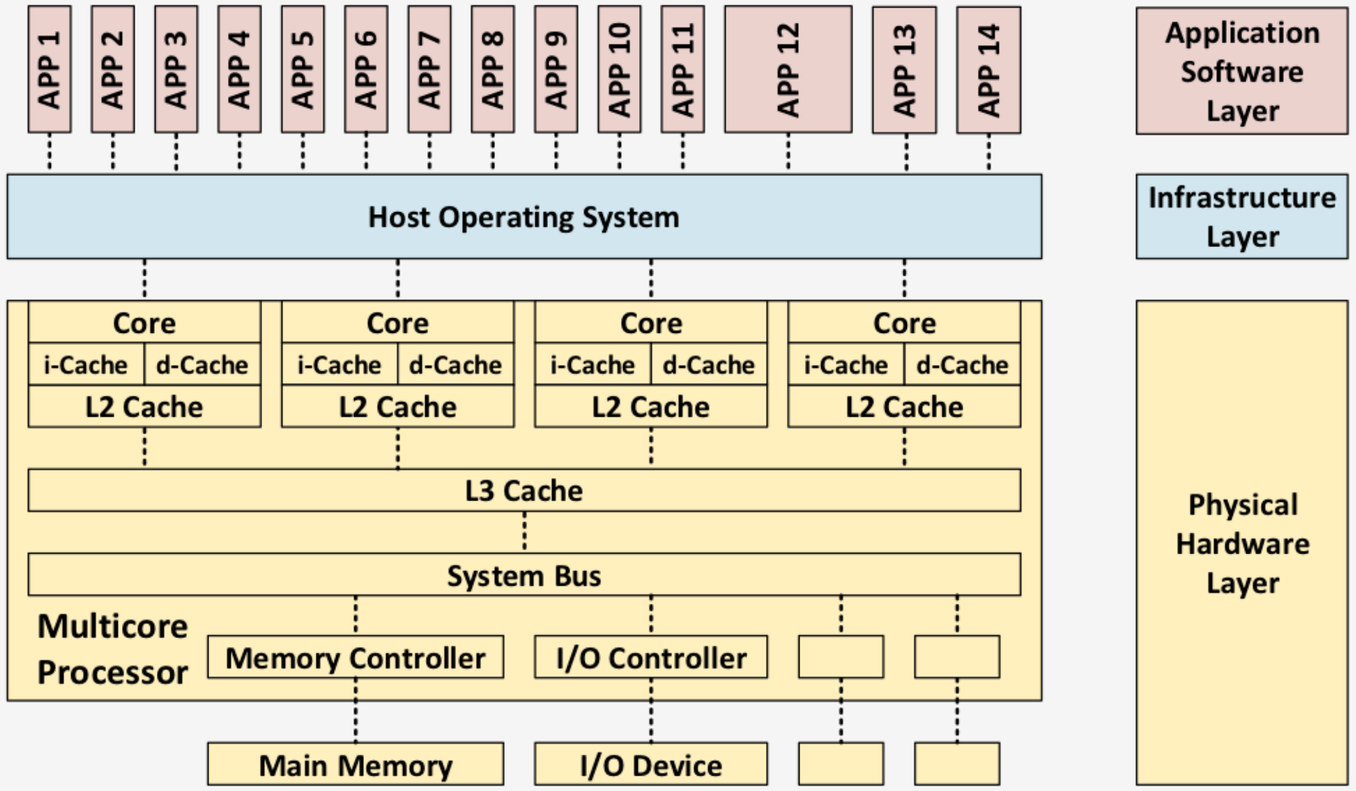
멀티스레드와 캐시 최적화

허진혁

Contents

01 문제 정의



멀티 코어 환경

여러 스레드가 동시에 실행되며 캐시와 메모리 공유

발생할 수 있는 문제점

1) Race Condition 발생

2) 캐시 미스

02 Thread Safe란?

여러 스레드가 동시에 하나의 접근하더라도 프로그램의 실행 결과가 항상 올바르게 유지되는 상태

원자성

어떤 연산이 쪼개지지 않고 딱 한번에
일어난 것처럼 보장되는 것

가시성

한 스레드가 바꾼 값이
다른 스레드에게 즉시 보이는 것

순서성

코드가 작성된 순서처럼
보이는 성질

동시 실행 환경에서 데이터 무결성과 일관성 보장

03 원자성

```
class Counter {
    int value = 0;
    void inc() { value++; }
}

public class Demo {
    public static void main(String[] args) throws Exception {
        Counter c = new Counter();
        int threads = 8, loops = 100_000;
        Thread[] ts = new Thread[threads];
        for (int i=0;i<threads;i++) {
            ts[i] = new Thread(() -> {
                for (int k=0;k<loops;k++) c.inc();
            });
            ts[i].start();
        }
        for (Thread t: ts) t.join();
        System.out.println("expected=" + (threads*loops) + ", actual=" + c.value);
    }
}
```

value++

1) load : 메모리에서 value 값 레지스터로 읽어옴

2) add : 1더하고

3) store : 결과를 메모리에 씀

여러 스레드가 동시에 작업 수행하면
인터리빙 발생 가능

```
expected=800000, actual=581196
```

04 가시성

```
lass Demo2 {
   private static boolean running = true;

   public static void main(String[] args) throws InterruptedException {
       Thread worker = new Thread(() -> {
           while (running) { }
           System.out.println("Worker stopped");
       });

       worker.start();
       Thread.sleep(1000);

       running = false;
       System.out.println("Main set running=false");
   }
```

```
Main set running=false
```

```
class Demo2 {
    private static volatile boolean running = true;

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            while (running) { }
            System.out.println("Worker stopped");
        });

        worker.start();
        Thread.sleep(1000);

        running = false;
