**OS 중간고사 정리**

시험 출제

Chapter6. Synchronization

멀티 스레드 프로그램에서 공유 메모리 사용 시 문제 발생!

① 공유 메모리에 동시에 접근할 경우

② 정보 저장 전에 호출하는 프로세스가 접근하는 경우

**Race condition**

두 개 이상의 프로세스 또는 스레드가 shared data를 서로 접근하려고 하는 상황

스레드의 순서가 꼬여서 이상한 데이터가 저장된다. (ex A 수행 중 인터럽트->B수행->A마저 수행)

멀티스레드 작업 할 때는 항상 발생한다고 생각할 것.

**Critical Section**

Critical section: race condition을 발생시키는 코드부분

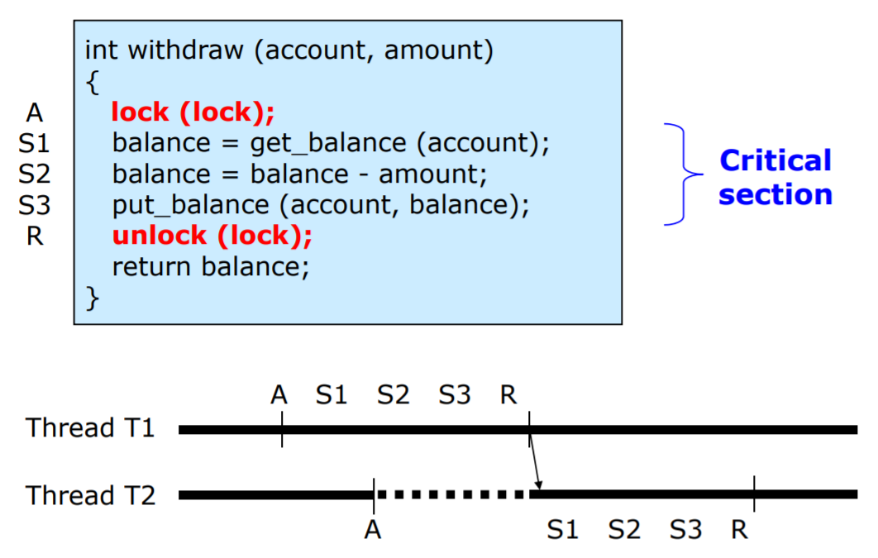
해결 방안 ① Mutual Exclusion: 한 프로세스(스레드)만 critical-section에 접근 가능하다.

② Progress: 아무도 안 쓰고 있을 때 바로 사용한다. (대기X)

③ Bounded waiting: 누군가가 사용하고 있을 때 대기한다.

단, 곧 내 차례가 오는 것이 보장 될 때 사용한다.

**Lock**: critical-section에 들어갈 때 락을 건다. 일종의 코드 사용표시.

T1 A: critical section진입. lock설정

T1 S1~: 코드 진행

T2 A: critical section 진입 시도.

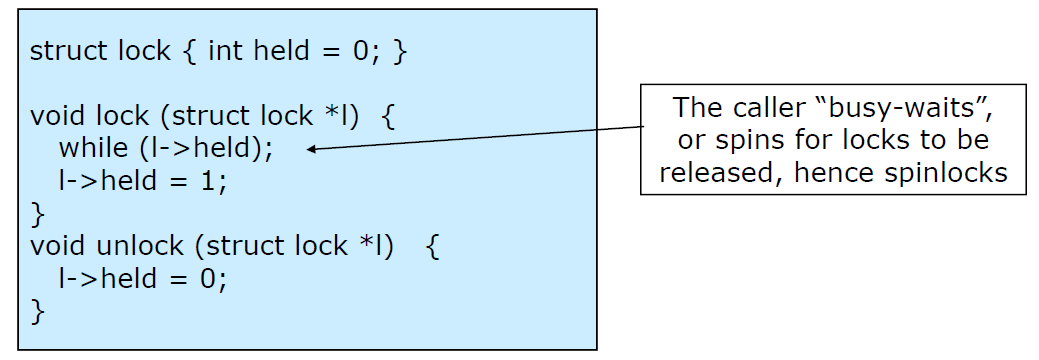
Lock이 걸려있으므로 대기.

