

1. 동기화 비용

멀티 스레드는 공유된 데이터를 읽고 쓰는 과정에서 동기화 문제가 발생할 수 있다. 그리고 동기화 문제를 해결하기 위해 뮤텝스, 세마포어 등의 기법을 사용한다. 이때 뮤텝스, 세마포어는 데이터에 접근 할 때 락을 획득하고 사용 후에는 락을 해제하는 작업을 수행한다. 이렇게 락을 획득하고 해제하는데 추가적인 시간이 소요된다.

따라서 스레드가 많아진다면 당연히 공유된 데이터에 접근하는 스레드가 많아질 것이고 동기화 기법으로 인한 추가적인 시간이 더 소요될 것이다.

2. 거짓 공유 (False Sharing)

먼저 다음과 같이 A 라는 배열을 공유한다고 해보자.

```
public interface FalseSharingThread extends Runnable {
    int[] A = new int[100];
}
```

그리고 위 인터페이스를 구현한 스레드 2개를 정의해보자.

```
public class FalseSharingThread1 implements FalseSharingThread {
    @Override
    public void run() {
        for (int i = 0; i < 100000 * 10000; i++) {
            A[0] = i;
        }
    }
}


public class FalseSharingThread2 implements FalseSharingThread {
    @Override
    public void run() {
        for (int i = 0; i < 10000 * 100000; i++) {
            A[5] = i * 2;
        }
    }
}
```

FalseSharingThread1 는 인덱스 0을 수정하고 FalseSharingThread2 는 인덱스 5를 수정하는 간단한 작업을 수행한다. 이제 이 2개의 스레드가 동시에 작업을 수행할때 메인 메모리와 캐시 메모리에서는 어떤 과정이 일어날까?

먼저 스레드간에 공유되는 스택 영역에는 배열 A가 다음과 같이 적재될 것이다.

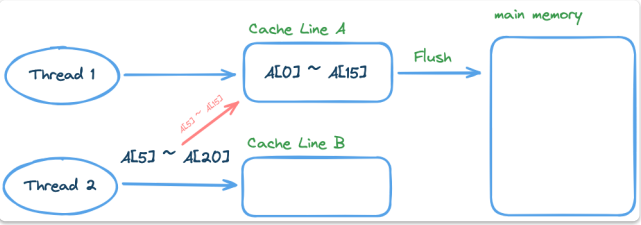


그리고 FalseSharingThread1 가 실행되고 배열 A에 접근할 때 사용 가능한만큼 캐시라인에 적재한다. (CPU에따라 캐시라인의 크기는 32 또는 64 또는 128 바이트를 가지기 때문에 현재 예시에서는 64 바이트를 가진다고 가정한 후 진행한다.) 그러면 FalseSharingThread1 가 A[0]에 접근하기때문에 A[15]까지의 데이터 (4byte * 16)가 하나의 캐시라인에 적재될 것이다. 그리고 A[0]의 값을 수정하는 작업을 수행할 것이다.

 **Note**

캐시 메모리는 자주 사용하는 데이터를 저장해놓는다.

FalseSharingThread2 도 마찬가지로 A[5] ~ A[20]를 캐시라인에 적재되어야 할 것이다. 그런데 A[5] ~ A[15]는 이미 FalseSharingThread1 에서 이미 캐시라인에 적재했다. 그래서 실제로 A[5] ~ A[15]는 데이터 변경이 일어나지않았지만 CPU는 변경되었을 수도 있다고 판단하여 A[0] ~ A[15]를 메인 메모리로 갱신해준다. 그런 다음 A[5] ~ A[20]를 캐시라인에 적재한다.



이렇게 스레드마다 적재되는 캐시라인을 갱신하지 않아도 문제되지 않지만 CPU는 이러한 사실을 무시하고 무조건 갱신하는 것을 메모리 **거짓 공유(False Sharing)** 라고 한다.

