프로세스 동기화 정리

동시다발적으로 실행되는 프로세스들은 서로 협력하며 영향을 주고 받는다

이 과정에서 자원의 일관성을 보장해야 한다.

동기화의 의미

공동의 목적을 위해 동시에 수행되는 프로세스

아무렇게나 마구 실행해도 괜찮을까? → No 올바른 수행을 위해 프로세서들은 동기화되어야 한다.

동기화란?

프로세스들의 수행 시기를 맞추는 것

실행 순서 제어: 프로세스를 올바른 순서대로 실행하기

상호 배제: 동시에 접근해서는 안 되는 자원에 하나의 프로세스만 접근하게 하기

상호 배제를 위한 동기화: Bank account problem

공유가 불가능한 자원의 동시 사용을 피하기 위한 동기화

한 번에 하나의 프로세스만 접근해야 하는 자원에 동시 접근을 피하기 위한 동기화

프로세스 A	프로세스 B	현재 잔액
잔액을 읽어 들인다		10만 원
읽어 들인 값에서 2만 원을 더한다		10만 원
문맥 교환		10만 원
	잔액을 읽어 들인다	10만 원
	읽어 들인 값에서 5만 원을 더한다	10만 원
	문맥 교환	10만 원
더한 값 저장		12만 원
	더한 값 저장	15만 원
		최종 잔액 = 15만 원?

읽는 순간에 따라 값이 달라짐

Producuer & Consumer problem

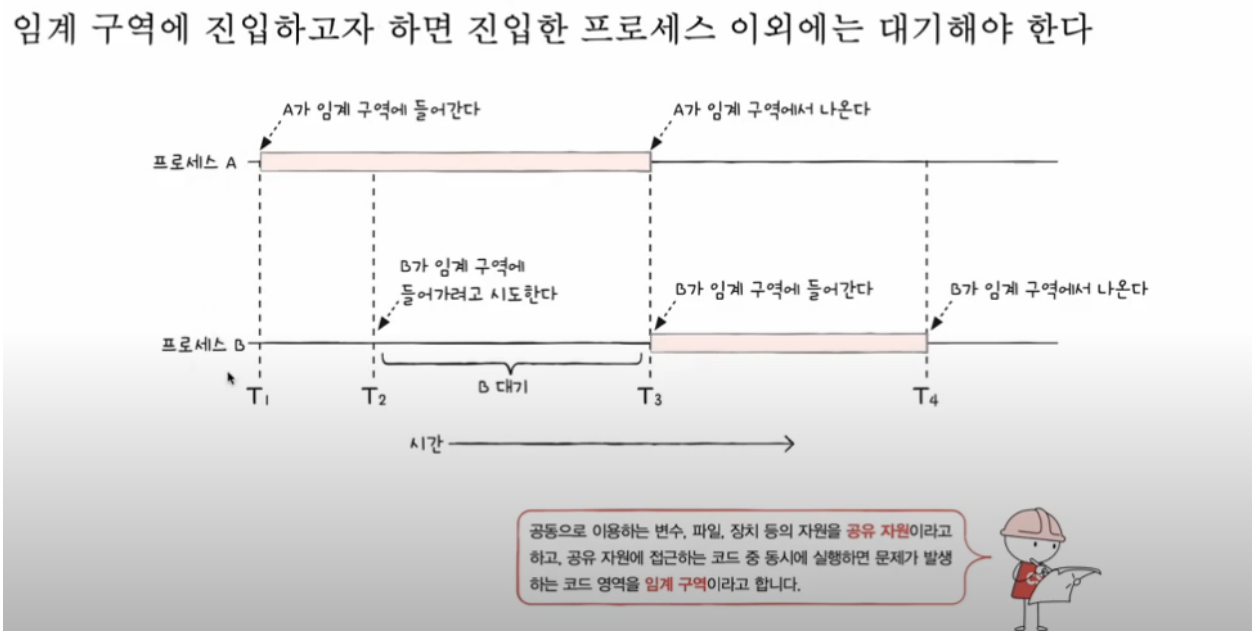
```
총합 = 10

생산자 () {
        버퍼에 데이터 삽입
        '총합' 변수 1 증가
}
소비자 () {
        버퍼에서 데이터 빼내기
        '총합' 변수 1 감소
}
```

실제 코드는 한줄이지만 결과를 보면 실행마다 달라짐

총합 ++ → 총합 변수를 레지스터에 저장, 레지스터 값 1 증가, 레지스터 값을 총합 변수에 저장

공유자원: 여러 프로세스 혹은 스레드가 공유하는 자원(전역 변수, 파일, 입출력 장치)

임계 구역: 동시에 실행하면 문제가 발생하는 자원에 접근하는 코드 영역 (총합 변수, 잔액 변수)



임계 구역에 동시에 접근하면 자원의 일관성이 깨질 수 있다.