T1 R: critical section탈출. lock해제

T2 A¨S1: lock해제 확인.

Lock 설정 후 코드 진행

held가 1일 때 while문을 돌게 된다.



1. A프로세스 접근: held = 0

Lock: held를 1로 만든다.

2. B프로세스 접근: held = 1

무한루프 돌다가 타임퀀텀 종료

3. A프로세스 실행: critical section 종료

Unlock: held를 0으로 만든다.

4. B프로세스 실행: held = 0

무한 루프 탈출 후 lock: held를 1로 만든다.

Spin lock: 무한 루프를 돌면서 unlock이 되기를 기다림

* Busy-waits: CPU를 계속 사용하면서 대기. CPU를 낭비하여 비효율적이다.

**Lock에서 발생하는 critical section**

Atomic operation: 더 이상 쪼갤 수 없는 함수. Held를 조작하는 과정에서도 interrupt가 발생하면

Critical section 문제가 발생하기 때문에, Atomic operation으로 구현해야 한다.

해결 방안 ① Software-only algorithms: 알고리즘을 통해 해결. but 오버헤드가 크기 때문에 사용X

② Hardware atomic instruction: CPU가 atomic instruction을 제공. (TestAndSet)

명령어 하나이기 때문에 인터럽트가 발생할 수 없다.

Critical section이 짧을 때만 OS가 사용하는 방법.

③ Disable/re-enable interrupt: lock을 거는 순간 인터럽트를 발생하지 못하게 하는 것.

OS가 사용하는 방법. CPU가 1개 있을 때만 가능하다.

**Semaphore**: 사용할 수 있는 shared data의 개수를 나타냄. (Integer 변수)

사용할 때 Wait() 호출: S-1, 완료 후 Signal() 호출: S+1.

대기하는 프로세스는 Waiting queue에 저장해서 busy-wait을 방지한다.

T1 S=1 T2

Wait(S) Wait(S)

1. T1이 Critical Section에 접근 Wait(S) 실행: S=0

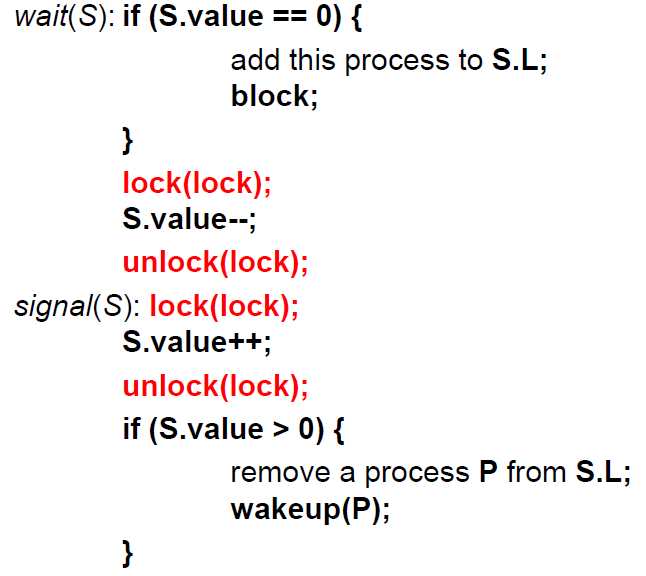
2. T2가 Critical Section에 접근 Wait(S) 실행: S=0이므로 Waiting

sync

3. T1 실행 종료 시 Signal(S) 실행: S=1

S=1이므로, Waiting상태인 T2를 Ready상태로 돌리고 실행!

Signal(S) Signal(S)

Semaphore 자체도 shard data이기 때문에 Wait()와 Signal()을 atomic하게 구현하는 것이 중요하다.

* Lock, Unlock을 사용해서 쪼개지지 않게!

어플리케이션 스레드들이 queue를 이용함으로써, busy-wait에 빠지지 않고 CPU를 효율적으로 사용할 수 있다.

**T1이 저장하고 T2가 읽는 shared data에서 critical section이 없는 이유**

T1은 write만, T2는 read만 하기 때문이다.