        System.out.println("Main set running=false");
    }
}
```

```
Worker stopped
Main set running=false
```

volatile : 한 스레드가 바꾼 값이 다른 스레드에게 즉시 보임

05 순서성

```
class Singleton {
    private static Singleton instance;

    private int[] data;

    private Singleton() {
        this.data = new int[1000];
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

instance = new Singleton();

1) 메모리 공간 확보 (allocate)

2) 생성자 호출해서 필드 초기화

3) 변수 instance에 참조 대입

동기화가 없다면 CPU가 최적화하면서 1-> 3-> 2로 실행 가능

스레드 A : instance에 초기화 안 된 객체 참조 저장

스레드 B : data 배열이 초기화되지 않은 부분 초기화 상태

06 Race Condition

여러 스레드가 동시에 공유 자원에 접근할 때, 실행 순서에 따라 결과가 달라지는 상황

1) 결과의 비결정성

코드가 실행될 때 마다 결과가 달라짐

2) 데이터 무결성 훼손

공유 자원의 값이 꼬이거나, 불변식 깨짐

3)간헐적 발생

운영 환경(코어 수, 스케줄링 등)에 따라 가끔 발생

06 Race Condition

Peterson 알고리즘

두 스레드가 동시에 임계 구역에 진입하지 않도록 보장하는 알고리즘

Peterson's solution

```
                        turn = 0;
                        flag[0] = false;
                        flag[1] = false;
P0                                          P1
while(true){                                while(true){
  flag[0] = true;                             flag[1] = true;
  turn =1;                                    turn = 0;
  while (flag[1] and turn ==1) do no-op;        while (flag[0] and turn ==0) do no-op;
    CS                                          CS
  flag[0] = false;                            flag[1] = false;
  remainder section;                          remainder section;
}                                           }
```

1) 두 스레드에서만 동작 → 다중 스레드 확장 불가

2) 현대 CPU 환경 부적합

3) 성능 한계

06 Race Condition

1) Lock 기반 제어

공유 자원에 접근하는 코드를 임계 구역으로 묶어,
동시에 1개 스레드만 접근

A) synchronized 키워드 사용

```
class Demo {
    private int value = 0;
    public synchronized void inc() { value++; }
    public synchronized int get() { return value; }
}
```

자바 키워드로 블록/메서드에 락을 걸어 간단하고 안전하게 동기화
but 세밀한 제어는 어렵다.

06 Race Condition

B) ReentrantLock