이를 레이스 컨디션이라고 한다.

Race Condition이 발생하는 경우

1. **커널 작업을 수행하는 중에 인터럽트 발생**

- 문제점 : 커널모드에서 데이터를 로드하여 작업을 수행하다가 인터럽트가 발생하여 같은 데이터를 조작하는 경우
- 해결법 : 커널모드에서 작업을 수행하는 동안, 인터럽트를 disable 시켜 CPU 제어권을 가져가지 못하도록 한다.

2. **프로세스가 'System Call'을 하여 커널 모드로 진입하여 작업을 수행하는 도중 문맥 교환이 발생할 때**
   - 문제점 : 프로세스1이 커널모드에서 데이터를 조작하는 도중, 시간이 초과되어 CPU 제어권이 프로세스2로 넘어가 같은 데이터를 조작하는 경우 (프로세스2가 작업에 반영되지 않음)
   - 해결법 : 프로세스가 커널모드에서 작업을 하는 경우 시간이 초과되어도 CPU 제어권이 다른 프로세스에게 넘어가지 않도록 함
3. **멀티 프로세서 환경에서 공유 메모리 내의 커널 데이터에 접근할 때**
   - 문제점 : 멀티 프로세서 환경에서 2개의 CPU가 동시에 커널 내부의 공유 데이터에 접근하여 조작하는 경우
   - 해결법 : 커널 내부에 있는 각 공유 데이터에 접근할 때마다, 그 데이터에 대한 lock/unlock을 하는 방법