즉, T1은 Signal(S)만 T2는 Wait(S)만 수행한다. → S=0으로 초기화

Deadlock: two or more processes are waiting indefinitely for an event that can be caused by

only one of the waiting processes.

= 하나의 프로세스만 수행될 수 있기 때문에, Waiting상태가 풀릴 수 없게 됨.

Starvation: A process may never be removed from the semaphore queue in which it is suspended

= 어떤 프로세스가 수행되지 못하고 계속 Waiting 상태

Counting semaphore: 1개 이상의 공유 메모리

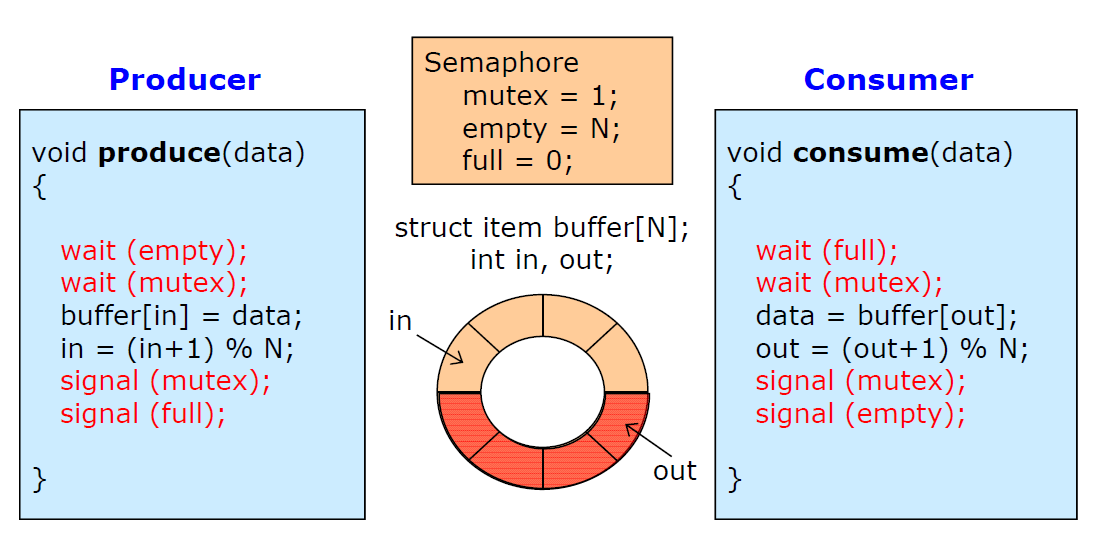
Binary semaphore: 1개의 공유메모리. 0과 1만을 사용

Mutex: binary semaphore을 lock으로 구현

큐 안에 빈 공간 개수

큐 안에 채워진 공간 개수

**Bounded Buffer Problem**

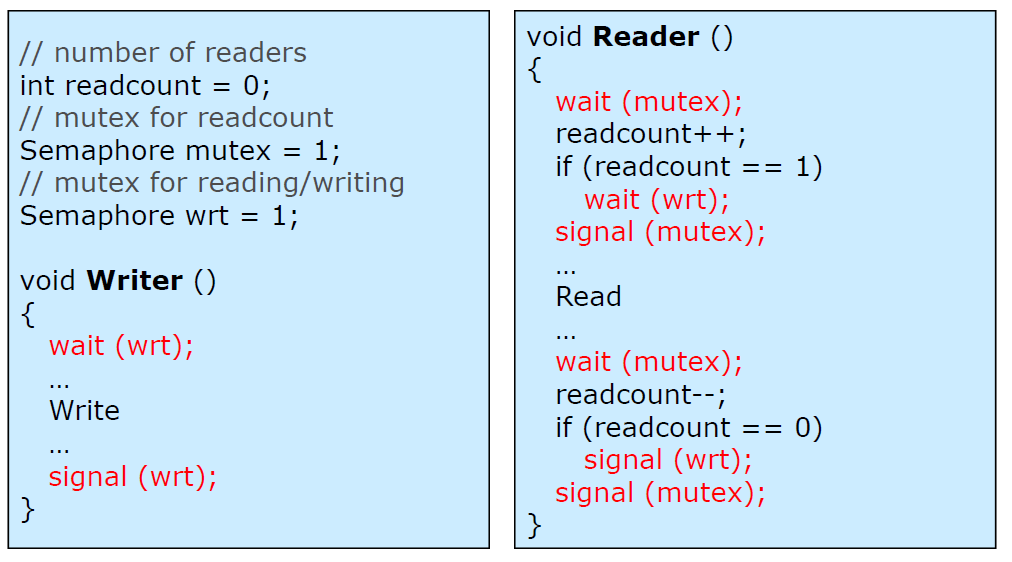


Critical problem과

Busy-wait를 해결한 코드

**Readers-Writers Problem**

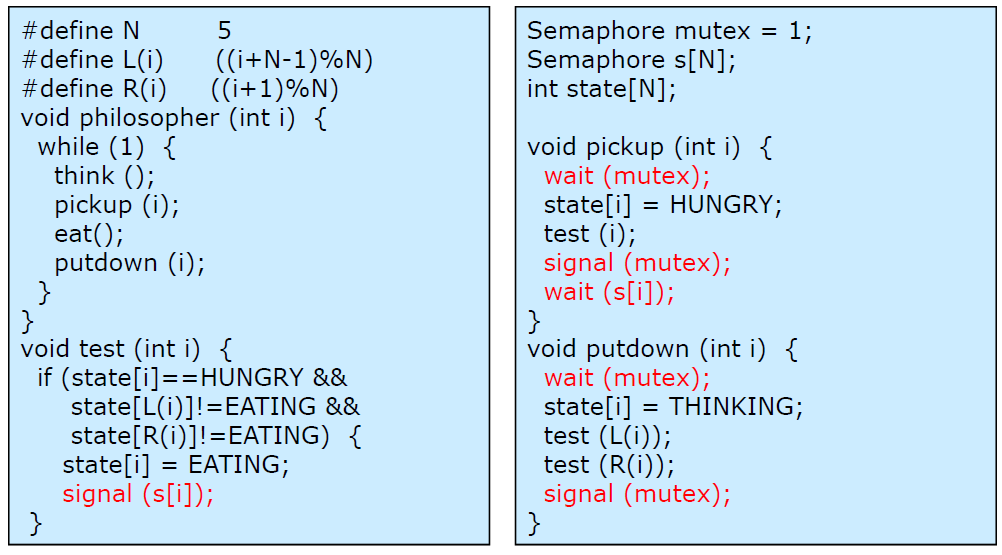
Shared data에 R-W, W-W를 동시에 하는 것은 불가능하지만 R-R (동시에 읽는 것)은 가능하다!

