2주차

- 컨텍스트 스위치 비용 (하드웨어적으로) - 10분
- Atomic (CAS), Syncronized (lock), voilate, (FolkJoinPool, BlockingDeque, java.util.concurrent) - 10분
- Tomcat 네트워크 요청을 받아서, 스레드를 할당받고, 이게 스프링까지 넘어와서 어떤식으로 스레드가 처리되는지? - 10분

컨텍스트 스위치

하드웨어적인 측면에서 **컨텍스트 스위칭**이란, CPU가 한 프로세스 또는 스레드의 실행 상태를 저장하고 다른 프로세스나 스레드의 실행 상태를 복원하여 작업을 전환하는 과정을 의미합니다. CPU는 한 번에 하나의 작업만 수행할 수 있기 때문에, 여러 작업을 동시에 진행하려면 작업 간 빠르게 전환하여 마치 병렬로 처리되는 것처럼 보이게 만듭니다. 이 과정에서 필요한 정보 저장 및 복원은 CPU의 하드웨어와 메모리 간의 상호작용을 통해 이루어집니다.

컨텍스트 스위칭의 주요 하드웨어적 단계는 다음과 같습니다:

1. CPU 레지스터 저장 및 복원

- 프로세스나 스레드가 실행 중인 동안 CPU는 프로그램 카운터, 스택 포인터, 레지스터 등 다양한 상태 정보를 사용합니다. 다른 작업으로 전환할 때, 현재 작업의 레지스터 값을 메모리나 특정 저장 위치에 보관하고 새로운 작업의 레지스터 상태를 복원하는 과정이 필요합니다.

2. 프로그램 카운터 전환

- 프로그램 카운터는 현재 실행 중인 명령어의 위치를 가리키는 CPU 레지스터입니다. 컨텍스트 스위칭 시 프로그램 카운터를 새로운 작업의 시작 위치로 업데이트해야 합니다. 이는 CPU가 다른 프로세스의 명령어를 실행하기 위한 필수 단계입니다.

3. TLB 플러시 (Translation Lookaside Buffer)

- TLB는 가상 주소를 물리 주소로 변환하는 데 사용되는 캐시입니다. 프로세스가 전환될 때마다, 이전 프로세스의 TLB 항목을 비우고 새 프로세스의 항목으로 갱신해야 합니다. TLB 플러시는 메모리 접근 성능에 영향을 줄 수 있습니다.

4. 파이프라인 플러시

- 현대 CPU는 여러 명령어를 동시에 처리하는 **파이프라이닝** 방식을 사용합니다. 그러나 컨텍스트 스위칭이 발생하면 현재 작업의 파이프라인에 쌓여있던 명령어가 무효화(플러시)되고, 새로운 작업의 명령어로

채워져야 합니다. 이 과정에서 파이프라인이 잠시 동안 비워지기 때문에, CPU의 효율이 떨어질 수 있습니다.

5. 캐시 오염(Cache Pollution)

- CPU 캐시는 자주 사용하는 데이터와 명령어를 저장해두는 고속 메모리입니다. 컨텍스트 스위칭 시 캐시에 있던 데이터가 새로운 작업의 데이터로 덮어씌워질 수 있으며, 이는 캐시 미스를 유발해 메모리 접근 지연을 증가시킵니다.

이러한 요소들이 모여 CPU의 자원을 빠르게 저장하고 불러오는 과정을 돕지만, 컨텍스트 스위칭이 너무 자주 발생하면 하드웨어적 오버헤드로 인해 시스템 전체 성능이 저하될 수 있습니다.

컨텍스트 스위칭의 비용

1. 레지스터 저장 및 복원

- 컨텍스트 스위치에서 가장 기본적인 단계는 현재 실행 중인 프로세스(혹은 스레드)의 레지스터 값을 저장하는 것입니다. CPU 레지스터(예: 프로그램 카운터, 스택 포인터 등)의 값을 메모리에 저장하고, 새 프로세스의 레지스터 값을 복원해야 합니다. 이 과정은 CPU의 속도와 메모리의 대역폭에 따라 비용이 달라집니다.