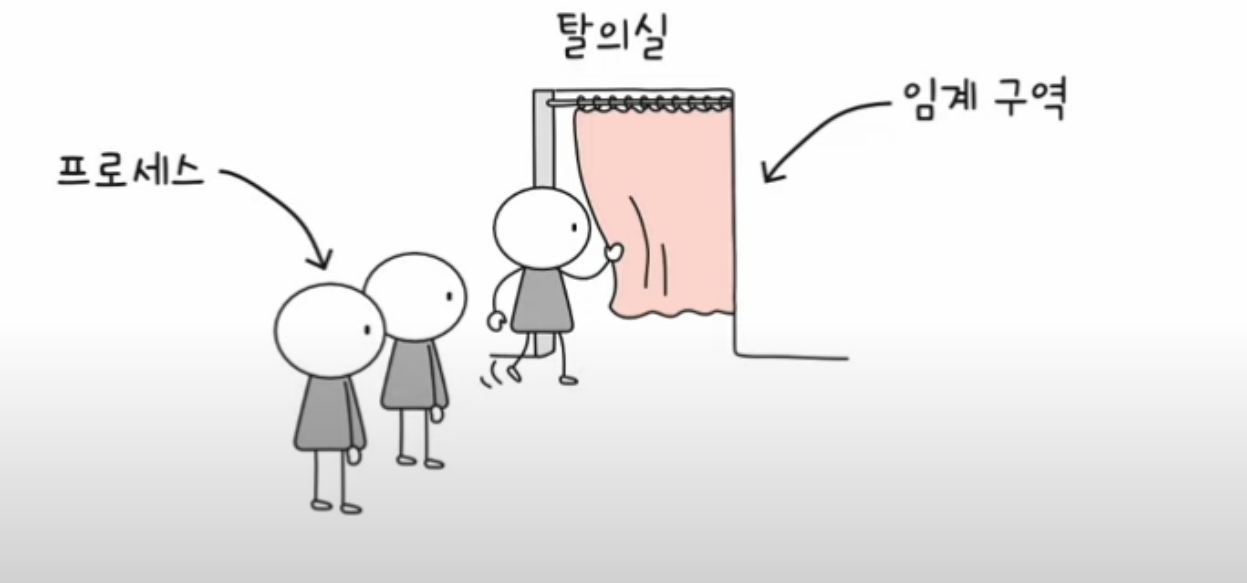
운영체제가 임계구역 문제를 해결하는 세 가지 원칙

1. 상호배제 : 한 프로세스가 임계 구역에 진입했다면 다른 프로세스는 들어올 수 없다.
2. 진행 : 임계 구역에 어떤 프로세스도 진입하지 않았다면 진입하고자 하는 프로세스는 들어갈 수 있어야 한다.
3. 유한대기 : 한 프로세스가 임계 구역에 진입하고 싶다면 언젠간느 임계 구역에 들어올 수 있어야 한다

→ 임계 구역에 들어오기 위해 무한정 대기해서는 안 된다.

동기화 기법

뮤텍스 락



자물쇠 역할: 프로세스들이 공유하는 전역 변수 lock

임계 구역을 잠그는 역할: acquire 함수

임계 구역의 잠금을 해제하는 역할: release 함수

```
acquire() {
while(lock == ture); // 만약 임계 구역이 잠겨 있다면, 임계 구역이 잠겨있
 lock = true;  // 만약 임계 구역이 잠겨 있지 않다면 임계 구역 잠금
}

release() {
 lock = false;  // 임계 구역 작업이 끝났으니 잠금 해제
}


acquire();
```

```
// 임계 구역
release();
```

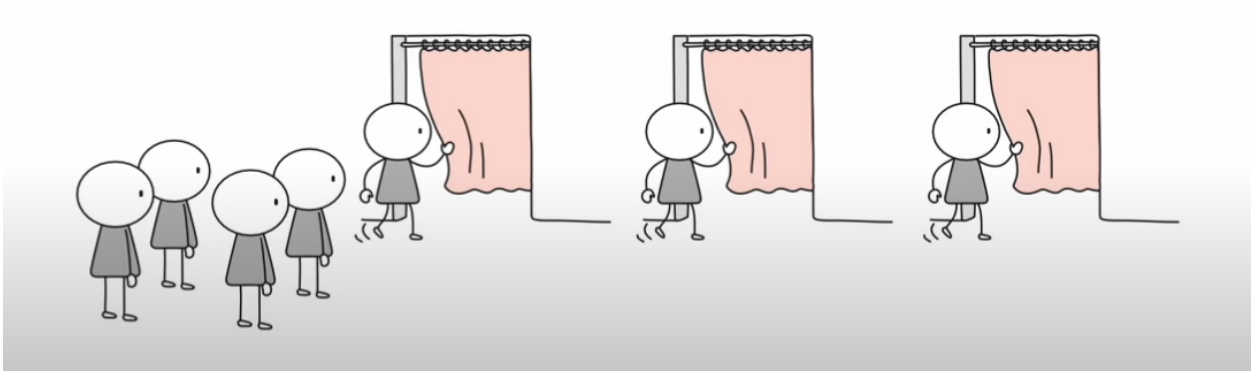
acquire 함수

- 프로세스가 임계 구역에 진입하기 전에 호출
- 임계 구역이 잠겨 있다면 열릴때까지 반복적으로 확인
- 임계 구역이 열려있다면 임계 구역을 잠그기
- busy waiting

release 함수

- 임계 구역에서의 작업이 끝나고 호출
- 현재 잠긴 임계 구역을 열기

세마포어



- 좀 더 일반화된 방식의 동기화 도구
- 공유 자원이 여러 개 있는 경우에도 적용 가능

세 개의 프로세스 P1, P2, P3가 두 개의 공유 자원(S=2)에 P1,P2,P3 순서로 접근한다고 가정

P1	P2	P3
wait()	wait()	wait()
// 임계 구역	// 임계 구역	// 임계 구역
signal()	signal()	signal()

