1. 멀티 스레드/ 프로세스 환경에서 여러 실행 흐름이 하나의 작업을 처리하는 경우 race condition 이 발생할 수 있는데 이들은 왜 일어나는 것이고 이를 해결하기위한 기법과 장단점에 대해서 설명하시오. (3개 이상 해결 방법 서술)
2. Java에서 여러 스레드를 사용해 하나의 연산을 처리하려고 할 때 Thread Safe하게 연산을 수행할 수 있도록 하는 방법에 대해서 3가지 이상 조사하시오.

   (직접 멀티 스레드를 활용해서 구현해보는 것을 추천)

Race Condition 이란?

Race Condition이란 두 개 이상의 프로세스가 공통 자원을 병행적으로(concurrently) 읽거나 쓰는 동작을 할 때, 공용 데이터에 대한 접근이 어떤 순서에 따라 이루어졌는지에 따라 그 실행 결과가 같지 않고 달라지는 상황을 말한다. 간단히 말하면 경쟁하는 상태, 즉 두 개의 스레드가 하나의 자원을 놓고 서로 사용하려고 경쟁하는 상황을 말한다.

임계 구역(Critical Section)

임계 구역은 프로세스의 코드 중 공유 자원 접근을 수행하는 코드 영역. 임계구역 문제를 적절히 처리하지 않으면, 공유 자원에 둘 이상의 프로세스가 동시에 접근하여 경쟁 상태가 발생할 수 있다.

경쟁 프로세스의 경우, 세 가지 제어 문제에 직면(상호 배제, 교착상태, 기아 상태)

상호 배제(Mutual Exclusion)

Race Condition을 막기 위해서는 두 개 이상의 프로세스가 공용 데이터에 동시에 접근하는 것을 막아야 한다. 즉, 한 프로세스가 공용 데이터를 사용하고 있으면 그 자원을 사용하지 못하도록 막거나, 다른 프로세스가 그 자원을 사용하지 못하도록 막으면 이 문제를 피할 수 있다. 이는 임계 구역(Critical Section)으로 불리는 코드 영역에 의해 구현된다. 이것을 상호 배제라고 부른다.

교착상태(Deadlock)

그러나 위와 같은 상호 배제를 시행하면 추가적인 제어 문제가 발생한다. 하나는 교착상태 즉 여기서 말하는 Deadlock. 프로세스가 각자 프로그램을 실행하기 위해 두 자원 모두에 엑세스 해야 한다고 가정할 때 프로세스는 두 자원 모두를 필요로 하므로 필요한 두 리소스를 사용하여 프로그램을 수행할 때까지 이미 소유한 리소스를 해제하지 않는다. 이러한 상황에서 두 프로세스는 교착 상태에 빠지게 되는 문제가 발생할 수 있다.

진행(Progress)

임계 구역을 실행 중인 프로세스가 없을 때, 임계 구역을 실행하고자 하는 프로세스가 여러 개 있다면 그 실행 순서를 정해주어야 한다.

유한 대기(Bounded Waiting)

한 프로세스가 임계 구역으로 들어가면, 다른 프로세스들은 그 프로세스가 임계구역에서 나올 때까지 기다려야 한다. 하지만 이러한 대기 시간이 너무 길어져선 안되고, 어느 정도의 일정 한도가 있어야 한다는 것이 유한 대기 조건이다.

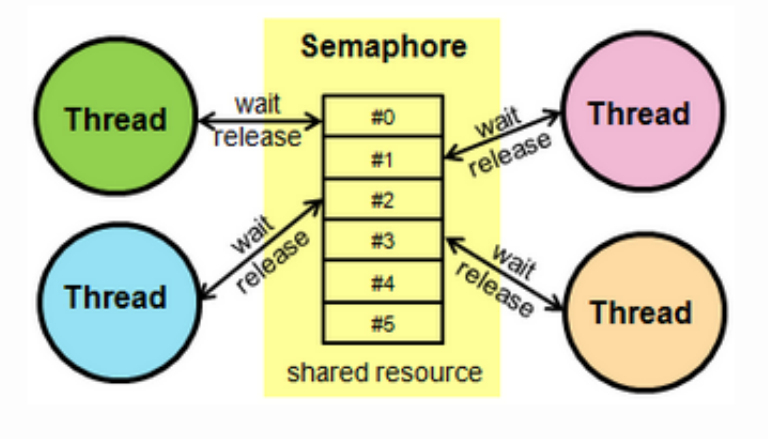
기아 상태(Starvation)

