프로세스 동기화

프로세스 동기화란

> 💡 멀티 프로세스 환경에서 2개 이상의 프로세스가 동시에 메모리에 접근하는 경우 데이터 일관성이 무너질 수 있기 때문에 이를 막기 위한 매커니즘

동기(Synchronous) / 비동기(Asynchronous)

동기

- 요청과 그 결과가 동시에 일어난다는 약속
- 시간이 얼마나 걸리던지 요청한 자리에서 결과가 주어져야 한다.
- 설계가 간단하고 직관적이지만 결과가 주어질 때까지 계속 대기 상태로 있어야 함

비동기

- 요청과 결과가 동시에 일어나지 않을 것이라는 약속
- 동기보다는 복잡하지만 요청 후 대기하지 않고 그 시간에 다른 작업이 가능하여 효율적으로 자원을 사용할 수 있다.
  ex) I/O 작업
- 요청 후 callback을 통해 처리 결과를 알 수 있다.

```
fucntion A(){

    console.log("A 시작");
    B();
    console.log("A 종료");
}

function B(){
    console.log("B 시작");
```

```
    setTimeout(1000,{
        console.log("B 종료");
    });
}
```

💡 결과 값
A 시작
B 시작
A 종료
B 종료

동기 처리인 경우에는 B 함수의 결과를 A 함수가 처리해야하지만
setTimeout은 비동기 함수이기 때문에 B 함수의 결과 종료와는 관계없이 A 함수는 종료되고
B함수가 callback을 통해 처리한 결과값을 알려준다.

Blocking / Non-Blocking

Blocking

- A함수가 B함수를 호출할 때 B 함수가 자신의 작업이 종료되기 전까지 A 함수에게 제어권을 돌려주지 않는 것을 의미합니다.
  → B 함수가 종료될 때까지 A 함수는 다른 일을 수행할 수가 없다.

Non-Blocking

- A함수가 B 함수를 호출할 때 B 함수가 A 함수에게 곧바로 제어권을 돌려주게 되는데 A함수는 곧바로 제어권을 가지고 다른일을 수행할 수 있습니다.

동기 / 비동기와 Blocking / Non-Blocking 차이

동기 / 비동기 = 호출당한 함수의 종료를 누가 처리하느냐

Blocking / Non-Blokcing = 제어권이 누구에게로 돌아가느냐

경쟁 상황 (Race Condition)

> 💡 커널 수행 중 인터럽트 발생 시, 커널 수행 중 context switching 발생 시 멀티 프로세서에서 커널 내부에 공유 자원을 동시에 접근하는 경우

임계 구역 (The Critical-Section)

- 멀티 프로세서에서 경쟁 상황을 해결하기 위해 한 개의 프로세만 들어갈 수 있도록 임계 구역을 지정하여 해결한다
- 하나의 프로세스가 ciritical section에 있을 때 다른 프로세스가 접근하지 못하도록 막는다

상호 배제 (Mutex : Mutual Exclustion)

- critical section을 가진 스레드의 런타임이 겹치지 않게 하는 것
- locking, unlocking 사용
- 이미 점유된 상태인 경우 CPU 자원을 내려놓고 대기 큐에서 대기
- 작업을 완료한 스레드는 뮤텍스락을 해제하고 대기큐에 신호를 보내 점유할 수 있도록 알려줌
- Non-Busy-Wait : CPU를 점유하지 않고 대기 큐에서 대기하는 방식

데커(Dekker) 알고리즘

- flag : 프로세스 중 어떤 것이 임계 구역에 들어갈지
- turn : 어떤 것이 임계 구역 사용 중인지
- 속도가 느리며 구현이 복잡함
- Busy waiting 때문에 CPU을 계속 점유함

피터슨(Peterson) 알고리즘

- 데커와 유사하지만 상대 프로세스에 진입 기회 양보

제과점(Bakery) 알고리즘

- 가장 작은 번호표를 가진 프로세스가 진입

스핀락

- 진입이 가능할 때까지 루프를 돌면서 재시도를 하는 방식
- 이때문에 CPU를 계속 점유하는 Busy-Waiting이 발생
- 대신 스케줄링 지원을 받지 않기 때문에 Context-Switching이 발생하지 않음
- 빠르게 진입이 가능한 경우에는 context-switching 비용이 들지 않기 때문에 효율이 좋지만
- 진입이 느린 경우엔 CPU 점유를 계속 하기 때문에 효율이 떨어진다
- 문맥 교환이 없기 때문에 멀티 프로세서 시스템에서만 사용이 가능
  - 1개의 프로세서를 점유하는 동안 다른 프로세서가 임계 구역에서의 작업을 끝 마쳐야 함

세마포어 (Semaphore)

- 세마포어는 여러 개의 스레드가 임계 구역에 접근이 가능하다
- wait, signal로 구현
- wait이 먼저 호출되어 임계 구역에 들어갈 수 있는지,
  우선적으로 실행되어야 할 스레드가 실행되는지 확인
- P 연산 / Wait 함수를 통해 세마포어 값 감소
  - 세마포어의 값이 음수인 경우 대기 큐에 존재하는 프로세스, 스레드를 의미한다.
- V 연산 / Signal 함수를 통해 세마포어 값 증가
- 세마포어가 다루는 프로세스 개수
  - 이진 세마포어 (binary Semaphore) → Mutex
  - Mutex → 이진 세마포어 (불가능)

        - lock 걸거나 해제할 수 있는 주체가 다르기 때문에 Mutex는 세마포어가 될 수 없음
    - 카운팅 세마포어 (n개)
- 대기 큐에서 나오는 방식에 따른 종류
    - 강성 세마포어 (FIFO)
    - 약성 세마포어 (명시 X)
- Non-Busy-Wait