실제로 위에서 구현한 FalseSharingThread1 와 FalseSharingThread2 를 실행해보면 약 70 ~ 80ms가 걸린다.

```
public class FalseSharing {
    public static void main(String[] args) {
        List<Thread> threads = new ArrayList<>();
        Thread thread1 = new Thread(new FalseSharingThread1());
        Thread thread2 = new Thread(new FalseSharingThread2());
        thread1.start();
        thread2.start();
    }
}
```

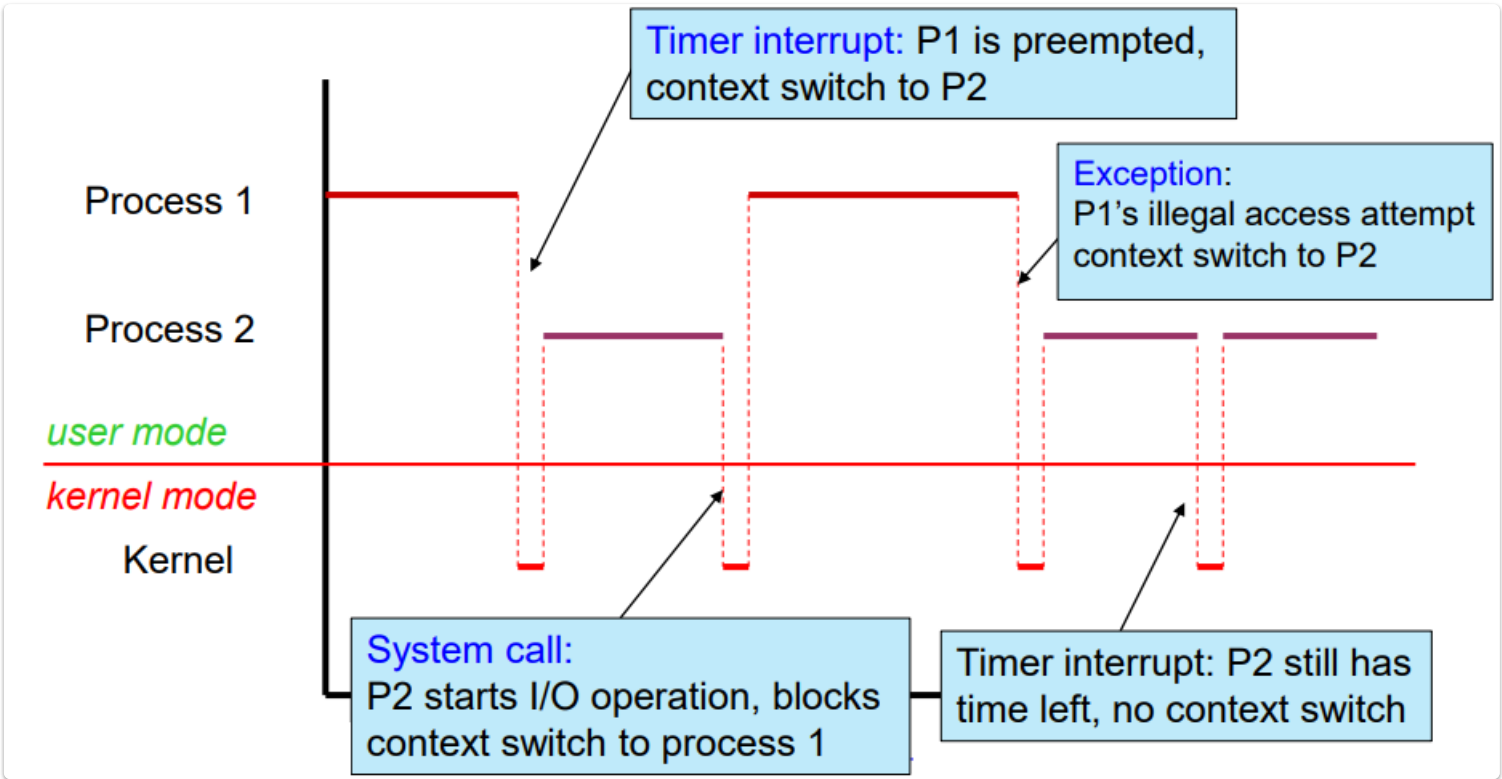
```
threads.add(thread1);
threads.add(thread2);
long beforeTime = System.currentTimeMillis();
for(int i=0; i<threads.size(); i++) {
    Thread t = threads.get(i);
    try {
        t.join();
    }catch(Exception e) {
    }
}
long afterTime = System.currentTimeMillis();
long secDiffTime = (afterTime - beforeTime);
System.out.printf("소요시간: %d", secDiffTime);
}
```