이 제어 문제는 '기아 상태' 라고도 한다. 이러한 문제는 프로세스들이 더 이상 진행을 하지 못하고 영구적으로 블록되어 있는 상태로, 시스템 자원에 대한 경쟁 도중에 발생할 수 있고 프로세스 간의 통신 과정에도 발생할 수 있는 문제이다. 두 개 이상의 작업이 서로 상대방의 작업이 끝나기만을 기다리고 있기 때문에 결과적으로는 아무것도 완료되지 못하는 상태가 되게 된다.

이러한 문제가 발생하지 않도록, OS는 다른 프로세스의 의도하지 않은 간섭으로부터 각 프로세스의 데이터 및 물리적 자원을 보호해야 하며 여기에 메모리, 파일 및 I/O 장치와 관련된 내용이 포함된다. 그리고 프로세스에서 수행하는 내용과 프로세스가 생성하는 결과는, 다른 동시 프로세스의 실행 속도와 무관, 즉 기능과 결과는 서로 독립적이어야 한다.

경쟁 조건을 예방하기 위한 방법

세마포(Semaphore)



공유된 자원의 데이터를 여러 프로세스가 접근하는 것을 막는 것이다. 또한 세마포어는 리소스의 상태를 나타내는 간단한 카운터라고 할 수 있는데, 일반적으로 비교적 긴 시간을 확보하는 리소스에 대해 이용하게 되며, 운영체제의 리소스를 경쟁적으로 사용하는 다중 프로세스에서 행동을 조정하거나 동기화 시키는 기술이다. 다시 말해서 하나의 스레드만 들어가게 할 수도 있고 여러 개의 스레드가 들어가게 할 수 있다(뮤텍스와의 차이점). 세마포는 락을 걸지 않은 스레드도 Signal을 보내 락을 해제할 수 있다는 점에서, wait 함수를 호출한 스레드만이 signal 함수를 호출할 수 있는 뮤텍스와 다르다. 세마포는 동기화를 위해 wait와 signal이라는 2개의 atomic operations를 사용한다. wait를 호출하면 세마포의 카운트를 1 줄이고, 세마포의 카운트가 0보다 작거나 같아질 경우에 락이 실행된다. Semaphore의 cnt 값은 지정한 만큼 한 공유 자원에 대해 여러 스레드가 돌게 할 수도 있다.

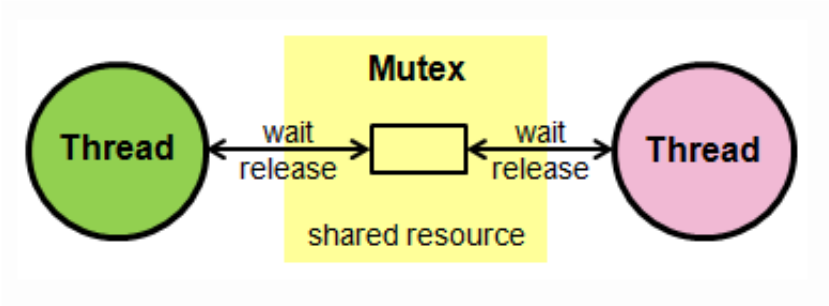
```
struct semaphore {
    int count;
    queueType queue;
};

void semWait (semaphore s) {
    s.count--;
    if (s.count <= 0) {
      // 락이 걸리고 공유 자원에 접근할 수 없음
    }
}

void semSignal (semaphore s) {
    s.count++;
    if (s.count <= 0) {
      // 아직 락에 걸려 대기중인 프로세스가 있음
```

```
    }
}
```

세마포어의 카운트가 0보다 작거나 같아져 동기화가 실행된 상황에, 다른 쓰레드가 signal 함수를 호출하면 세마포어의 카운트가 1증가하고, 해당 쓰레드는 락에서 나올 수 있다.

