JAVA NETWORK

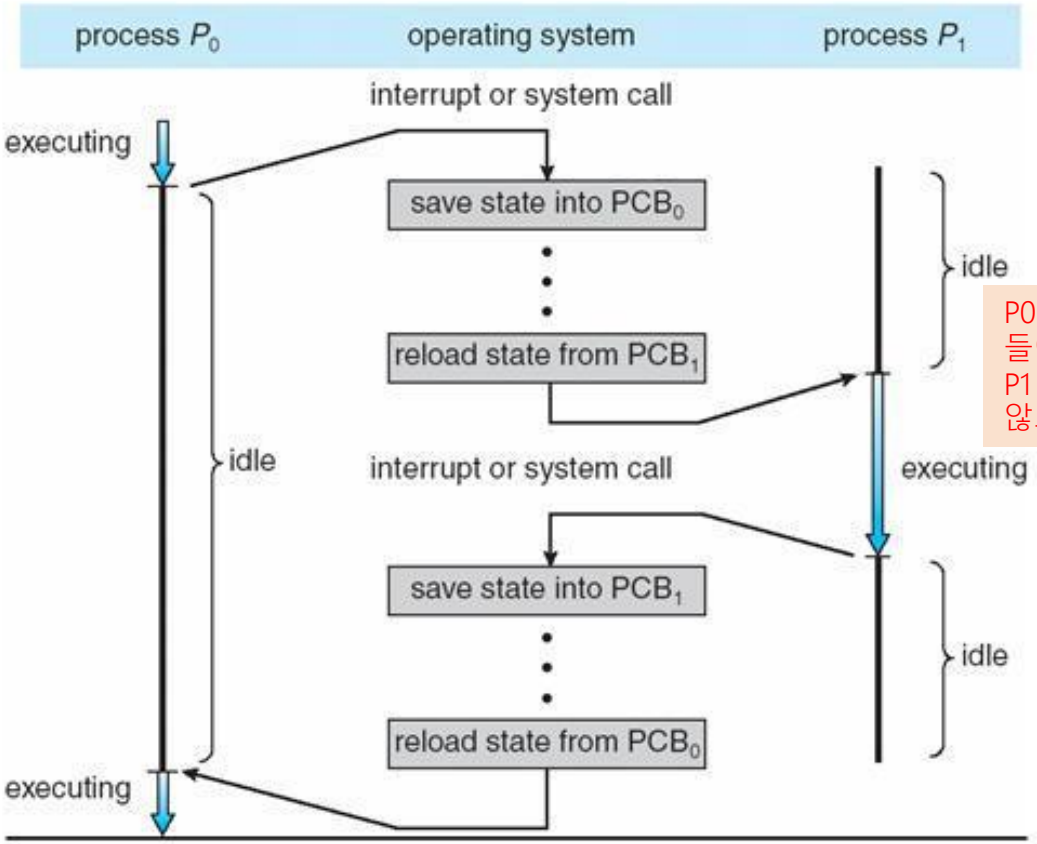
한다은

(https://github.com/RayJUNRein/JavaProgramming.git)

◆ 컨텍스트 스위칭 (Context Switching)

: CPU가 프로세스를 실행하고 있는 상태에서 인터럽트에 의해 다음 우선 순위를 가진 프로세스가 실행되어야 할 때, 기존의 프로세스 정보들은 PCB에 저장하고 다음 프로세스의 정보를 PCB에서 가져와 교체하는 작업

-> 멀티 프로세싱, 스레딩 운영이 가능하게 함



✓ 스레드가 프로세스보다 빠른 이유

: 스레드는 프로세스 영역인 text, data, heap 제외하고 자신의 영역인 스택 및 간단한 정보만 저장하므로 훨씬 빠름

✓ 단점

: 기존 프로세스의 정보를 저장하고 다음 프로세스의 정보를 불러오는 과정에서 CPU는 정지 상태가 됨.
너무 자주 일어나면 오버헤드 발생해 성능 저하