❶ 프로세스 P1 wait 호출. S는 현재 2이므로 S를 1 감소시키고 임계 구역 진입
❷ 프로세스 P2 wait 호출. S는 현재 1이므로 S를 1 감소시키고 임계 구역 진입
❸ 프로세스 P3 wait 호출. S는 현재 0이므로 무한히 반복하며 S 확인
❹ 프로세스 P1 임계 구역 작업 종료, signal() 호출. S를 1 증가
❺ 프로세스 P3 S가 1이 됨을 확인. S는 현재 1이므로 S를 1 감소시키고 임계 구역 진입

busy waiting에 대해

- 마치 공회전하는 자동차와 같아 cpu 자원의 매우 큰 낭비이다
- 하지만 cpu 코어가 많아 코어 하나가 **busy waiting 하는 것이 문맥 교환하는 것보다 효율적인 경우 오히려 장점**으로 작용하기도 한다
- 임계 구역 진입을 위해 무한루프를 돌며 대기하는 것 대신, 프로세스를 중지시키고 큐에 넣는다

```
wait() {
    S--;
    if(S < 0) {
        add this process to Queue;
        sleep();
    }
}

signal() {
    S++;
    if( S<= 0) {
```

```
    remove a process p from Queue
    wakeup(p);
}
}
```

세 개의 프로세스 P1, P2, P3가 두 개의 공유 자원(S=2)에 P1,P2,P3 순서로 접근한다고 가정

P1	P2	P3
wait()	wait()	wait()
// 임계 구역	// 임계 구역	// 임계 구역
signal()	signal()	signal()

❶ 프로세스 P1 wait 호출. S를 1 감소시키면 S는 1이므로 임계 구역 진입
❷ 프로세스 P2 wait 호출. S를 1 감소시키면 S는 0이므로 임계 구역 진입
❸ 프로세스 P3 wait 호출. S를 1 감소시키면 S는 −1이므로 본인의 PCB를 대기 큐에 넣고 대기 상태로 전환
❹ 프로세스 P1 임계 구역 작업 종료, signal() 호출. S를 1 증가하면 0이므로 대기 상태였던 P3를 대기 큐에서 꺼내 준비 큐로 옮겨줌
❺ 깨어난 프로세스 P3 임계 구역 진입
❻ 프로세스 P2 임계 구역 작업 종료, signal() 호출. S가 1 증가하면 1
❼ 프로세스 P3 임계 구역 작업 종료, signal() 호출. S가 1 증가하면 2

모니터

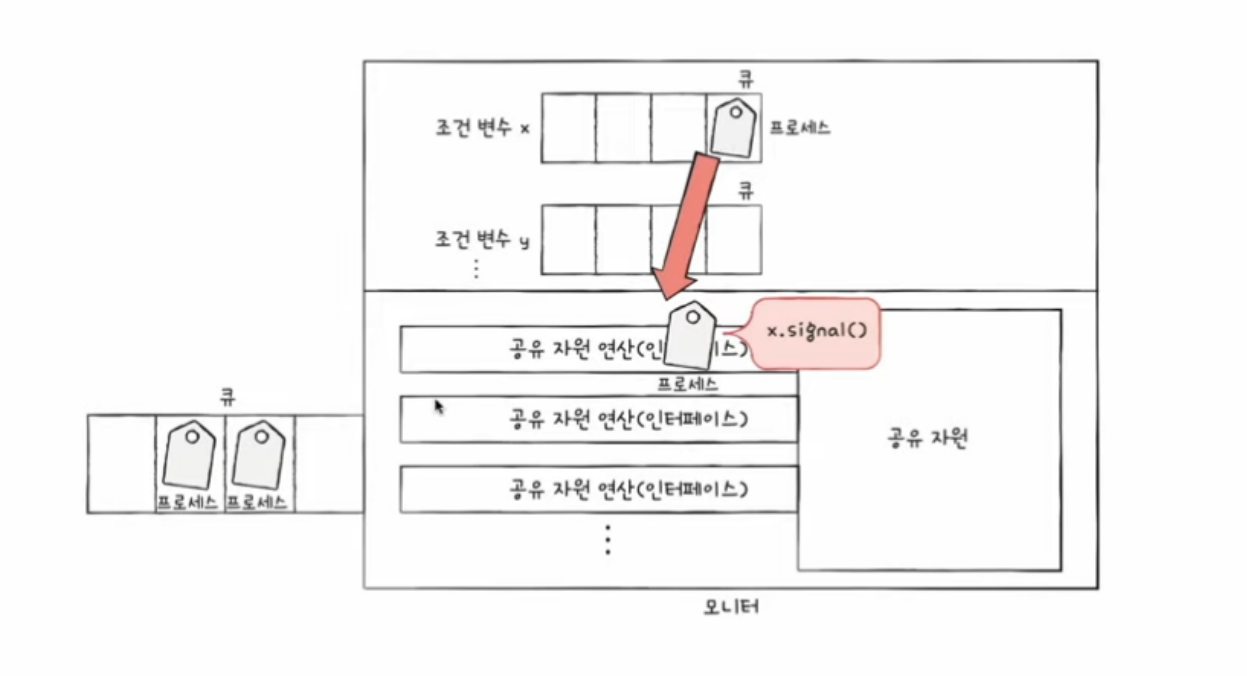
매번 임계구역 앞뒤로 wait(), signal()을 호출하는 건 실수할 여지가 있음

