



# Chapter 6. Synchronization Tools (2)

## 6.5 Mutex Locks

---

- Mutex locks : 가장 간단한 동기화 도구
- Semaphore : 더 강력하고, 편리하고, 효과적인 도구
- Monitor : 뮉텍스, 세마포어의 단점 극복
- Liveness : 프로세스의 진행을 보장

### Mutex Lock?

- mutex : mutual exclusion(상호 배제)의 축약형
- critical section 보호, race condition 방지
  - 여러 프로세스가 임계구역에 들어가는 것을 방지한다.
  - 하나의 프로세스가 임계구역에 있는 **lock** 상태에서 다른 스레드가 임계구역에 접근할 수 없게 한다.
- process는 임계구역에 들어가기 전, lock을 획득하고, 빠져나올 때, lock을 반환

### 구조

```

while (true) {
    acquire lock

    critical section

    release lock

    remainder section
}

```

- `acquire()`, `release()` : lock이 사용 가능하다면 > `acquire()` 호출 성공 > lock은 사용 불가능 상태가 됨 > 작업 끝나면 `release()`호출하여 lock을 해제
- `available` : 불리언 변수, lock의 가용여부 결정

## acquire(), release()

```

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

release() {
    available = true;
}

```

- 두 함수의 호출은 원자적으로 실행되어야 한다.
- 이전 시간에 본 하드웨어 기법 중 하나를 종종 사용하여 구현함 (`compare_and_swap`)

## 단점

- Busy waiting (바쁜 대기)
  - `acquire()`호출 > `available == false` > 임계 구역에 들어가기 위해 계속 반복문을 수행해야함
  - 멀티프로그래밍 시스템에서 분명한 문제임 > 바쁜대기로 인해 더 생산적으로 쓸 수 있었던 CPU cycle을 낭비하게 됨
- Spinlock

- lock이 사용가능해 지길 기다리면서 프로세스가 계속 회전하는 것
- 장점 : 상당한 시간이 걸리는 문맥교환을 하지 않는다.
  - 즉, process가 짧은 시간 lock을 소유한다면 spinlock은 유용하다.
- 다중 처리 시스템에서 사용할 수 있다.
  - 한 스레드가 임계구역을 A core에서 실행하는 동안, 다른 스레드가 B core에서 회전을 수행하는 방식으로 locking을 구현할 수 있다.

## 6.6 Semaphores

### Semaphore?

- mutex와 유사하게 동작하지만, 프로세스를 더 정교하게 동기화할 수 있음

### Semaphore 정의

- 변수 **S**
  - 정수 변수이며, wait()/signal()로만 접근이 가능하다.
  - wait() == P()
  - signal() == V()
  - wait(), signal()은 원자적 연산이다.

### wait() signal() 정의

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

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

- S 값을 변경하는 연산(S--, S++)은 반드시 원자적으로 수행되어야 함
- 즉, 한 스레드가 S값을 변경하면 아무도 동시에 동일한 S값을 변경할 수 없다.
  - wait()의 경우, S값을 검사하는 부분(S<=0)과 변경하는 부분(S--;)이 인터럽트 없이 수행되어야 한다.

## 종류

### a. Binary Semaphores

- 0,1 로만 구분 : mutex lock과 유사하게 동작

### b. Counting Semaphore

- 제한없는 domain
- 유한한 갯수를 가진 자원에 대한 접근을 제어



#### mutex Vs Binary semaphore

- 가장 큰 차이는 소유권

(소유권? 커널 객체가 자신을 소유한 스레드의 정보를 저장하는 것)

- mutex는 커널 객체 중 유일하게 소유권의 개념을 가진다.

ex ) A스레드가 객체를 소유 > B스레드가 해당 객체를 해제하려 하면 error를 뱉어냄 > B스레드가 소유권을 가지지 않는다고 설정 > 즉, A스레드가 소유를 한 채로 종료하면, 다른 모든 스레드가 소유권을 가질 수 없으므로 deadlock 발생

- semaphore는 소유권 개념이 없으므로 이러한 문제가 발생하지 않는다.