거짓 공유가 일어나지 않도록 구현하고 돌려보면 얼마나 걸릴까? 각 스레드가 참고하는 메모리의 위치가 겹치지 않도록 `FalseSharingThread2` 에서 `A[50]`을 수정하도록 하고 실행해봤다. 결과는 약 30 ~ 40ms가 걸리는 것을 확인할 수 있었다.

이러한 사실을 통해 스레드가 더 많아진다면 거짓 공유가 일어날 가능성이 많아질 것이고 이로 인한 성능 이슈가 발생할 것이라는 예상이 가능하다.

3. 컨텍스트 스위칭

멀티 프로세스 대신 멀티 스레드를 사용하는 이유 중 하나가 프로세스의 컨텍스트 스위칭 보다 스레드의 컨텍스트 스위칭 비용이 적다는 것이다. 구체적으로 OS로부터 자원을 할당받을 때 프로세스는 스레드보다 더 많은 자원을 할당받는다. 그리고 컨텍스트 스위칭을 하면 스레드보다 많은 데이터를 PCB에 저장하고 다른 프로세스로 변경하게된다. 하지만 스레드이든 프로세스든 컨텍스트 스위칭을 하는 과정에서 오버헤드가 발생한다.



오버헤드 또한 프로세스보다 스레드가 더 적지만 스레드가 많아질 수록 컨텍스트 스위칭 횟수가 증가할 것이고 오버헤드도 많아지게 될 것이다.

Note

컨텍스트 스위칭 오버헤드는 CPU가 현재 프로세스나 스레드의 상태를 저장하고 다른 프로세스나 스레드로 전환할 때 발생하는 비용이다. 그리고 이 과정에서 시간과 자원을 소모하게되어 성능에 영향을 미친다.

4. 리소스 낭비

스레드가 많더라도 실제 서비스에서는 한두개만 사용할 경우 나머지 놓고 있는 스레드들은 자원을 불필요하게 차지한다. 스레드가 아무런 작업을 수행하지 않더라도 자원을 소비하기 때문에 낭비가 발생하게된다.

5. 애플리케이션의 동작

만약 애플리케이션이 순차적으로 실행되어야하는 특징을 가진다면 멀티 스레드를 사용한다고해도 싱글 스레드와 비교하여 더 나은 성능을 기대하기 어려울 것이다.

정리

스레드가 많아질수록 다음과 같은 문제가 발생할 수 있다.

- 공유된 데이터에 접근하는 스레드가 많아지면서 동기화 기술 비용 발생
- 거짓 공유가 발생할 가능성이 많아지면서 성능 저하가 발생
- 컨텍스트 스위치 과정에서 오버헤드 발생
- 아무런 작업을 하지 않는 스레드가 발생하면서 리소스 낭비
- 순차적으로 실행되어야하는 애플리케이션에서는 스레드가 많아도 성능이 비슷

[reference]
<https://blog.naver.com/hermet/68290454>
<https://inpa.tistory.com/entry/%F0%9F%91%A9%E2%80%8D%F0%9F%92%BB-ls-more-threads-always-better>