2. 캐시 미스(Cache Miss)

- CPU는 프로세스가 자주 사용하는 데이터를 캐시에 저장해두는데, 컨텍스트 스위치 시 캐시에 저장된 데이터가 다른 프로세스의 데이터로 교체되면서 캐시 미스가 발생할 수 있습니다. 캐시 미스는 CPU가 필요한 데이터를 메인 메모리에서 다시 로드하게 만들어 성능 저하를 유발합니다.

3. TLB(Translation Lookaside Buffer) 플러시

- 가상 메모리를 사용하는 경우, 프로세스마다 다른 가상 주소 매핑을 가지므로, TLB라는 고속 메모리 캐시를 사용해 가상 주소를 물리 주소로 변환합니다. 컨텍스트 스위치 시 다른 프로세스의 TLB 항목으로 교체해야 하며, 이로 인해 이전 프로세스의 TLB 항목을 플러시하고 다시 채우는 오버헤드가 발생합니다.

4. 파이프라인 플러시

- 현대의 CPU는 명령어 파이프라이닝을 사용하여 여러 명령어를 동시에 실행합니다. 컨텍스트 스위치가 발생하면 현재 파이프라인에 있는 명령어를 모두 비워야 하므로, 파이프라인이 플러시되고 이를 다시 채우는 시간이 필요합니다.

5. 상호 배타적 리소스 관리 및 동기화 비용

- 여러 스레드나 프로세스가 같은 리소스에 접근할 경우 동기화가 필요하며, 이 과정에서 락(Lock)이나 세마포어와 같은 동기화 메커니즘이 사용됩니다. 특히 커널이 이를 처리할 때 추가적인 비용이 발생합니다.

Atomic (CAS), Syncronized (lock), voilate, (FolkJoinPool, BlockingDeque, java.util.concurrent)

여기서는 Java에서 멀티스레딩 및 동기화 메커니즘에 사용되는 주요 개념들을 다룹니다. 이 중에서도 **Atomic (CAS)**, **synchronized (lock)**, **volatile**, **ForkJoinPool**, **BlockingDeque**, 그리고 **java.util.concurrent 패키지**의 기능을 살펴보겠습니다.

1. Atomic (CAS - Compare And Swap)

Atomic 클래스들은 `java.util.concurrent.atomic` 패키지에 포함되어 있으며, 단일 변수의 원자적(atomic) 연산을 제공합니다. **CAS(Compare-And-Swap)**은 원자성을 유지하기 위해 사용되는 연산으로, 다음과 같이 동작합니다:

1. 현재 변수의 값과 예상 값이 동일하면 새 값으로 교체합니다.
2. 그렇지 않으면 교체하지 않고, 대신 현재 값만 반환합니다.

CAS는 비교와 대체가 한 번에 이루어지기 때문에 데이터 레이스를 방지할 수 있습니다. 하드웨어 레벨에서 CAS 연산을 지원하므로 빠르고 효율적이며, **lock-free** 방식의 병렬 처리가 가능합니다.

예시

- **AtomicInteger**: 정수형 변수를 원자적으로 업데이트
- **AtomicReference**: 객체 참조를 원자적으로 업데이트

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
    private AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();  // CAS를 사용해 원자적 증가
    }

    public static void main(String[] args) {
        AtomicExample example = new AtomicExample();
        example.increment();
        System.out.println(example.counter.get()); // 결과: 1
```

```
    }
}
```

2. Synchronized (lock)

Synchronized 키워드는 특정 메서드나 코드 블록에 대한 동기화를 제공합니다. `synchronized`를 사용하면, 여러 스레드가 동시에 해당 블록에 접근하지 못하게 하여 **상호 배타적**으로 실행할 수 있습니다. Java에서 `synchronized`는 객체 수준의 락을 사용해 임계 구역(critical section)을 보호합니다.

사용 예시