## Counting Semaphore

### a. 사용법

- S를 사용가능한 자원의 갯수로 초기화
- 자원을 사용할 때 > wait() 호출 > 이때 S값은 감소
- 자원을 방출할 때 > signal()을 수행 > 이때 S값은 증가
- S가 0이면 모든 자원은 사용중임
  - 이후, 자원을 사용하려는 프로세스는 S가 양수가 될 때까지 block

### b. 예시

- P1, P2 두개의 프로세스 있음
- P1은 S1을, P2는 S2를 병행하게 수행하고 싶음
- S2 명령은 S1 명령이 끝나야만 수행할 수 있음

- P1, P2는 synch 세마포어를 공유, synch는 0으로 초기화

p1에 삽입할 명령

```
S1;  
signal(synch);
```

p2에 삽입할 명령

```
wait(synch);  
S2;
```

- `synch == 0` 이므로,
  - S2는 P1의 signal() 수행 후에만 가능
  - signal() 수행은 S1 수행 후에만 가능

## 구현

- mutex에 있는 busy waiting 문제는 세마포어에도 존재한다.
- 문제 해결을 위해 wait(), signal()을 수정
- wait()
  - S가 양수가 아니면 대기해야함
  - 대기 대신에, 봉쇄(일시정지)한다.
  - `waiting queue`에 넣는다.
- signal()
  - 다른 프로세스가 signal()연산을 하면 기다리던 프로세스가 재시작해야함
  - 대기 큐에 있던 프로세스를 `ready queue`로 옮긴다.
  - 이때, CPU는 CPU스케줄링에 따라, 실행중인 프로세스에서 새로 준비된 프로세스로 변경이 되거나, 안될 수 있다.

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

- 각 세마포어는 한 개의 정수 value와 프로세스 리스트 list를 가진다.

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

- wait() 해야한다면, 프로세스를 list에 넣고, 기다림(일시중지 = sleep)

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

- signal() 해야한다면, 프로세스를 list에서 꺼내고, wakeup() 연산으로 깨워준다.



바쁜 대기를 하는 세마포어(S)는 음수값을 가질 수 없다.  
 하지만, 이와같이 변경할 경우, S는 음수값을 가질 수 있다.  
 그리고, 음수의 값(의 절대값)은 대기하고 있는 프로세스 갯수이다.

## 6.7 Monitors

### difficulty of using Semaphores

- 세마포어는 동기화를 할때, 편리하고,효과적임
- 하지만, 잘못 사용하는 경우, 다양한 오류가 발생한다.

```

signal(mutex);
...
critical section
...
wait(mutex);

```

- 위의 경우에는 signal 후에 임계구역에 진입하므로, 여러 프로세스가 임계구역에 진입할 수 있다.

```

wait(mutex);
...
critical section
...
wait(mutex);

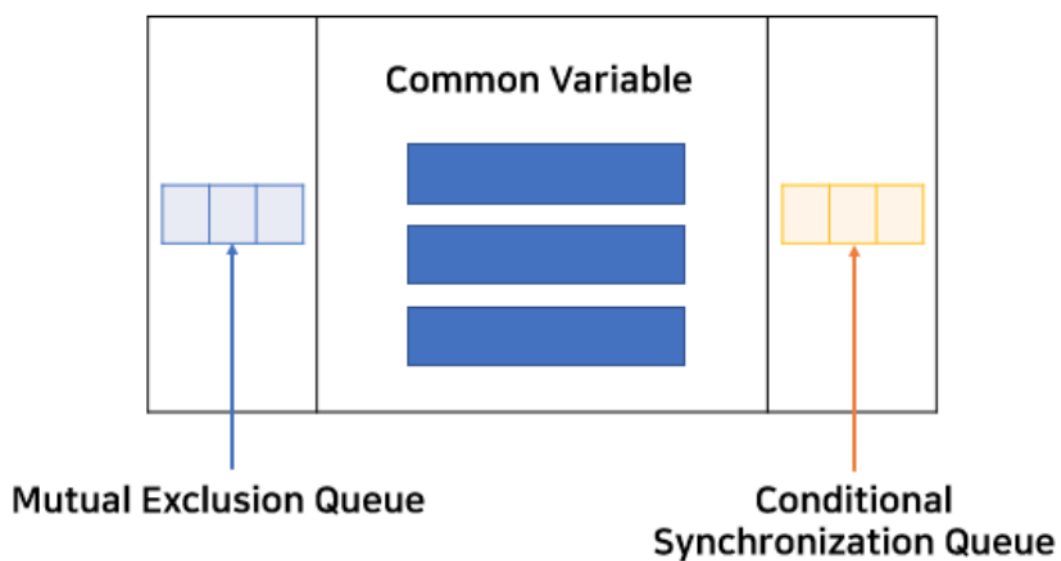
```