✓ 발생 상황

1. I/O Interrupt
2. CPU 사용시간 만료
3. 자식 프로세스 Fork

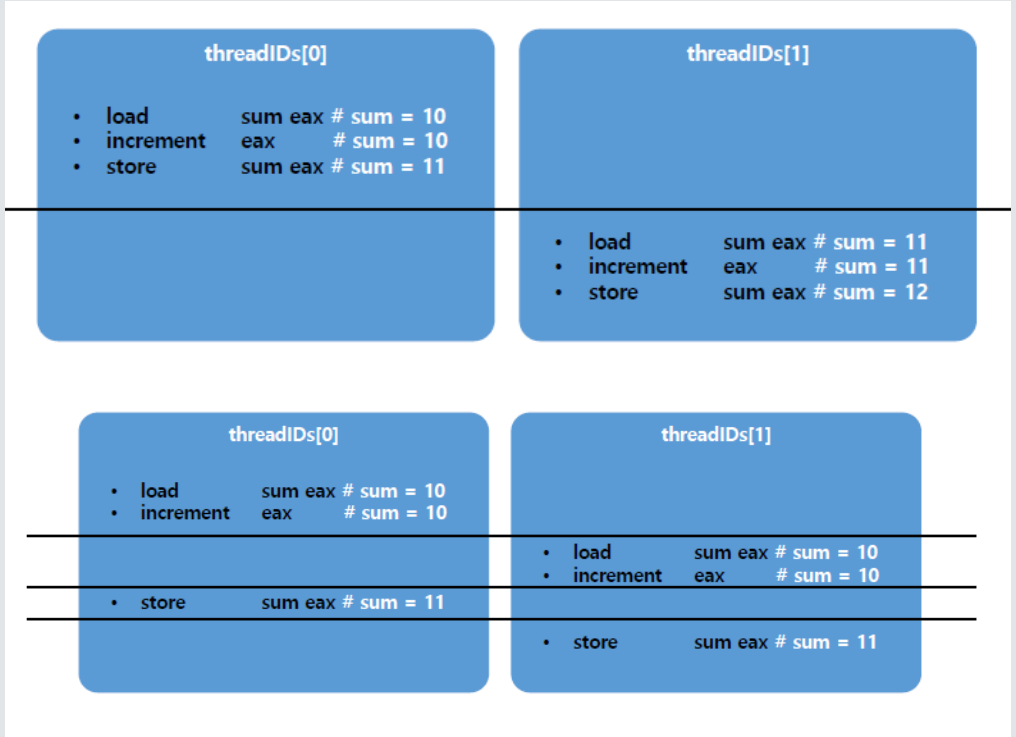
* '스케줄러 ' 가 결정

◆ Memory Sharing

- 스레드는 자신의 프로세스의 가상 메모리를 공유할 수 있음
- 단, 스레드 각자가 가진 스택은 공유 X
- 예외) 스택에서 static 변수는 공유 O

◆ Unsafe Region

- ex) 원하는 결과(위) 와 실제 실행 결과(아래)



* 언제 어디서 스케줄러가 컨텍스트 스위칭을 발생시킬지 모름

공유되는 변수는 이를 신경써야 함

◆ 크리티컬 섹션 (Critical Section)

✓ 개념적 크리티컬 섹션

: 공유 변수를 함께 이용하는 임계 구역

✓ 기능적 크리티컬 섹션

1. 동일한 프로세스 내의 여러 스레드 사이에서만 동기화 가능
2. 커널 모드 객체(Mutex, Semaphore)가 아닌 유저 모드 동기화 객체이므로 가볍고 빠름
3. 먼저 접근한 스레드는 EnterCriticalSection 통해 락을 획득하고, 이후 들어오려는 스레드는 대기 LeaveCriticalSection으로 락 해제하면 대기 중이던 다른 스레드 접근 가능
4. 대기 중인 스레드는 다른 스레드에게 CPU를 양보 해야 하므로 컨텍스트 스위칭이 발생 대기 중에는 CPU 점유하지 않음
   -> CS 발생시키지 않기 위해 스핀 락 사용하기도 함

◆ 크리티컬 섹션 (Critical Section)

✓ 기능적 크리티컬 섹션 (상세 설명)

