동시성 모델 Part1

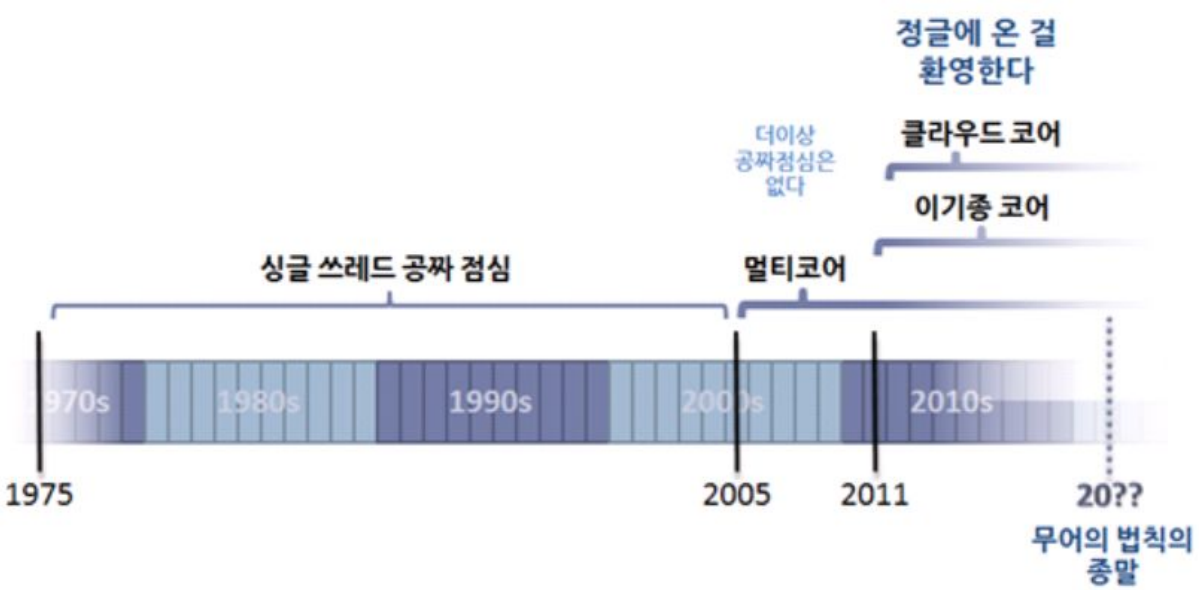
백찬규

Index

1. 동시성
2. 스레드와 잠금장치
3. 개발 환경 돌아보기

1. 동시성

The Free Lunch is Over

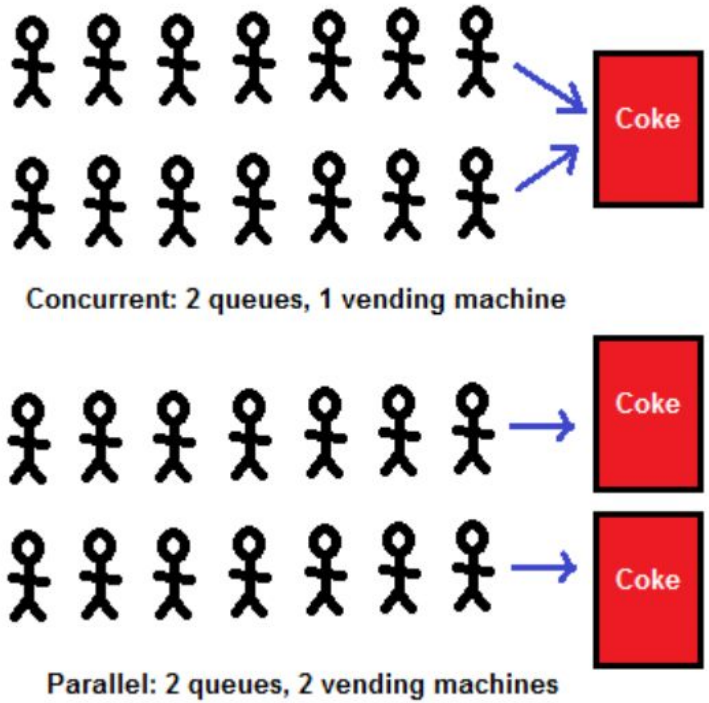


- “free lunch”: 싱글 스레드 애플리케이션이 CPU 클럭 스피드 증가에 따라 수행 성능이 좋아지는 것

- 멀티코어 기반으로 하드웨어 환경이 변함 > 멀티코어를 활용하는 소프트웨어 개발이 중요해짐

- 순차적 프로그래밍을 넘어서자!