사용자가 다루기에 편한 동기화 도구

상호배제를 위한 동기화

- 인터페이스를 위한 큐
- 공유자원에 접근하고자 하는 프로세스를 큐에 삽입
- 큐에 삽입된 순서대로 공유자원 이용

- 조건 변수를 이용하여 실행 순서를 제어할 수 있음
- 조건변수.wait() : 대기 상태로 변경, 조건 변수에 대한 큐에 삽입
- 조건변수.signal() : wait()으로 대기 상태로 접어든 조건변수를 실행 상태로 변경



- Java의 synchronized 키워드는 스레드 동기화를 할 때 사용되는 대표적인 기법
- 자바의 모든 인스턴스는 모니터를 가지고 있으며, 모니터를 통해 스레드 동기화를 수행
- 모니터의 라이프 사이클은 synchronized 키워드에 의존하기 때문에, 인스턴스가 가지는 `wait()` , `notify()` , `notifyAll()` 메소드는 모두 synchronized 블록에서만 유의미함

모니터에는 조건 변수가 있는데, 이를 통해 `wait()`, `notify()`, `notifyAll()` 메소드가 구현되어 있음

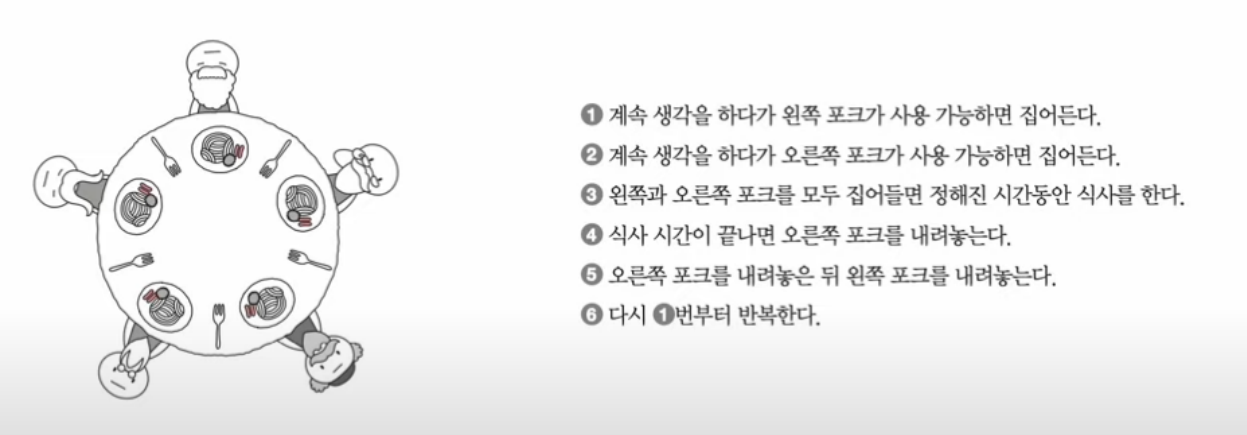
- `wait()`
  - Lock을 가진 스레드가 다른 스레드에 Lock을 넘겨준 다음 대기해야 할 때 사용
- `notify()`