세마포어는 크게 Counting Semaphores, Binary Semaphore 2종류가 있다. 카운팅 세마포어는 세마포어의 카운트가 양의 정수값을 가지며, 설정한 값만큼 쓰레드를 허용하고 그 이상의 쓰레드가 자원에 접근하면 락이 실행된다. 바이너리 세마포어는 세마포어의 카운트가 1이며 Mutex처럼 사용될 수 있다.

뮤텍스(Mutex)



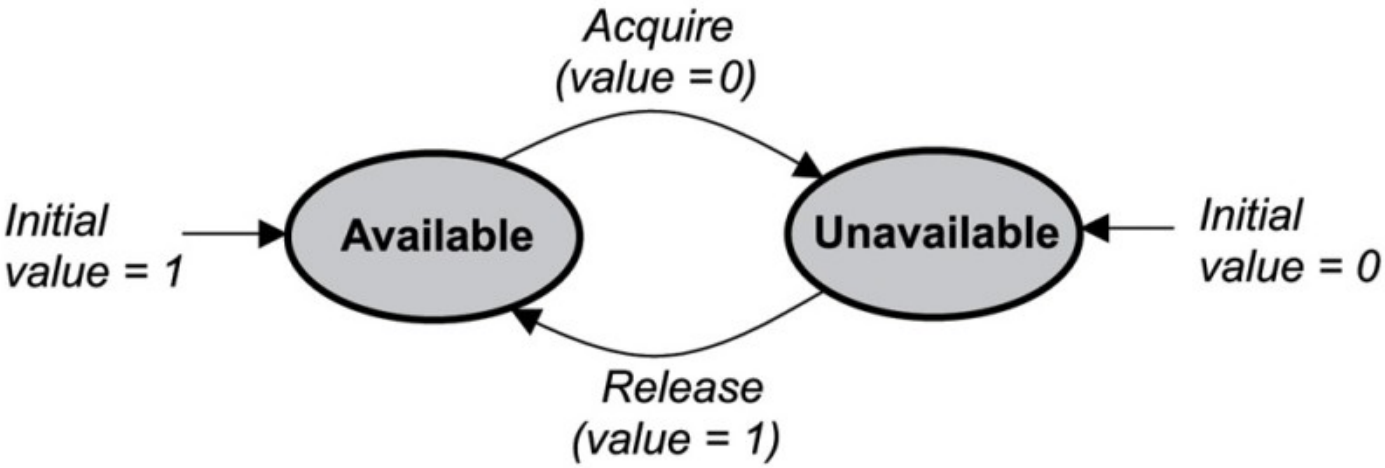
공유된 자원의 데이터를 여러 스레드가 접근하는 것을 막는 방법이다. 즉, Critical Section(각 프로세스에서 공유 데이터를 액세스하는 프로그램 코드 부분)을 가진 스레드들의 Running time이 서로 겹치지 않게 각각 단독으로 실행되게 하는 기술이다. 다중 프로세스들이 공유 리소스에 대한 접근을 조율하기 위해 locking과 unlocking을 사용하는데, 다시 말해서 상호배제를 함으로써 두 스레드가 사용할 수 없다는 뜻이다.