Concurrency vs Parallelism



- 동시성(병행성): 동시에 실행되는 것처럼 보인다
  - 논리적
  - 병렬 착각
  - 문제의 속성(한꺼번에 여러 일을 다룸)
  - ex) 싱글코어 멀티 스레드

- 병렬성: 실제로 동시에 실행이 된다.
  - 물리적
  - 해법의 속성(한꺼번에 여러 일을 처리)
  - ex) 멀티코어 멀티 스레드

동시성 프로그래밍

문제의 속성이자 문제 그 자체인 동시성을 잘 다루는 프로그래밍

1. 동시적인 세계 > 동시적 소프트웨어
2. 분산된(되어야하는) 세계 > 분산 소프트웨어
3. 예측 불가한(충돌, 장애) 세계 > 탄력적 소프트웨어
4. 복잡한 세계 > 단순한 소프트웨어

2. 스레드와 잠금장치

잠금장치를 이용한 멀티 스레딩

장점

- 직관적인 개념
- 다들 한번쯤 배웠음
- 폭넓은 적용범위

단점

- 제대로 사용하기 어려움
- 분산 메모리 아키텍처 미지원

멀티스레딩이 어려운 이유 - 1) 경쟁 조건

```
public class Counting {
  public static void main(String[] args) throws InterruptedException {
    class Counter {
      private int count = 0;
      public void increment() { ++count; }
      public int getCount() { return count; }
    }
    final Counter counter = new Counter();
    class CountingThread extends Thread {
      public void run() {
        for(int x = 0; x < 10000; ++x)
          counter.increment();
      }
    }

    CountingThread t1 = new CountingThread();
    CountingThread t2 = new CountingThread();
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println(counter.getCount());
  }
}
```

- 출력시 20000 아닌 매번 다른 값
- 스레드 상호배제 안되어 메서드 increment 수행 간 경쟁 조건 발생
- count++ 의 수행 순서가 꼬임
  - 읽기
  - 수정하기
  - 쓰기

멀티스레딩이 어려운 이유 - 1) 경쟁 조건

```
public class Counting {

  public static void main(String[] args) throws InterruptedException {

    class Counter {
      private int count = 0;
      public synchronized void increment() { ++count; }
      public int getCount() { return count; }
    }
    final Counter counter = new Counter();

    class CountingThread extends Thread {
      public void run() {
        for(int x = 0; x < 10000; ++x)
          counter.increment();
      }
    }

    CountingThread t1 = new CountingThread();
    CountingThread t2 = new CountingThread();

    t1.start(); t2.start();
    t1.join(); t2.join();

    System.out.println(counter.getCount());
  }
}
```

> increment 메서드에 잠금장치를 이용

- 스레드가 increment 호출시 Counter 객체의 락 요구
- count++ 수행 간 상호배제 보장
- 리턴시 락 해제

멀티스레딩이 어려운 이유 - 2) 메모리 가시성 보장x

```
public class Counting {

  public static void main(String[] args) throws InterruptedException {

    class Counter {
      private int count = 0;
      public synchronized void increment() { ++count; }
      public synchronized int getCount() { return count; }
    }
    final Counter counter = new Counter();

    class CountingThread extends Thread {
      public void run() {
        for(int x = 0; x < 10000; ++x)
          counter.increment();
      }
    }

    CountingThread t1 = new CountingThread();
    CountingThread t2 = new CountingThread();

    t1.start(); t2.start();
    t1.join(); t2.join();

    System.out.println(counter.getCount());
  }
}
```