- 대기하고 있는 스레드 중 하나를 깨움

- `notifyAll()`
  - 대기하고 있는 스레드 모두를 깨움

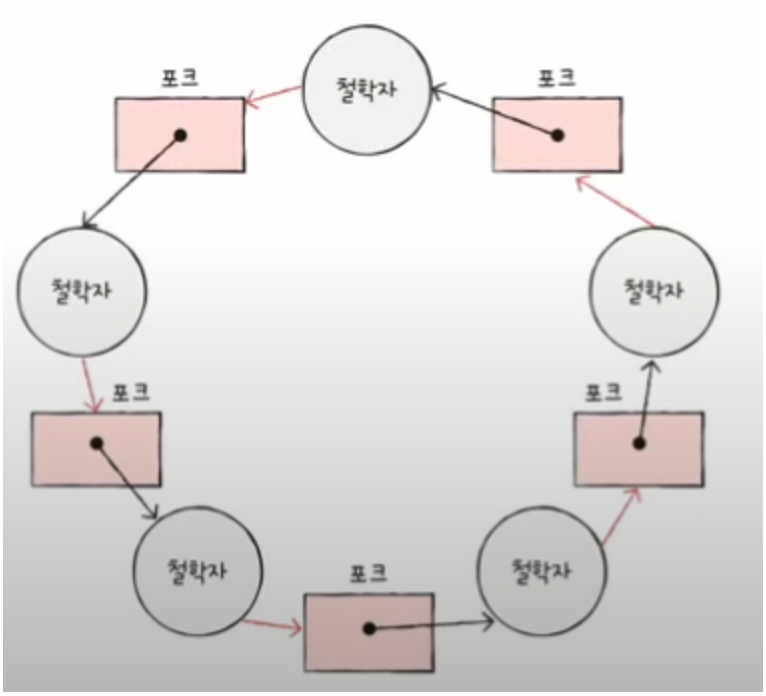
교착 상태

식사하는 철학자 문제



자원 할당 그래프

- 프로세스는 원, 자원의 종류는 사각형
- 사용할 수 있는 자원의 개수는 자원 사각형 내에 점으로 표현
- 프로세스가 어떤 자원을 할당 받아 사용 중이라면 자원에서 프로세스를 향해 화살표를 표시
- 프로세스가 어떤 자원을 기다리고 있다면 프로세스에서 자원으로 화살표를 표시



1. **상호 배제(Mutual exclusion)**

> 자원은 한번에 한 프로세스만 사용할 수 있음

2. **점유 대기(Hold and wait)**

> 최소한 하나의 자원을 점유하고 있으면서 다른 프로세스에 할당되어 사용하고 있는 자원을 추가로 점유하기 위해 대기하는 프로세스가 존재해야 함

3. **비선점(No preemption)**

> 다른 프로세스에 할당된 자원은 사용이 끝날 때까지 강제로 빼앗을 수 없음

4. **순환 대기(Circular wait)**

> 프로세스의 집합에서 순환 형태로 자원을 대기하고 있어야 함

예방(prevention)

교착 상태 발생 조건 중 하나를 제거하면서 해결한다 (자원 낭비 엄청 심함)

상호배제 부정 : 여러 프로세스가 공유 자원 사용

→ 프린터를 동시에 사용, 현실적으로 불가능

점유대기 부정 : 프로세스 실행전 모든 자원을 할당

→ 자원의 활용률을 낮추는 방식

비선점 부정 : 자원 점유 중인 프로세스가 다른 자원을 요구할 때 가진 자원 반납

→ 선점이 가능한 자원에 한해 효과적 CPU, 모든 자원이 선점 가능한 것은 아님 (프린터)

순환대기 부정 : 자원에 고유번호 할당 후 순서대로 자원 요구

→ 모든 자원에 번호를 붙이고 오름차순으로 자원을 할당, 모든 자원에 번호를 붙이는 것은 어려운 작업, 어떤 자원에 어떤 번호를 붙이느냐에 따라 활용률이 달라짐

회피(prevention)