뮤텍스와 세마포어의 비교

공통점

- 동시에 실행되는 프로세스 / 스레드들 간에 공유 자원을 보호한다.
- 잘못 사용 시 교착 상태가 발생할 수 있음

차이점

- 뮤텍스는 락
    - 한 개의 프로세스 / 스레드 접근
- 세마포어는 공유 변수
    - N 개의 프로세스 / 스레드 접근
- 뮤텍스와 세마포어 값 변경 권한
    - 뮤텍스의 경우 락을 소유한 프로세스 / 스레드만 획득 및 해제가 가능
    - 세마포어의 경우 소유하지 않은 프로세스 / 스레드도 세마포어 값 수정 가능
- 세마포어의 경우 작업 간의 실행순서 동기화가 가능

모니터 (Monitor)

- 상호배제가 가능 데이터 구조를 제공, 프로그래머가 구현
- 상호배제 가능한 데이터 = Abstracted Data Type, 객체 지향 컨셉과 유사
- 하나의 프로세스 내 다른 스레드 간의 동기화에 사용

- 라이브러리 혹은 프레임워크가 제공
- syncronized, wait(), notiry() 등의 키워드로 편하게 동기화 가능

구조

- Entry queue (진입 큐)
  - 모니터 내의 procedure 수만큼 존재
- Mutual exclusion
  - 모니터 내에는 항상 하나의 프로세스만 진입 가능
- Information hiding (정보 은폐)
  - 공유 데이터는 모니터 내의 프로세스만 접근 가능
- Condition queue (조건 큐)
  - 모니터 내의 특정 이벤트를 기다리는 프로세스가 대기
- Signaler queue (신호 제공자 큐)
  - 모니터에 항상 하나의 신호제공자 큐 존재

데드락

둘 이상의 프로세스가 다른 프로세스가 점유하고 있는 자원을 서로 기다릴 때 무한 대기에 빠지는 상황

발생 조건

상호 배제

- 한 번에 하나의 프로세스 하나만 해당 자원을 사용할 수 있다.

점유 대기

- 자원을 최소한 하나 보유하고, 다른 프로세스에 할당된 자원을 점유하기 위해 대기하는 프로세스가 존재한다.

비선점

- 이미 할당된 자원을 강제로 빼앗을 수 없다.

순환 대기

- 대기 프로세스의 집합이 순환 형태로 자원을 대기하고 있는 상태

데드락 해결 방법

1. 데드락이 발생하지 않도록 예방
2. 데드락 발생 가능성을 인정하면서 적절하게 회피
3. 데드락 발생을 허용하지만 데드락을 탐지하여 회복하기

데드락 예방

위의 4가지 발생 조건 중 하나라도 발생하지 않게 한다면 데드락을 예방할 수 있다.

자원의 상호 배제 조건 방지

- 한번에 여러 프로세스가 공유 자원을 사용할 수 있게 합니다.
- 다만 동기화나 데이터의 일관성 문제가 생길 수 있습니다.

점유 대기 조건 방지

- 프로세스 실행에 필요한 모든 자원을 한꺼번에 요구하고 허용할 때까지 작업을 보류해서 다른 자원을 점유하기 위한 대기 조건을 성립하지 않도록 합니다.

비선점 조건 방지

- 우선순위가 높은 프로세스가 이미 다른 프로세스에게 할당된 자원을 선점할 수 있도록 합니다.

순환 대기 조건 방지

- 자원이 순환 형태로 대기하지 않도록 단방향으로만 자원을 요구할 수 있도록 합니다.

단점 : 데드락이 자주 발생하지 않으며 예방 방법이 시스템의 처리량이나 효율성을 떨어뜨린다.

데드락 회피

- 안정 상태 : 프로세스들에게 데드락이 발생하지 않으면서 모두에게 자원을 할당할 수 있는 상태
- 안전 순서 : 특정 순서로 프로세스에게 자원 할당을 진행할 때 데드락이 발생하지 않는 순서
- 불안정 상태 : 데드락 발생 가능성이 있는 상황

: 자원을 할당한 이후에도 Safe State (안정 상태)를 유지 할 수 있도록 할당을 허용하자는 것

Banker’s Algorithm

> 💡 미리 결정된 모든 자원들의 최대 가능한 할당량을 가지고 시뮬레이션 해서 safe state에 들 수 있는지 여부를 검사한다.

- 자원 할당량을 사전에 파악하고 데드락을 회피할 수 있도록 함
- 단점
  - 미리 최대 자원 요구량을 알아야 한다.
  - 할당할 수 있는 자원 수가 일정해야 한다.

데드락 탐지(Detection) 및 회복(Recovery)

탐지 기법

- Allocation, Request, Available 등으로 데드락 발생 여부 탐색
- 자원 할당 그래프를 통해 탐지하는 방법도 있음

회복 기법

- 단순히 프로세스 1개 이상 중단시키기
  - 교착 상태에 빠진 모든 프로세스를 중단시키는 방법
    - 연산 결과가 모두 날라가서 결과가 폐기 될 수 있는 부작용이 생김
  - 프로세스를 하나씩 중단시키면서 탐지 알고리즘으로 데드락을 탐지하면서 회복하는 방법

        - 매번 탐지 알고리즘을 호출 및 수행해야 하므로 부담이 되는 작업일 수 있다.
- 자원 선점하기
    - 프로세스에 할당된 자원을 선점하여 교착 상태를 해결할 때까지 그 자원을 다른 프로세스에 할당해 주는 방법