- 특수한 경우 getCount 가 오래된 메모리 값을 가져오는 문제 발생
- 쓰는 스레드와 읽는 스레드 모두 동기화 되어야함

> getCount 메서드에도 잠금장치를 이용

멀티스레딩이 어려운 이유 - 3) 데드락

```
class Philosopher extends Thread {
  private Chopstick left, right;
  private Random random;
  private int thinkCount;

  public Philosopher(Chopstick left, Chopstick right) {
    this.left = left; this.right = right;
    random = new Random();
  }

  public void run() {
    try {
      while(true) {
        ++thinkCount;
        if (thinkCount % 10 == 0)
          System.out.println("Philosopher " + this + " has thought " + thinkCount + " times");
        Thread.sleep(random.nextInt(1000));     // Think for a while
        synchronized(left) {                    // Grab left chopstick
          synchronized(right) {                 // Grab right chopstick
            Thread.sleep(random.nextInt(1000)); // Eat for a while
          }
        }
      }
    } catch(InterruptedException e) {}
  }
}
```

- 둘 이상의 잠금장치를 가지려는 스레드 간 데드락의 위험성 발생

> 잠금장치 요청시 공통의 순서를 따르게 하는 방식으로 해결

3. 개발 환경 돌아보기

3. 개발 환경 돌아보기

```
def create_like(self, user):
    self.likes.create(user=user)
    News.objects.filter(id=self.id).update(like_count=F('like_count') + 1)
```

gimo-song 10 days ago

1. self를 쓰면 filter를 사용하지 않을텐데 쿼리랑 별개의 문제 아닌가요?
2. 해당 함수에서 어떤 race condition이 발생할 수 있는지 궁금합니다.

F객체 왜쓰냐의 맥락에서...

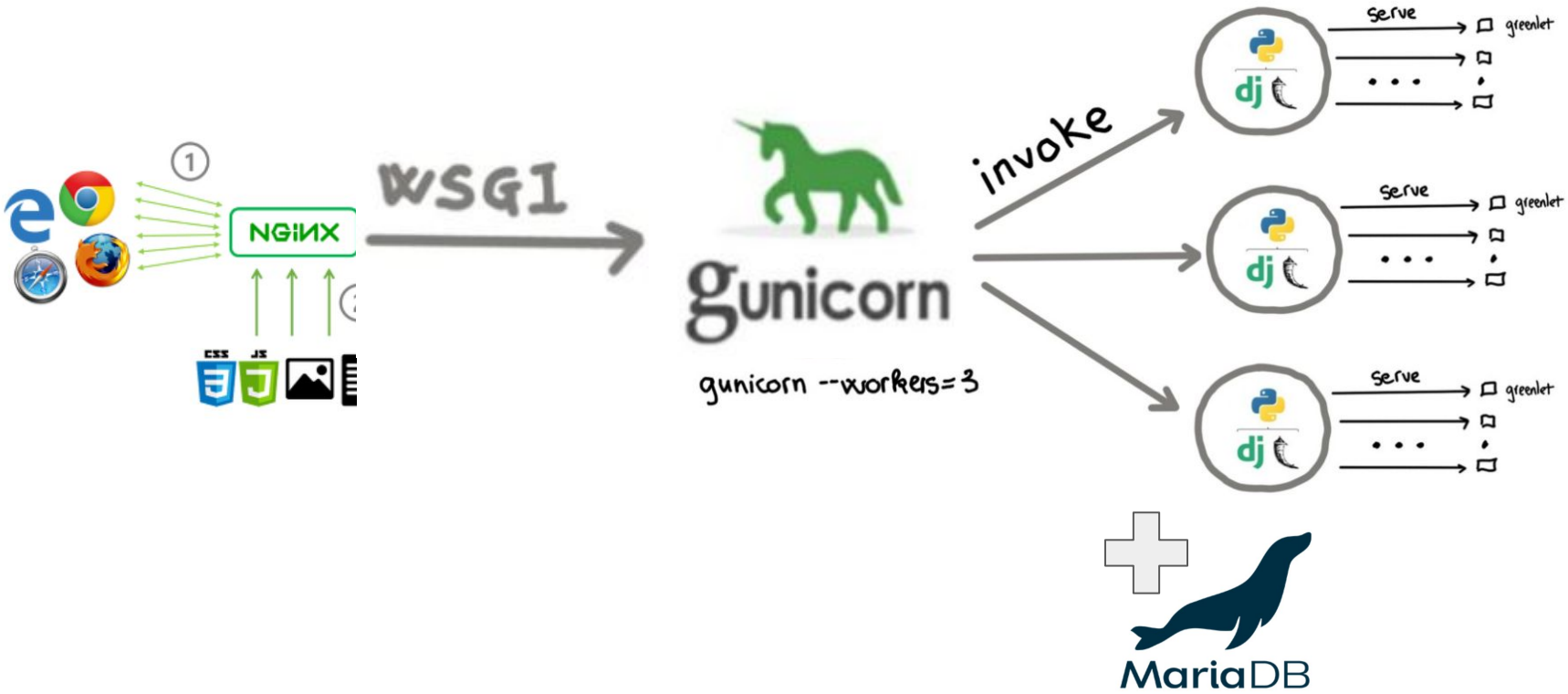
race condition?