```
public class SynchronizedExample {
    private int counter = 0;

    public synchronized void increment() {  // 메서드 동기화
        counter++;
    }

    public void decrement() {
        synchronized (this) {  // 블록 동기화
            counter--;
        }
    }
}
```

`synchronized`의 단점으로는 스레드가 임계 구역에서 기다려야 할 때 성능이 저하될 수 있다는 점이 있습니다.

3. Volatile

`volatile` 키워드는 **메모리 가시성** 문제를 해결하는 데 사용됩니다. 일반적으로 각 스레드는 CPU 캐시를 통해 데이터를 읽고 쓰기 때문에, 메모리에 대한 변경 사항이 다른 스레드에 즉시 반영되지 않을 수 있습니다. `volatile`을 사용하면 해당 변수가 **메인 메모리에 직접 쓰여지고 읽히게 되어, 모든 스레드가 변수의 최신 값을 즉시 볼 수 있게 합니다.**

사용 예시

```
public class VolatileExample {
    private volatile boolean isActive = true;

    public void stop() {
        isActive = false;
    }
```

```
    public void doWork() {
        while (isActive) {
            // 작업 실행
        }
    }
}
```

`volatile`은 동기화를 대체하지 못하며, 복잡한 작업보다는 상태 플래그와 같은 단순한 변수의 가시성 제어에 유용합니다.

4. ForkJoinPool

ForkJoinPool은 Java의 병렬 프로그래밍을 지원하기 위해 `java.util.concurrent` 패키지에 추가된 프레임워크로, **작업을 작은 단위로 분할(fork)하고 병렬로 실행한 후 합병(join)**하는 방식을 사용합니다. 특히 재귀적으로 분할 가능한 작업을 병렬로 처리하는 데 유리합니다.

ForkJoinPool은 `RecursiveTask`와 `RecursiveAction` 클래스를 사용해 작업을 정의하며, `invoke()` 메서드로 작업을 실행합니다.

예시

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class ForkJoinExample extends RecursiveTask<Integer> {
    private int start, end;

    public ForkJoinExample(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= 10) {
            int sum = 0;
            for (int i = start; i <= end; i++) sum += i;
            return sum;
        } else {
            int mid = (start + end) / 2;
            ForkJoinExample leftTask = new ForkJoinExample(start, mid);
            ForkJoinExample rightTask = new ForkJoinExample(mid + 1, end);
            leftTask.fork();
            rightTask.fork();
```

```
            return leftTask.join() + rightTask.join();
        }
    }

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        ForkJoinExample task = new ForkJoinExample(1, 100);
        int result = pool.invoke(task);
        System.out.println("Sum: " + result);
    }
}
```

5. BlockingDeque

BlockingDeque는 `java.util.concurrent`의 인터페이스로, **양방향 큐**에서 스레드가 데이터 추가와 제거 작업을 동기화하며 블로킹할 수 있는 큐입니다. `putFirst()`와 `putLast()`, `takeFirst()`와 `takeLast()` 메서드를 통해 큐의 양쪽에서 작업을 수행할 수 있습니다. 생산자-소비자 패턴에서 주로 사용됩니다.

사용 예시

```
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class BlockingDequeExample {
    private BlockingDeque<String> deque = new LinkedBlockingDeque<>();

    public void produce(String item) throws InterruptedException {
        deque.putLast(item);  // 큐의 끝에 아이템 추가
        System.out.println("Produced: " + item);
    }

    public void consume() throws InterruptedException {
        String item = deque.takeFirst();  // 큐의 앞에서 아이템 소비
        System.out.println("Consumed: " + item);
    }
}
```

6. java.util.concurrent 패키지

`java.util.concurrent` 패키지는 **멀티스레딩**과 **병렬 처리를 위한 다양한 도구와 클래스**를 제공합니다. 이 패키지는 자주 사용되는 유틸리티를 제공하며, 다음과 같은 주요 클래스를 포함합니다:

- **Executors**: 다양한 스레드 풀을 생성하는 팩토리 메서드 제공 (`newFixedThreadPool()`, `newCachedThreadPool()` 등).
- **CountDownLatch**: 한 스레드가 다른 스레드의 작업이 완료될 때까지 기다리도록 설정하는 데 사용.
- **CyclicBarrier**: 여러 스레드가 특정 지점에서 만나도록 동기화하는 도구.
- **Semaphore**: 리소스 접근을 제한하여 동시에 실행할 수 있는 스레드 수를 제한.
- **ConcurrentHashMap**: 동기화된 해시 맵으로, 여러 스레드가 동시에 데이터를 안전하게 읽고 쓸 수 있음.

예시 - CountDownLatch

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(3);

        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
                try {
                    System.out.println("Thread " +
Thread.currentThread().getName() + " is working");
                    Thread.sleep(1000);
                    latch.countDown();
                    System.out.println("Thread " +
Thread.currentThread().getName() + " finished work");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        latch.await();  // 모든 작업이 끝날 때까지 대기
        System.out.println("All threads finished");
    }
}
```

Thread의 동작과정

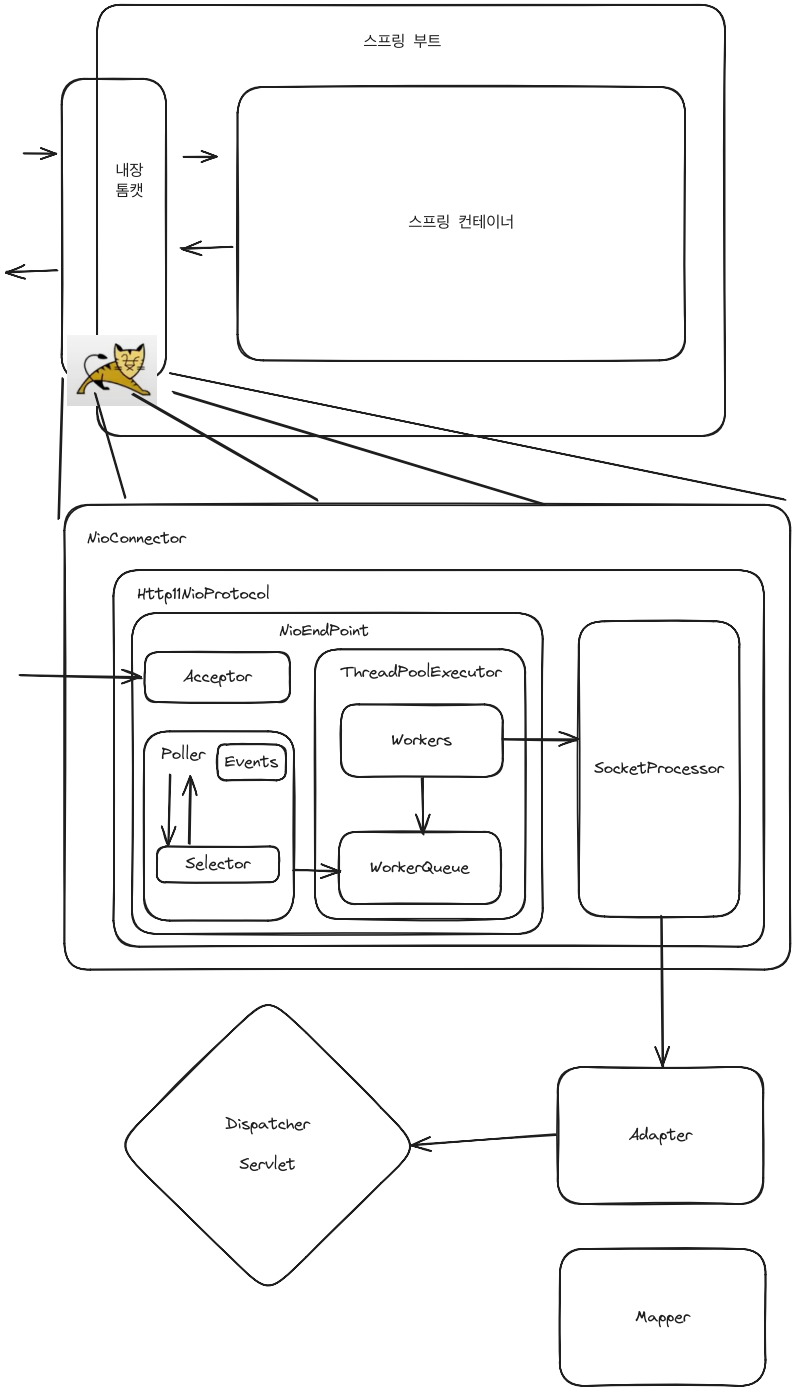
참고 : https://www.youtube.com/watch?v=prnilLbdOYA&t=12s

Tomcat과 Spring의 스레드 흐름

1. 클라이언트 요청 수신:

- 클라이언트가 HTTP 요청을 Tomcat 서버로 전송합니다. Tomcat은 서블릿 컨테이너로서 이 요청들을 수신하고 처리합니다.

