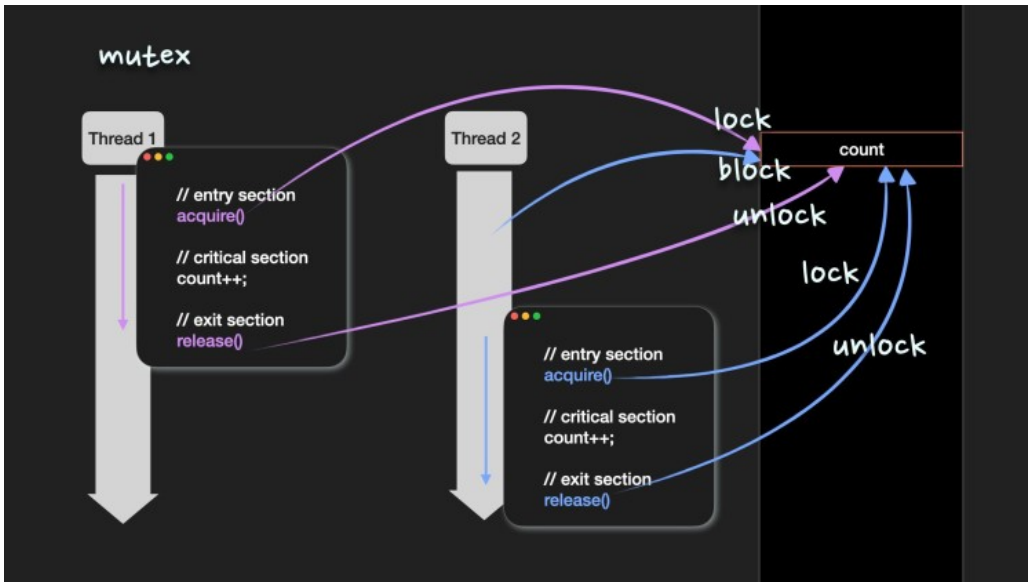
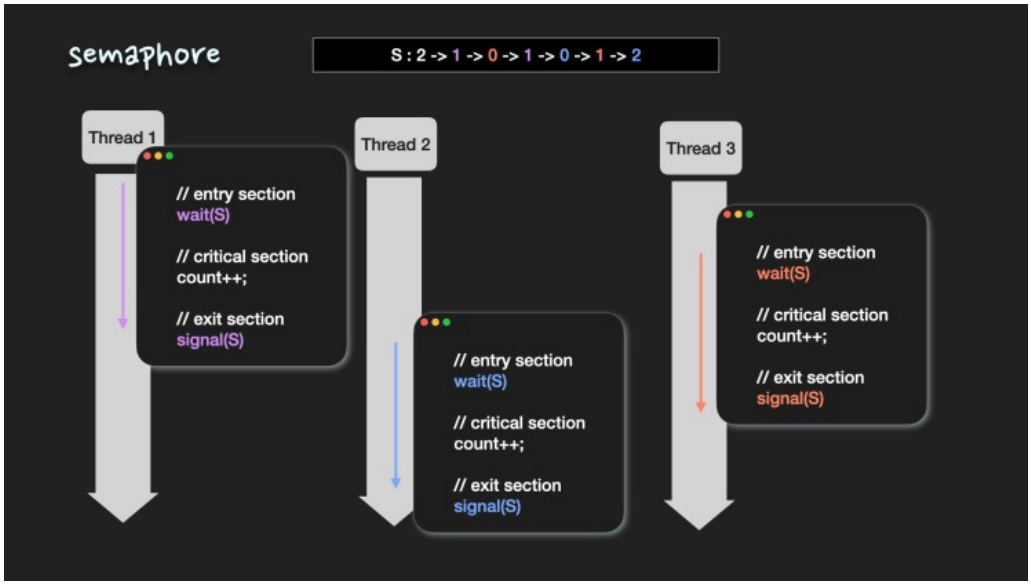
● Multi process/thread 환경에서 동기화 문제를 어떻게 해결하나?