프로세스? 스레드? 커넥션?

GIL?

db transaction?

3. 개발 환경 돌아보기

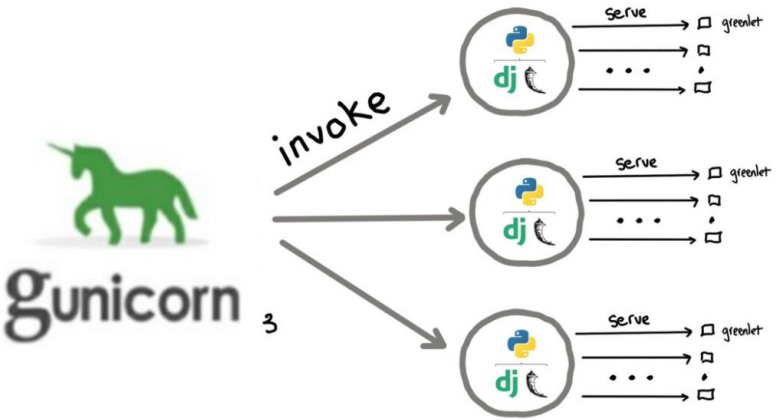

1
NGINX
WSGI
gunicorn
gunicorn --workers=3
invoke
serve
greenlet
dj
MariaDB

3. 개발 환경 돌아보기 - gunicorn

3가지 측면의 동시성 지원

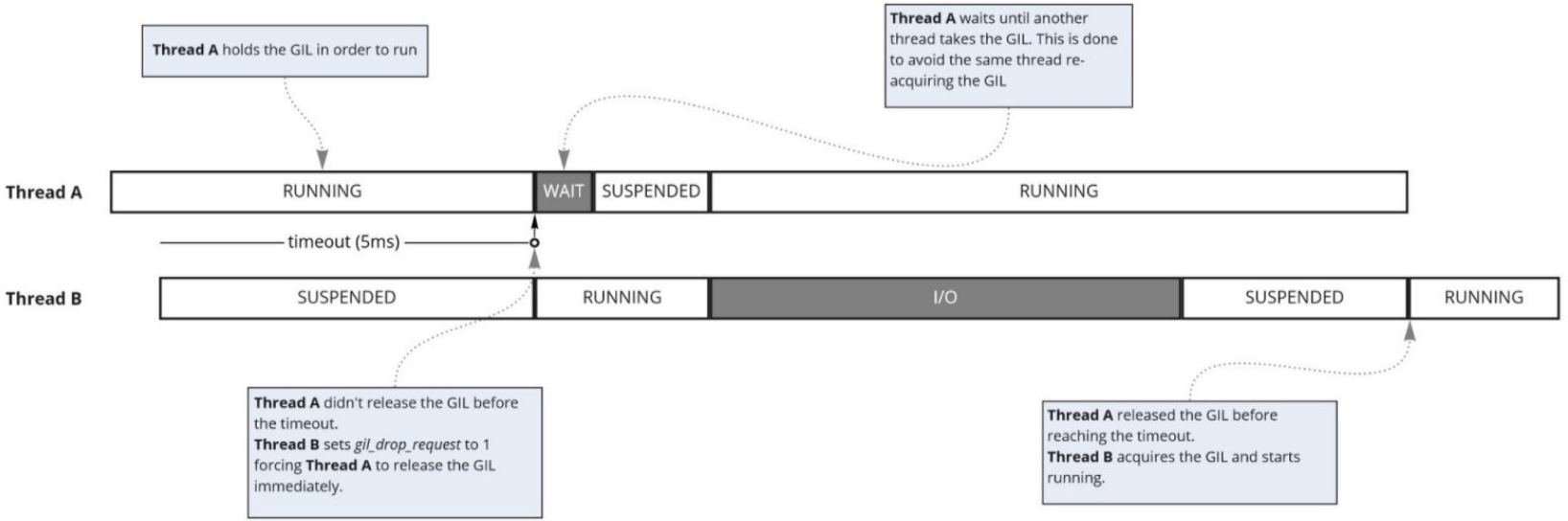
1. workers(UNIX processes)
   a. worker_class = ‘sync’(디폴트)
2. threads
   a. 워커 하나 당 여러 스레드
   b. worker_class = ‘gthread’
3. pseudo-threads
   a. 워커 하나 당 더 가벼운 여러 스레드
   b. worker_class = ‘gevent’