2. **Tomcat 스레드 풀 관리**:
   - Tomcat에는 **HTTP NIO 커넥터**로 불리는 스레드 풀이 있어서 들어오는 요청을 관리합니다. 요청이 오면 이 스레드 풀에서 하나의 스레드를 할당하여 요청을 처리합니다. 만약 모든 스레드가 사용 중이라면 요청은 대기열에 들어가서 스레드가 비워질 때까지 기다립니다.
3. **서블릿 및 Spring 컨트롤러 실행**:
   - Tomcat의 스레드는 요청을 **서블릿**으로 전달하고, 서블릿은 요청을 처리한 후 Spring 프레임워크의 디스패처 서블릿으로 전달합니다.
   - Spring의 디스패처 서블릿은 같은 Tomcat 스레드를 사용하여 적절한 **컨트롤러** 메서드를 호출하고 비즈니스 로직을 처리하게 합니다.
4. **비즈니스 로직 처리 및 비동기 처리 옵션**:
   - 컨트롤러에서 동기식 처리가 이루어질 때는 요청 스레드가 해당 로직을 끝까지 처리합니다.
   - 비동기 처리가 필요할 때는 Spring의 **@Async** 어노테이션, **CompletableFuture** 또는 사용자 지정 **ExecutorService**를 통해 별도의 스레드를 사용합니다. 이렇게 하면 원래 요청 스레드는 다른 작업을 처리할 수 있어 효율성이 향상됩니다.
5. **예외 처리**:
   - 처리 중 예외가 발생하면 Spring은 **@ControllerAdvice**나 **ExceptionHandler** 메서드를 통해 예외를 관리하고, 표준화된 방식으로 클라이언트에게 응답을 보냅니다.
6. **응답 반환**:
   - 비즈니스 로직 처리가 완료되면 Spring은 같은 Tomcat 스레드를 사용하여 응답을 디스패처 서블릿을 통해 Tomcat 서버로 반환합니다.
   - 이후 Tomcat은 설정에 따라 연결을 닫거나, HTTP keep-alive 설정에 따라 연결을 유지합니다.
7. **클라이언트 응답 수신**:
   - 마지막으로 클라이언트는 처리된 응답을 수신하면서 요청-응답 라이프사이클이 완료됩니다.


스프링 부트
내장
톰캣
스프링 컨테이너
NioConnector
Http11NioProtocol
NioEndPoint
Acceptor
ThreadPoolExecutor
Poller
Events
Workers
SocketProcessor
Selector
WorkerQueue
Dispatcher
Servlet
Adapter
Mapper