- **크리티컬 섹션**은 프로세스 하나에 포함된 여러 개의 스레드가 공유 리소스에 접근할 때 배타적 제어를 하기 위한 구조
- Windows 객체 중 하나, 프로그램에서 CRITICAL_SECTION 타입 변수 선언해서 사용
- 일반 Windows 객체와 달리 프로세스의 메모리 공간에 확보된 변수를 이용

  -> 동일한 프로세스 내의 스레드 동기화에 사용 O / 다른 프로세스 간 동기화에는 사용 X
- 배타적 제어를 하기 위한 과정

  1) 공유 리소스에 대해 크리티컬 섹션 객체 정의 (InitializeCriticalSection 으로 변수 초기화)

  2) 접근하려는 스레드는 객체의 소유권을 시스템에 요구 (EnterCriticalSection / TryEnterCriticalSection 으로 소유권 획득 요구)

  3) 소유권을 얻은 스레드는 리소스에 접근하고 추후 소유권 반납 (LeaveCriticalSection 으로 소유권 반납)
- 크리티컬 섹션 = EnterCriticalSection 과 LeaveCriticalSection 으로 둘러싸는 공유 리소스에 접근하는 부분의 코드

  = 스레드 간 경합을 일으킬 가능성이 있는 위험한 구역
- 소유권은 하나뿐이므로 한 스레드가 소유권 가진 경우 소유권 요구한 다른 스레드는 대기 상태로 들어가서 돌아오지 않음

  소유권 획득할 수 있는 상태 되면 Windows 가 자동으로 스레드 실행 재개

  대기 상태 중에는 CPU 파워 소비하지 않아 busy wait 문제 발생하지 않음

  EnterCriticalSection 대신 TryEnterCriticalSection 사용하면 바로 복귀하므로 소유권 획득할 때까지 다른 처리 진행 가능

◆ 동기화 기법 1. 뮤텍스 (Mutex)

: Critical Section 지정해 이 곳의 lock 권한을 하나의 스레드에게만 부여

서로 다른 프로세스에 속한 스레드도 가능

◆ 동기화 기법 2. 모니터 (Synchronization)

: Mutex(Lock) 과 Condition Variables(Queue) 가짐

- Critical Section 지정해 이 곳의 lock 권한을 하나의 스레드에게만 부여
- wait(), notify() 매서드 사용

◆ 동기화 기법 3. 세마포어 (Semaphore)

: 동시에 리소스에 접근 가능한 허용 가능한 Counter의 수를 가진 Counter

- Binary Semaphore) Counter 1개 (= Mutex)
- Counting Semaphore) Counter 2개 이상

◆ 동기화 (Synchronizing)

: 의도하지 않은 곳에서

컨텍스트 스위칭(Context Switching)이 일어나도

프로그램이 정상 작동하도록 만드는 것

✓ Wait

1. 동기화된 지역이 동작하기 전에,

   이 구간을 들어가도 되는지 확인
2. 조건이 충족되어 들어가도 된다면,

   Unsafe Region의 변수 등을 이용하러 들어감
3. 조건이 충족되지 않는다면, 계속 대기

✓ Signaling

1. 동기화된 지역에서 모든 일을 마치고

   Wait하고 있는 스레드를 깨워 들어가게 함

◆ 동기화 기법 2. 모니터 (Synchronization) (상세 설명)

✓ 사용

1. 동기화 매서드 사용

```
ex) public synchronized void method() { }
```

2. 객체 변수에 사용

```
ex) private Object sharedObj = new Object();
    public void method() {
    synchronized(sharedObj) { } } pthread_mutex_unlock(&mutexTest);
```

✓ 상태제어 매서드 (동기화된 지역 내에서 사용)

1. wait() – 스레드가 lock 권한을 가지고 있다면 반납하고 대기하게 만듦
2. notify() - wait 하고 있는 스레드에게 다시 lock권한 부여해서 동작하게 함