뮤텍스는 자원에 대한 접근을 동기화하기 위해 사용되는 상호배제 기술이다. 이것은 프로그램이 시작될 때 고유한 이름으로 생성된다. 뮤텍스는 Locking 메커니즘으로 오직 하나의 스레드만이 동일한 시점에 뮤텍스를 얻어 임계 영역에 들어올 수 있다. 그리고 오직 이 스레드만이 임계영역에서 나갈 때 뮤텍스를 해제할 수 있다.

```
wait (mutex);
…..
Critical Section
…..
signal (mutex);
```

락을 얻은 쓰레드만이 임계 영역을 나갈 때 락을 해제해줄 수 있다. 이러한 이유는 뮤텍스가 1개의 락만을 갖는 Locking 메커니즘이기 때문

이진 세마포



[원본 코드]

```
class SharedData {
  int balance; // 공유 자원 변수
  void add() { balance++; }
  void sub() { balance--; }
  int getBalance() { return balance; }
}
```

[이진 세마포 적용 코드]

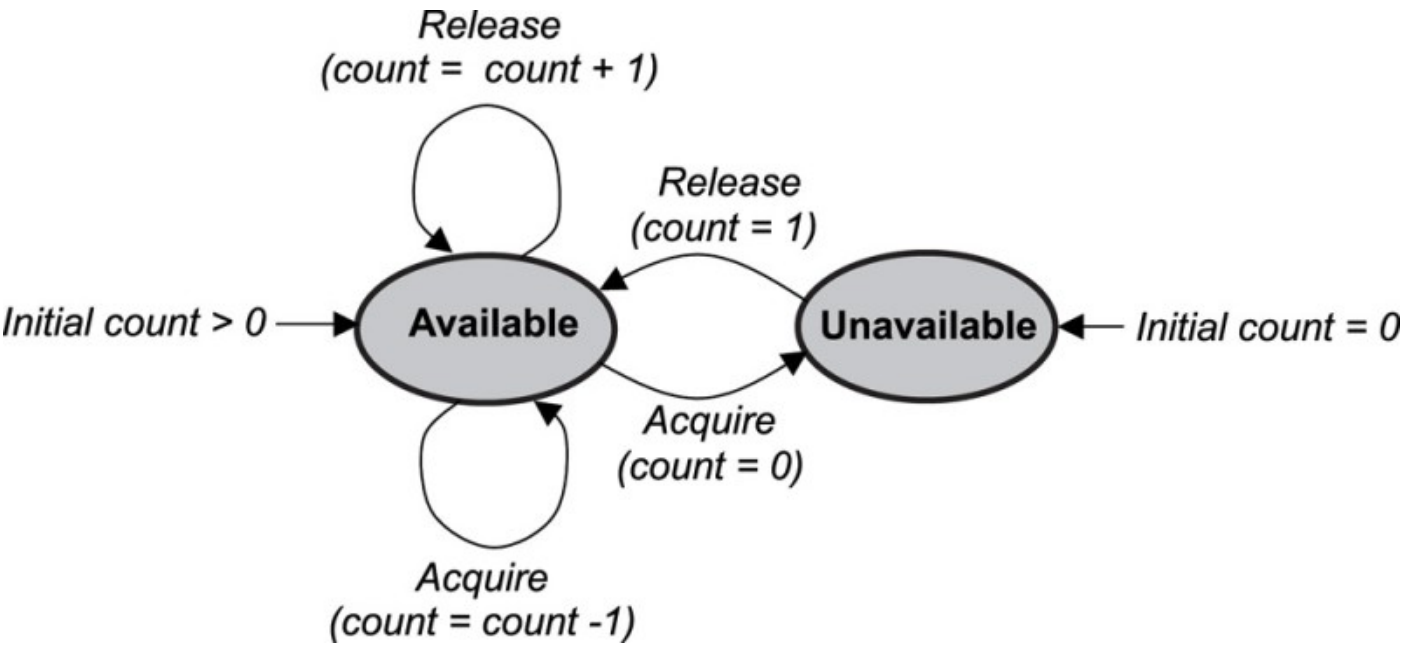
```
import java.util.concurrent.Semaphore;

class SharedData {
  int balance; // 공유 자원 변수
  Semaphore sem; // 세마포어 변수

  SharedData() { sem = new Semaphore(1); } // 세마포어 변수 1로 초기화

  void add() {
    try {
      sem.acquire(); // P
    } catch (InterruptedException e) {}
    balance++; // 임계구역
    sem.release(); // V
  }
```

```
  void sub() {
    try {
      sem.acquire(); // P
    } catch (InterruptedException e) {}
    balance--; // 임계구역
    sem.release(); // V
  }

  int getBalance() { return balance; }
}
```