Writer가 Starvation이 될 수 있다는 문제점이 있음.

readcount도 shared data가 되기 때문에 보호하는 semaphore를 추가!

**Dining Philosopher**

원탁에 철학자가 다섯 명, 젓가락을 사이에 다섯 개를 두고 배가 고플 때 젓가락 두 개를 가지고 식사를 하는 방법. 이때 shared data는 젓가락, 프로세스는 철학자가 된다.

철학자가 동시에 왼쪽 젓가락을 들면 deadlock발생!

젓가락이 아니라 철학자의 상태를 저장하는 semaphore 생성 (양쪽 젓가락을 동시에 드는 형식) → deadlock 해결

왼쪽, 오른쪽이 번갈아 가면서 식사를 할 경우에 발생하는 starvation은 aging을 이용하여 해결한다.

\*semaphore는 전역변수로 사용해야 하는데 전역변수가 많으면 코드 이해가 어려운 문제가 있음.

Critical Regions: critical section을 명시, 컴파일러 차원에서 critical section을 해결하자. (개념적)

**Monitor**: 프로그래밍 언어가 지원하는 방식. 한번에 하나의 스레드만 실행할 수 있다.

자바에서만 유사한 기능인 synchronized라는 키워드가 존재한다.

Condition value: critical section안에서 이벤트를 대기할 경우 lock을 다시 풀어줘야 한다.

이벤트를 기다리게 하는 wait()와 프로세스를 다시 깨우는 signal()이 존재한다.