▶ multi process/thread 환경에서는 서로 다른 thread가 메모리 영역을 공유하기 때문에, 둘 이상의 thread가 동일한 자원에 접근하고 조작하여 실행 결과가 접근이 발생한 순서에 따라 달라지는 경쟁상황에 의해서 동기화 문제가 발생할 수 있습니다. 동기화 문제를 해결하기 위해서는, 공유 데이터를 사용하는 코드 영역을 임계 영역(critical section)으로 지정해놓고, 쓰레드가 진행 중인 작업을 다른 쓰레드가 간섭하지 못하도록 동기화(synchronization) 시켜야 합니다. 동기화 기법으로는 대표적으로 mutex와 semaphore 기법이 있습니다. Mutex(mutual exclusion)란 1개의 쓰레드만이 임계 영역에 접근할 수 있도록 하여, 경쟁 상황(race condition)를 방지하는 기법으로, mutex lock을 사용합니다. 즉, process/thread는 임계영역에 들어가기 전에 entry section에서 acquire() 함수를 통해 반드시 lock을 획득해야 하고, 임계구역을 빠져나올 때는 exit section에서 release() 함수를 통해 lock을 반환해야 합니다. 공유 자원을 점유하는 thread가 lock을 획득하면, 다른 thread는 해당 쓰레드가 lock을 반환할 때까지 busy waiting을 하며 해당 임계 영역에 접근할 수 없는 것입니다. 이와 비교하여 Semaphore은 S개의 thread만이 공유 자원에 접근할 수 있도록 제어하는 동기화 기법으로, semaphore 변수 (세마포)S에 동시에 접근 가능한 process/thread의 갯수를 저장합니다. S가 0보다 크면 entry section에서 wait()함수를 통해 임계영역으로 들어갈 수 있고, 임계영역에 들어가면 S값을 1 감소시키고, 임계영역에서의 작업이 끝나고 exit하면 exit section에서 signal() 함수를 통해 S값을 1 증가시킵니다. 이때 S값이 0이 되면 다른 process/thread는 세마포 값이 0보다 커질 때까지 busy waiting이 걸려 임계영역으로 접근할 수 없습니다. 정리하자면, mutex는 오직 1개의 process/thread만이 공유 자원에 접근할 수 있고, semaphore는 세마포 변수의 값만큼의 process/thread들이 동시에 자원에 접근할 수 있습니다. semaphore 값이 0, 1만 가질 수 있는 경우 binary semaphore라고 하는데, 이는 mutex와 유사하게 작동합니다. 따라서 mutex는 binary semaphore라고 할 수 있습니다.





→ 자바에서 동기화 할 수 있는 방법은 자동으로 lock의 잠금과 해제가 관리되는 synchronized블럭을 사용하거나, ReentrantLock()과 같은 java.util.concurrent.locks패키지가 제공하는 lock클래스들을 이용하는 방법이 있습니다.