- 위의 경우에는 임계구역 사용 후에 signal을 하지않으므로 mutex는 영구 봉쇄 됨

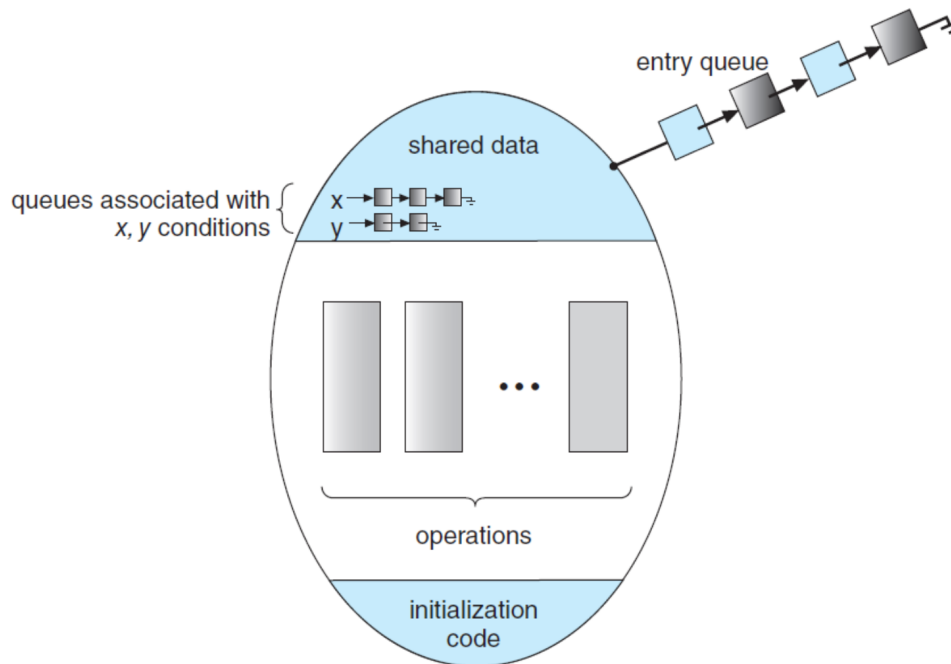
## Monitor?

- 세마포어 보다 높은 수준의 동기화 기능을 제공한다.

## Monitor의 구조



- 모니터는 **공유자원 + 공유자원접근함수** 로 이루어짐
- **mutual-exclusion(상호배제)queue, conditional-synchronization(조건동기)queue** 두 개의 queue를 가진다.
- **mutual-exclusion queue** : 공유 자원에 하나만 진입하도록 하는 큐 (강의에서는 **entry queue** 인듯?)
- **conditional-synchronization queue** : 공유자원을 사용하고 있는 프로세스가 특정한 호출(**wait()**)을 통해 조건동기 큐로 들어갈 수 있다.
- 조건 동기 큐에 들어간 프로세스A는 공유자원을 사용하는 다른 프로세스B가 **notify()**을 호출하여 깨울 수 있다. (깨워 주더라도 B가 완전히 해당 구역을 나가야 비로소 큐에 있던 프로세스가 실행됨)



## Java의 monitor

- 자바는 모니터를 제공함 : monitor-lock, intrinsic-lock
- **synchronized** 키워드, **wait()**, **notify()** 메소드를 조합하여 사용

### a. synchronized 키워드

- 임계영역에 해당하는 코드 블록을 선언할 때 사용
- 해당 코드 블록에는 **모니터락을 획득**해야 진입 가능



```
synchronized (object) {
    // critical section
}
```

```
public synchronized void add() {
    // critical section
}
```

## b. wait() and notify()

- 스레드가 어떤 객체의 wait() 메소드를 호출 > 해당 객체의 모니터락을 획득하기 위해 대기상태(wait queue)로 진입
- 스레드가 어떤 객체의 notify() 메소드를 호출 > 해당 객체 모니터에 대기중인 스레드 하나를 깨움
- notifyAll() > 해당 객체 모니터에 대기중인 스레드 전부를 깨움 → 전부 ready queue로 이동됨



notify()를 사용하면 하나의 스레드만 깨워준다. (보통은 큐에 있는 가장 앞의 스레드)

notifyAll()을 사용하는게 좋다.

왜냐하면 waiting queue에 있는 모든 스레드를 ready queue로 보내줌으로써 모든 스레드가 공평하게 기회를 가지게 된다.

## c. Condition Variables

- 동기화를 위해 대부분 mutual exclusion으로는 충분하지 않음 > 연산을 하는 스레드는 특정 조건P가 참이 될 때 까지 기다릴 필요가 있다.
- 이를 위해, 모니터 unlock > 특정 시간 wait > 모니터 lock > 조건P를 살피는 루프를 가지는 해결책이 있음
- 위 해결책은 이론적으로 동작하고, 데드락 상태를 유발하지 않음
- 그러나 구현 시 이슈 발생함 > 적절한 대기시간을 산정하는게 어렵다. (너무 작으면 스레드가 cpu를 독차지, 너무 크면 효과가 없다)
- 그래서 필요한 방법은 조건P가 참이 될 때 시그널을 보내는 것

## Semaphore vs Monitor

- 세마포어 : 초기화를 통해 들어갈 수 있는 한계를 둔다. 들어갈 때 acquire(), 나올 때 release() 함수를 실행시켜주어야 한다. (병렬 프로그래밍 환경에서 여러 프로세스가 동