```java
import java.util.concurrent.locks.ReentrantLock;

class Demo {
    private int value = 0;
    private final ReentrantLock lock = new ReentrantLock();

    public void inc() {
        lock.lock();
        try {
            value++;
        } finally {
            lock.unlock();
        }
    }
}
```

장점 : 원자성, 가시성, 순서성 확실히 보장

단점 : 락 경합 → 성능 저하, deadlock 위험

ex) 은행 계좌, 결제 시스템 등

클래스 기반 락으로 타임아웃, 인터럽트 대응, 조건 변수 등 고급 기능을 제공

but 직접 unlock을 관리해야 한다.

06 Race Condition

2) Lock - free 전략

스레드가 락을 기다리며 막히지 않고(non-blocking)

충돌 시 재시도나 다른 설계로 해결하는 전략

A) Atomic 변수

하드웨어 CAS 기반 연산 활용 원자성 보장

ex) AtomicInteger, LongAdder 등

```
private AtomicInteger count = new AtomicInteger(0);

public void increment() {
    count.incrementAndGet();
}
```

락보다 빠르고 데드락 없음

but 경합 심하면 CAS 재시도 비용 증가 → 성능 저하

ex) 로그 집계 등

06 Race Condition

B) Immutable 객체

객체를 한 번 만든 뒤 절대 수정하기 않음

바뀌면 새 객체로 교체

ex) String, Java record

```
public record User(String id, String name) {}
```

여러 스레드가 동시에 읽어도 안전, 동기화 불필요

but 수정할 때마다 새 객체 생성

06 Race Condition

C) Thread - local (스레드 한정)

공유하지 않고 스레드마다 독립 데이터 유지

→ 동시성 문제 해결

ex) ThreadLocal<T>

락 필요 없고 각 스레드 전용 데이터 가질 수 있음

but 스레드 풀 환경에서 반드시 remove()로 정리 (메모리 누수 방지)

```
private static final ThreadLocal<Integer> local = ThreadLocal.withInitial(() -> 0);

local.set(local.get() + 1);
```

06 Race Condition

D) Concurrent Collections

내부적으로 락 분할이나 CAS 사용해 병렬 접근 최적화

ex) ConcurrentHashMap 등

장점 : 대기와 데드락이 없어 성능과 확장성 뛰어남

단점 : 경합이 심한 상황에서 CAS 재시도로 성능 저하

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.putIfAbsent("a", 1);
map.compute("a", (k, v) -> v+1);
```

다수 스레드에서 접근해도 안전 , API가 원자적 연산 제공

but 내부적으로 부분적 락이나 CAS 사용

06 Race Condition

컬렉션	구현 방식	특징
ConcurrentHashMap	주로 CAS + 일부분 lock	대부분 연산 CAS 기반 lock-free 구조 변환 시 락 사용
ConcurrentLinkedQueue / Deque	CAS 기반	완전 lock-free head/tail 갱신 CAS로 처리
ConcurrentSkipListMap / Set	주로 CAS + 일부분 lock	정렬 맵, 대부분 CAS로 동작 구조 변환 시 락 사용
CopyonWriteArrayList / Set	lock	쓰기 전 전체 복사 ReentranLock 사용(읽기 위주)
BlockingQueue 계열	lock	내부에서 ReentrantLock put/take 제어

07 캐시 미스

CPU가 필요한 데이터를 캐시에서 찾지 못해 더 느린 메모리에서 가져오는 상황

1) Cold miss

처음 접근하는 데이터라 캐시 존재 x

2) Capacity miss

작업 집합이 캐시 용량 초과 → 오래된 데이터가 사라짐

3) Conflict miss

캐시가 특정 방식으로만 저장 (Direct-mapped 등)