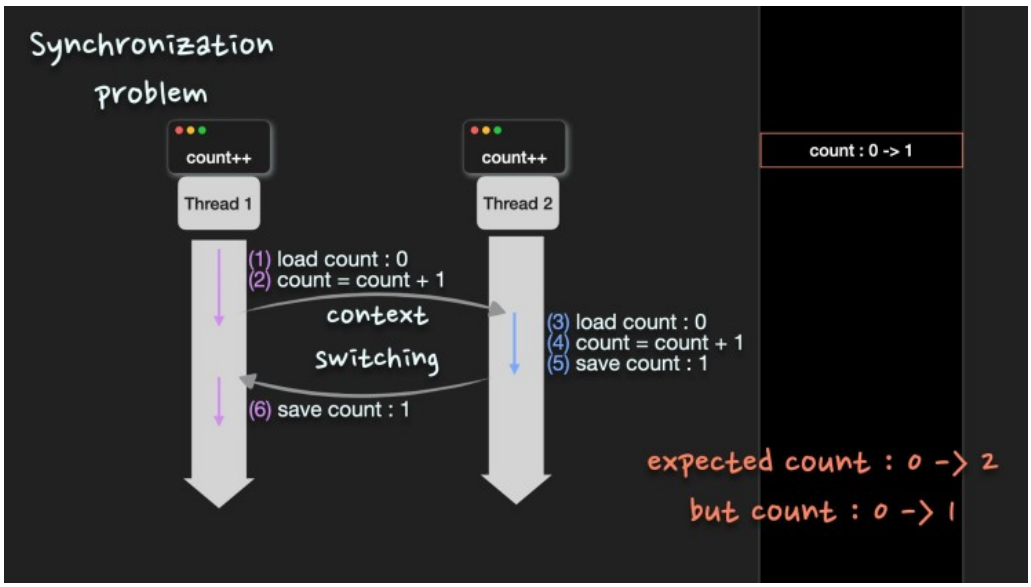
→ busy waiting은 다른 process/thread가 생상적으로 사용할 수 있는 CPU를 낭비한다는 단점이 있습니다. 자바에서 특정 쓰레드가 객체의 락을 가진 상태로 오랜 시간을 보내지 않도록 하는 것도 중요하기 때문에, 이를 개선하기 위해서 wait()와 notify()를 사용할 수 있습니다. 동기화된 임계 영역의 코드를 수행하다가 작업을 더 이상 진행할 상황이 아니면, 일단 wait()를 호출하여 쓰레드가 락을 반납하여 다른 쓰레드가 락을 얻어 해당 객체에 대한 작업을 수행할 수 있도록 하고, 나중에 작업을 진행할 수 있는 상황이 되면 notify()를 호출해서, 작업을 중단했던 쓰레드가 다시 락을 얻어 작업을 진행할 수 있게 하는 것입니다.

→ notify()가 호출 되었을 때, 어느 쓰레드가 통지를 받는지는 알 수 없습니다. 때문에 운이 나쁘면 특정 쓰레드는 계속 통지를 받지 못하고 오랫동안 기다리게 되는데, 이를 기아(starvation)현상이라고 합니다. 이 현상을 막으려면, notifyAll()을 사용해야 하는데, 이럴 경우 여러 쓰레드가 통지를 받아 lock을 얻기 위해 서로 경쟁하는 경쟁 상태(race condition)가 발생할 수 있습니다. 이를 개선하려면 수동으로 lock을 잠그고 해제하며 lock을 관리하는 ReentrantLock 클래스로부터 newCondition()을 호출하여 쓰레드 종류 별로 Condition 클래스의 인스턴스를 생성해서 wait(), notify()대신 Condition의 await(), signal()를 함께 사용하면, 쓰레드의 대기와 통지의 대상을 보다 명확하게 구분할 수 있어 기아와 경쟁 상태를 개선할 수 있습니다.