시에 중요자원에 접근하지 않도록 사용하는 데이터 구조)

- 모니터 : 함수 앞에 synchronized 키워드(java)를 넣어주면 알아서 상호배제 기능을 수행한다. (동시에 공통 리소스에 접근하는 여러 프로세스를 방지하는 프로그래밍 언어 구조)
- 1. 사용하는 기술에 차이를 둠 ⇒ 상호 배제를 달성하는 데 사용되는 코드는 단일 위치에 더 구조화되어있는 반면 세마포어 용 코드는 대기 및 신호 함수 호출로 배포 =? 이게 무슨말이죠
- 2. 세마포어를 구현할 때 실수하기가 매우 쉽고 모니터를 구현할 때 실수 할 가능성이 거의 없음

## 6.8 Liveness

### Liveness

- 프로세스가 실행하는 동안 **Progress** 되는 것을 보장하기위해 시스템이 충족해야하는 조건
- Progress가 되지 않는다는 것 == bounded waiting이 지켜지지 않는 것
  - 이러한 프로세스는 **Liveness 실패** 라고 한다.
- **교착상태** 와 **우선순위 역전** 현상은 **라이브니스 실패** 를 야기한다.

### Deadlock

- 2개 이상의 프로세스가 무기한으로 기다리는 현상
  - signal(S)가 수행되기를 P0은 무한정 기다림
  - signal(Q)가 수행되기를 P1은 무한정 기다림

$P_0$	$P_1$
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>



교착상태 Vs 기아

- 교착상태는 서로 필요한 자원을 대기하면서 결코 일어나지 않는 사건을 기다림
- 기아상태는 결코 사용할 수 없는 자원을 기다림

## Priority Inversion (우선순위 역전)

- 우선 순위  $A > B > C$  프로세스가 있을 때,
  1. C가 R(공유자원) 사용 중 > Lock 상태가 되어 A가 선점 불가(wait)
  2. B(R을 필요로 하지 않음)가 ready 상태가 되어, C를 선점
  3. B가 모두 수행됨 > 이후, C가 하던 일을 이어서 수행
  4. C가 모두 수행됨
  5. A가 수행됨

⇒ A 우선순위가 가장 높는데, 마치 제일 낮은 것 처럼 수행 > 우선순위 역전 현상

## 해결방법

- **PIP(priority-inheritance protocol)** : 우선순위를 공유자원을 사용하러 온 프로세스의 우선순위로 바꾸는 것
- **PCP(priority-ceiling protocol)** : 공유자원을 사용하는 프로세스의 우선순위를 가장 높은 우선순위로 바꾸는 것

## PIP로 해결

1. C가 R을 사용 중 > A가 R을 사용하기 위해 대기중
2. PIP를 통해 C의 우선순위를 A와 같게 바꿈 → `C = A > B`
3. B가 ready 상태가 됨 > C보다 우선순위가 낮으므로 선점 불가
4. C가 모두 수행됨 > 이후, R을 반환하고 원래 우선순위로 돌아감
5. A가 수행됨
6. B가 수행됨

현강's Q : <https://www.infllearn.com/questions/318194>

### 1. 모니터와 세마포어 차이? Conditional var을 사용하는 경우 결국 프로그래머가 wait(), signal() 해주는 것 아닌가?

- Conditional var은 상호배제 해주는 것과 다른 하나의 개념인 것으로 이해함 (세마포어와 상관없는 다른 개념)
- Deadlock이 일어날수 있는 문제(철학자들 식사문제) 해결하기위한 추가적인 수단 정도로 이해됨
- 즉, mutex lock은 resource를 제어하고, conditional var은 process를 제어하는 것 같음



1. 모니터 안에서는 하나의 프로세스만 활성화 될수있도록 보장해준다

### 2. conditional var 사용범위 → 하나의 공유자원에 대해 동작하는가?

- 개발자가 정하는 것 같음
- 철학자 문제에서 철학자 별로 conditional var가 존재(5개)하고, 젓가락(resource) 사용 여부에 따라 생각하거나 먹거나 하므로
- (젓가락0, 젓가락1) 에 대해 `con_var1` 이 동작, (젓가락1, 젓가락2)에 대해 `con_var2` 가 동작..

### 3. condition 변수가 하나인 경우에는 세마포어와 동일한 것 아닌가?

- conditional 변수가 하나라는 것은 process(thread)가 하나인 것 ? → 그럼 굳이 queue에 들어가서 기다릴 필요가 있을까? → 음.. conditional 변수가 하나일거면 그냥 필요없는 거 아닌가요? (O 그렇다고 생각함)

### 4. 세마포어 + conditional var 할 수 있나?

- 안됨. conditional var은 애초에 monitor에서 사용하기위해 만든 것임