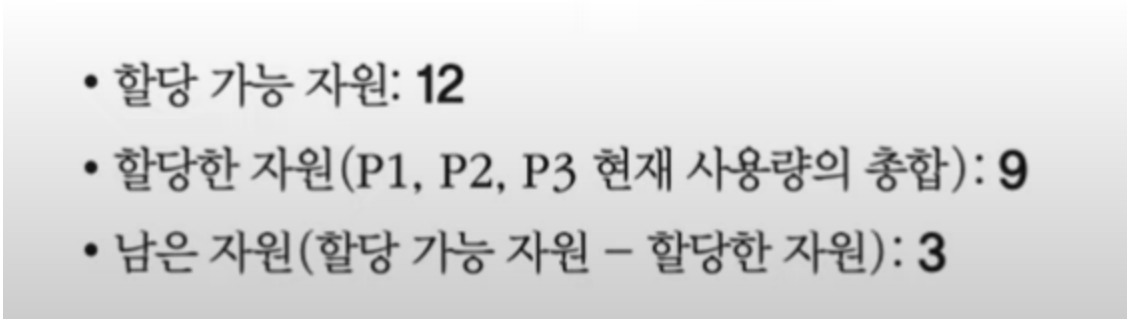
교착 상태가 발생하지 않을 만큼 조심 조심 할당하기

안전 순서열 - 교착 상태 없이 안전하게 프로세스들에 자원을 할당할 수 있는 순서

안전 상태 - 교착 상태 없이 모든 프로세스가 자원을 할당 받고 종료될 수 있는 상태

불안전 상태 - 교착 상태가 발생할 수 도 있는 상태

프로세스	요구량	현재 사용량
P1	10	5
P2	4	2
P3	9	2



P2 → P1 → P3 순으로 자원을 할당하면 안전 순서열이 존재함

은행원 알고리즘

A, B, C : 각 자원 // P1, P2, P3 : 각 프로세스

Allocation : 프로세스가 처음 실행됐을 때 할당받은 자원의 양을 나타낸다.

Max Need : 프로세스가 최대로 요청할 수 있는 범위를 나타낸다. 항상 Max Need만큼 요청하지 않을 수도 있다.

Current Available : 프로세스들이 자원을 요청했을 때 빌려줄 수 있는 자원의 수를 타나낸다.

Remaining Need : 프로세스가 이미 할당받은 자원을 제외하고 최대로 요청할 수 있는 자원의 갯수를 나타낸다. Max Need - Allocation 값이다.

Total A=10, B=5 and C=7 Resources

Process	Allocation			Max Need			Current Availabe			Remaining Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	4	2	2				2	1	1
P5	0	0	2	5	3	3				5	3	1
	7	2	5									

Banker's Algorithm은 자원을 요청하는 프로세스의 Remaining Need값이 Current Available을 초과할 경우자원을 할당해주지 않는다. 반대로 Remaining Need 값이 Current Available을 초과하지 않는 Safe State에만 자원을 할당해준다.

Available값은 할당해줄수 있는 최대 값을 나타내는데 만약 Remaining Need 값이 Current Available값보다 1이라도 넘치게 되면 절대 할당을 해주지 않는다. P1을 예시로 보면 P1의 Remainiing Need 값은 7 4 3이기 때문에 Current Abailable값을 훨씬 넘는다. 물론 P1이 Remaining Need값 전부를 항상 할당 요청하는 것은 아니다. 언제는 A자원만 1개를 요청할 때도 있고 언제는 ABC 전부 다 최대로 요청할 수도 있다. Current Available안에서 요청하더라도 P1의 요청은 잠재적으로 Current Available값을 초과한 자원을 요청할 수 있는 가능성이 있기 때문에 절대 할당해주지 않는다.

장점

1. 교착 상태가 발생하기 전에 예방할 수 있다

2. 교착 상태를 예방하기 때문에 시스템 성능 향상을 기대할 수 있다

단점

1. 프로세스의 자원 요청 순서를 변경하지 못할 경우 사용하지 못한다.

2. 프로세스가 얼만큼의 자원을 사용할 지 미리 파악하고 알고있어야 하며 중간에 자원 사용량이 늘어나는 상황은 허용하지 않는다.

3. 항상 Safe State를 유지해야 하기 때문에 자원 사용량이 낮아진다.

교착 상태 검출 후 회복

- 선점을 통한 회복
- 교착 상태가 해결될 때까지 한 프로세스씩 자원을 몰아주는 방식

- 프로세스 강제 종료를 통한 회복
- 교착 상태에 놓인 프로세스 모두 강제 종료(작업 내역을 잃을 위험)
- 교착 상태가 해결될 때까지 한 프로세스씩 강제 종료(오버헤드)

무시 방법

타조 알고리즘