◆ Race Condition

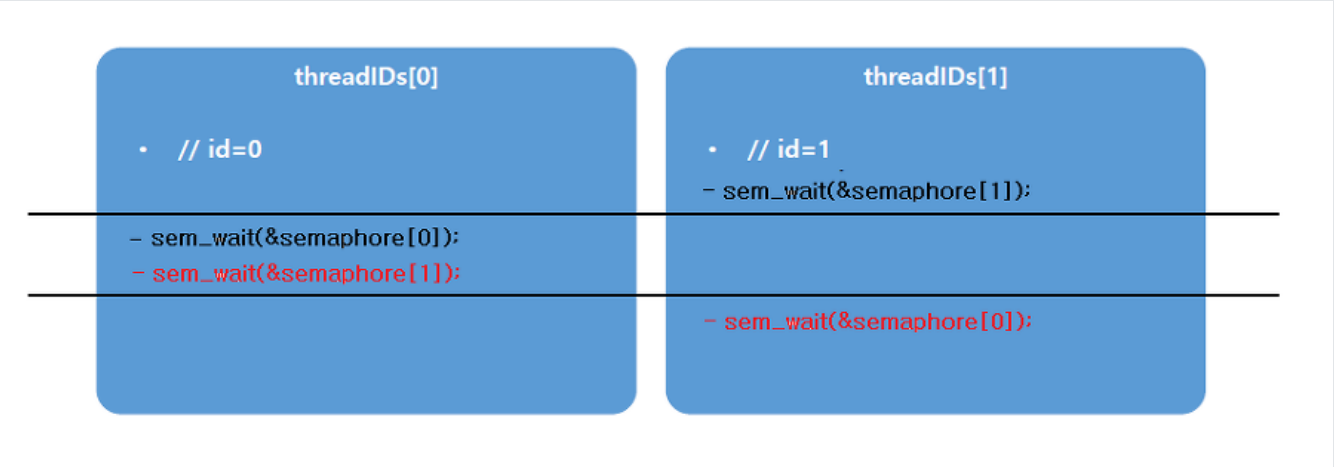
: 스케줄러가 누구에게 먼저 CPU 점유권을 줄지 모르는 상황에서 일어남

-> 해결하기 위해 동기화가 필요함

◆ 교착 상태 = 데드락 (Deadlock)

: 2개 이상의 작업이 서로 상대방의 작업이 끝나기를
기다리고 있어서 서로가 아무것도 완료하지 못함

-> 동기화가 잘못되면 일어남



✓ 발생 조건 및 회피 방법

발생은 조건 모두 만족할 때
회피는 하나라도 해결하면

1. 상호배제(Mutual exclusion)

: 프로세스들이 필요로 하는 자원에 대해
배타적 통제권 요구

⇒ 공유 불가능한 해당 자원 공유하도록 만듦

2. 점유대기(Hold and wait)

: 프로세스가 할당된 자원을 가진 상태에서
다른 자원 기다림

⇒ 한 프로세스가 실행되기 전 모든 자원 할당

3. 비선점(No preemption)

: 프로세스가 어떤 자원 사용을 끝낼 때까지
그 자원 뺏지 못함

⇒ 자원 점유하던 프로세스가 다른 자원 요구할 때
점유하던 모든 자원 반납하고
요구 자원 사용하기 위해 대기시킴

4. 순환대기(Circular wait)

: 각 프로세스는 순환적으로
다음 프로세스가 요구하고 자원 가짐

⇒ 각 자원에 고유 번호 할당해
순서대로 자원 요구하도록 함

◆ 프로듀서 컨슈머 (Producer and Consumer) (아직 정리 중)

✓ 사용 이유

- 프로듀서 컨슈머는 하나의 Buffer 공유

1. 시작) 프로듀서의 세마포어 = 1개, 컨슈머의 세마포어 = 0개
2. 생산된 아이템이 0개라면, 프로듀서는 세마포어를 P를 통해 소비하여 제작 시작
3. 프로듀서가 생산 마치면, 프로듀서의 세마포어를 V를 통해 1개 늘리는 것이 아닌
   컨슈머의 V를 통해 1개 늘려 컨슈머가 깨어나도록 함
4. 다음 프로듀서가 다시 아이템을 생산하려 해도 이미 프로듀서의 세마포어가 0개라
   생산을 하러 갈 수 없고 sleep 상태 들어감
5. 이때 컨슈머는 세마포어 수가 0개에서 1개 되었으니 아이템 소비
   그 후 컨슈머의 V가 아닌 프로듀서의 V를 해줘서 아이템 생산할 수 있게 함