, progress, bounded waiting을 모두 만족시킨다.

1. Describe why Mutual Exclusion is needed in concurrent program environment. You have to show example

* Concurrent program environment란 여러 개를 수행하는 것처럼 보여도 결국 한번에 한 개의 process만을 수행하는 환경을 말한다. 이렇게 context switch가 일어나면서 두 개 이상의 process(or thread)가 번갈아 가면서 수행될 때 process가 shared data에 접근하면서 즉, 1번에서 이야기한 critical section에 접근하면서 동기화 문제가 발생한다.  
  예를 들면 공동계좌 은행 인출 문제가 있다. A의 요청에 대한 응답 process와 B의 요청에 대한 응답 process가 있다고 하면, 두 process가 동시에 critical section에 진입해서 수행된다면 결과는 non-deterministic하다.(때에 따라 다를 수 있다.) 따라서 concurrent program environment에서 mutual exclusion은 꼭 필요한 조건이다.

1. Describe detailed the semaphore operation of synchronization metho