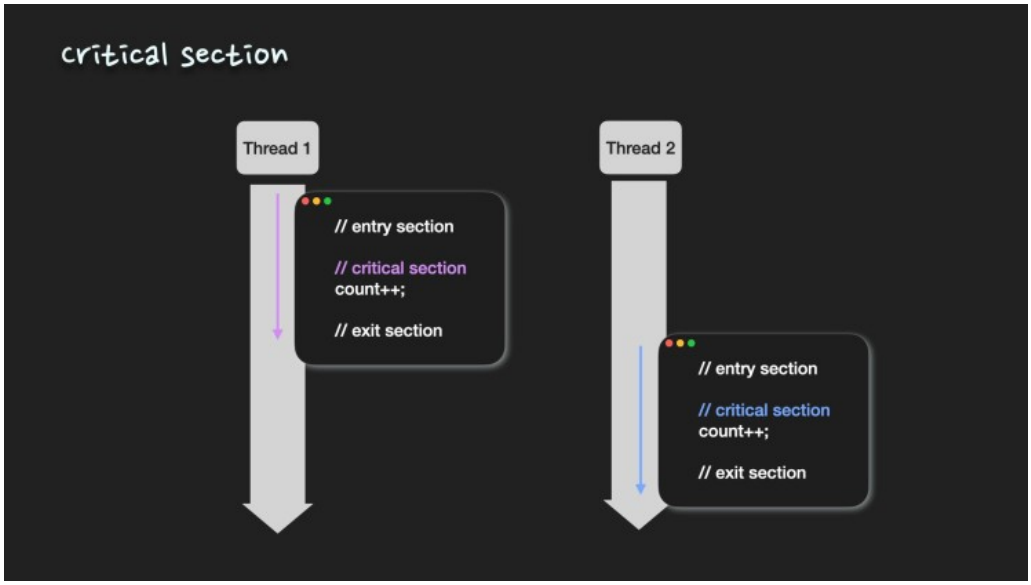
```
// Mutex - entry section에서는 acquire()함수가 lock을 획득하고 exit section에서는
// release()함수가 lock을 반환한다

acquire() // entry section

// critical section

release() // exit section


release() {
    available = true;
}

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

```
// Semaphore
wait(S) // entry section

// critical section

signal(S) // exit section


wait(S) {
    while (S ≤ 0); // busy wait
    S--;
}

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

● 동기화 문제: 동기화 문제란 서로 다든 thread가 메모리 영역을 공유하기 때문에 여러 thread가 동일한 자원에 동시에 접근하여 엉뚱한 값을 읽거나 수정하는 문제입니다. count++를 CPU 입장에서 분해해보면 3개의 atomic operations으로 나뉩니다. 1) count 변수의 값을 메모리로부터 CPU register로 가져옵니다. 2) count 변수의 값을 1 증가시킵니다. 3) 변경된 count 값을 메모리에 저장합니다. CPU는 atomic operation을 연산하게 됩니다. 따라서 count++을 하기 위해 3번의 연산을 하게 됩니다. 시분할 시스템으로 작동하는 multi process/multi thread 시스템에서, 두 개의 thread가 동일한 데이터인 count에 동시에 접근을 하여 조작을 하는 상황을 가정해보겠습니다. thread1에서도 count++를 하고, thread2에서도 count++를 한다면 그 실행 결과가 접근이 발생한 순서에 따라 달라질 수 있습니다. 이를 경쟁상황(race condition)이라고 합니다. 즉, 둘 이상의 thread가 동일한 자원에 접근하여 조작하고, 그 실행 결과가 접근이 발생한 순서에 따라 달라지는 경쟁상황에 의해서 동기화 문제가 발생할 수 있습니다. 경쟁 상황으로부터 보호하기 위해, 우리는 한 순간에 하나의 process/thread만 해당 자원에 접근하고 조작할 수 있도록 보장해야 합니다. 다시 말해서 process/thread들이 동기화되도록 할 필요가 있습니다.



● 임계영역(critical section): 공유 데이터를 사용하는 코드 영역입니다. 한 process/thread가 자신의 임계구역에서 수행하는 동안에는 다든 process/thread들은 그들의 임계구역에 들어갈 수 없어야 합니다. 즉, 임계영역 내의 코드는 원자적으로(atomically) 실행이 되어야 합니다. 원자적으로 실행 되기 위해서 각각의 process/thread는 자신의 임계구역으로 진입하려면, 진입 허가를 요청해야합니다. 이 부분을 entry section이라고 하고, 진입이 허가되면 임계영역을 실행할 수 있습니다. 임계영역이 끝나고 나면 exit section으로 퇴출을 하게 됩니다. 임계영역의 원자성을 보장하여 process/thread들이 동기화되도록 할 수 있습니다. 동기화 방법은 대표적으로 Mutex와 Semaphore가 있습니다.



Resources: inflearn 개발남 노씨