3. 개발 환경 돌아보기 - gunicorn



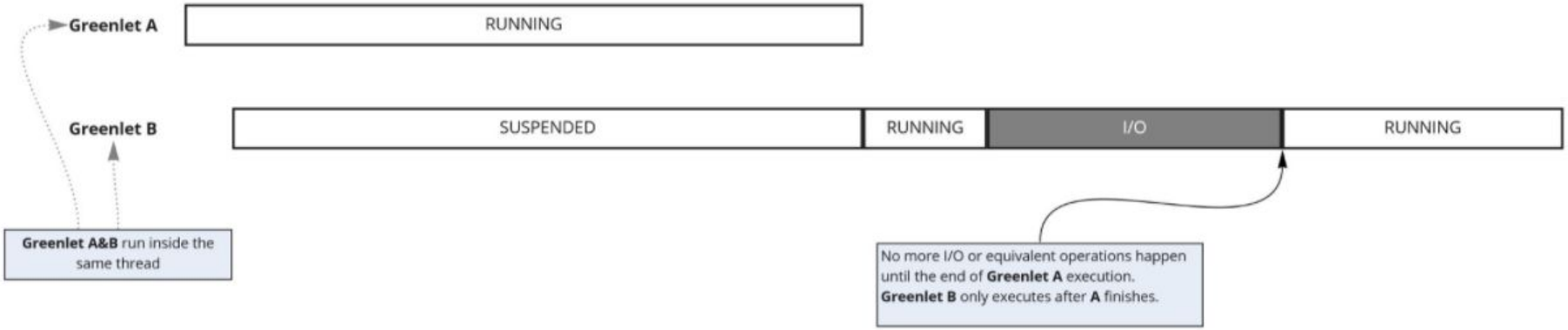
- worker_class = ‘gevent’
- workers = cpu_count() * 2 + 1
- worker_connection = 1000
- ***max concurrent request = workers * worker_connection***

3. 개발 환경 돌아보기 - GIL



- cpython에서 Reference Counting이 Atomic하게 동작하도록 보호
- GIL 가진 스레드만 동작하도록 보장
- race condition이 없음을 보장하지 않음(타임아웃시 GIL을 릴리즈함)

3. 개발 환경 돌아보기 - GIL X gevent



- pseudo-thread는 동일한 스레드 내부에서 동작
  - GIL에 영향 받지 않음
- cooperative > avoid interruptions and thread-switching
- I/O opertaion 발생 않는 이상 같은 워커 내에서 순차적 수행이 보장됨

3. 개발 환경 돌아보기 - db

- ATOMIC_REQUESTS = True
  - view 함수 호출 전 트랜잭션 시작
  - 리턴시 트랜잭션 커밋, 예외 발생시 롤백
- InnoDB > REPEATABLE READ

```python
def create_like(self, user):
    self.likes.create(user=user)
    self.like_count += 1
    self.save(update_fields='like_count')
    # News.objects.filter(id=self.id).update(like_count=F('like_count') + 1)
```