세마포 변수(sem)을 1로 초기화하고 각 임계구역 전후로 P와 V를 놓아주면 임계구역 진입 전 sem 값을 1 감소시키고 임계구역에 진입한다. sem 값이 0으로 변한 상태에선 다른 스레드가 sem.acquire() 부분에 도달해도 다음 코드로 실행될 수 없기 때문에, 임계구역에 대한 상호 배타성이 보장된다. 앞서 임계구역에 있던 스레드가 sem.release()를 만나면 다시 sem 값이 1로 증가하고, 그때부턴 대기 중이던 다른 스레드가 sem.acquire() 다음 줄로 넘어갈 수 있게 된다. 이런 식으로 0 또는 1 값을 가지는 세마포를 이진 세마포라고 한다.

대신 이진 세마포는 상호 배타만 보장해주지 순서는 보장하지 않는다. 프로세스의 실행 순서는 그때그때 context switching 상황마다 매번 달라지며, 맨 처음 무조건 add 또는 sub로 시작하게끔 설계하고 싶다면 2번의 코드에 세마포를 하나 더 추가해야 한다.

[카운팅 세마포]



카운팅 세마포라고 하는 이유는 세마포어 변수가 0과 1로 한정되지 않고, 다른 양의 정수 값을 가질 수도 있기 때문이다. 예를 들어 아직 context switching 전이어서 add가 실행된 뒤 또 add가 실행되는 경우를 들 수 있다. 이 경우, release는 2번 호출되기 때문에 sem2값도 두번 더해진다. 따라서 하단의 카운팅 세마포어 다이어그램에선 available state의 initial count를 =0이 아닌 >0 으로 표기하고 있다. 이는 이진 세마포어 다이어그램과 크게 차이가 나는 부분이다.

```
import java.util.concurrent.Semaphore;

class SharedData {
  int balance; // 공유 자원 변수
  Semaphore sem, sem2; // 세마포어 변수 추가

  SharedData() {
    sem = new Semaphore(1);
    sem2 = new Semaphore(0); // 추가된 변수는 0으로 초기화
  }

  void add() {
    try {
      sem.acquire(); // P
    } catch (InterruptedException e) {}
    balance++; // 임계구역
    sem.release(); // V
    sem2.release(); // V2
  }

  void sub() {
    try {
      sem2.acquire(); // P2
      sem.acquire(); // P
    } catch (InterruptedException e) {}
    balance--; // 임계구역
    sem.release(); // V
  }

  int getBalance() { return balance; }
}
```

우선 sem2의 initial count는 0이기 때문에 unavailable state로 시작한다. 이 상태에서는 아무리 sem2.acquire()로 문을 두들겨도 available state로 넘어갈 수 있는 열쇠가 생기지 않는다. 이 열쇠는 sem2.release()로만 확보가 될 수 있는데, 코드에서 release를 하는 녀석은 add 함수이며, 그것도 자신의 임계구역을 다 넘어온 뒤 release 해주고 있다. 따라서, 상단의 코드는 무조건 add부터 실행될 것이고, 만약 sub가 release의 주도권을 갖게끔 코드를 수정한다면 반드시 sub부터 시작되는 프로그램이 될 것이다.

Thread-Safe란?

다른 스레드들이 잘못된 동작을 드러내거나 예측할 수 없는 결과를 생성하지 않고 동일한 리소스에 액세스할 수 있는 프로그래밍 방법론

Stateless Implementations

대부분의 경우 다중 스레드 응용 프로그램의 오류들은 여러 스레드들 간에 상태를 잘못 공유한 결과이다.

Stateless implementations을 이용(?) -> 상태 비저장이 무엇을 의미하는가?

Immutable Implementations

Thread-Local Fields

Synchronized Collections

Concurrent Collections

https://www.baeldung.com/java-thread-safety