서로 다른 데이터가 같은 캐시 라인에 매핑 → 충돌

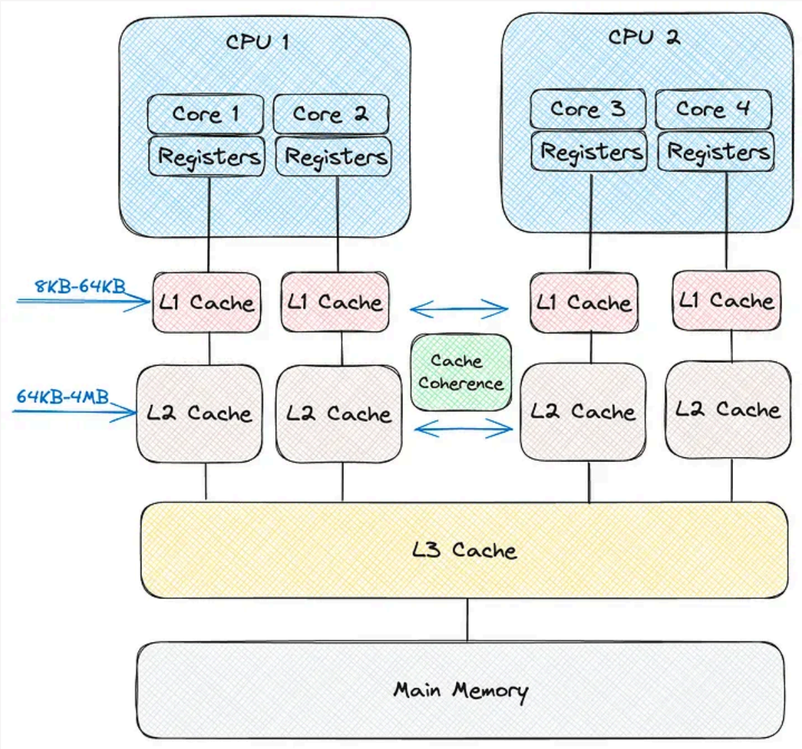
4) Coherence miss (멀티 코어)

다른 코어가 같은 캐시 라인 데이터 수정해서 내 캐시 무효화

07 캐시 미스

캐시 메모리

CPU와 메인 메모리 사이에 위치한 작은 고속 메모리



캐시 라인 단위로 데이터 관리

CPU가 메모리 읽으면, 해당 주소가 포함된 캐시 라인을 캐시에 적재

다음 접근이 같은 캐시 라인에 있으면 캐시 히트

없으면 캐시 미스

07 캐시 미스

A) 지역성 활용

1) 시간 지역성

최근에 접근한 데이터는 다시 접근할 가능성 높다

```
int sum = 0;
for (int i = 0; i < 1000; i++) {
    sum += arr[0];
}
```

2) 공간 지역성

연속된 메모리 영역은 같이 접근될 가능성 높다

```
for (int i = 0; i < N; i++) {
    sum += arr[i];
}
```

07 캐시 미스

C/Java 배열은 row-major order: 한 행의 원소들이 메모리에 연속 저장

행 우선 탐색

```
for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
        sum = a[i][j];
    }
}
```

열 우선 탐색

```
for (int j=0; j<N; j++) {
    for (int i=0; i<N; i++) {
        sum += a[i][j];
    }
}
```

```
N=8192 (67108864 elements)

Run 1: row-first=  34 ms, col-first=1239 ms  (ratio=36.44x)
Run 2: row-first=  35 ms, col-first=1211 ms  (ratio=34.60x)
Run 3: row-first=  29 ms, col-first=1223 ms  (ratio=42.17x)
Run 4: row-first=  34 ms, col-first=1223 ms  (ratio=35.97x)
Run 5: row-first=  31 ms, col-first=1233 ms  (ratio=39.77x)
```

그렇다면 파이썬은?

07 캐시 미스

B) 데이터 구조 / 배치 최적화

1) 구조체 필드 재배치

Hot : 자주 쓰는 필드

Cold : 잘 안쓰는 필드

```
class User {
    int id;          // 자주 쓰임
    int age;         // 자주 쓰임
    String address; // 잘 안 쓰임
    String bio;      // 잘 안 쓰임
```

2) AoS → SoA

AoS : 구조체 배열, 각 요소가 구조체

```
class Point { float x, y, z; }
Point[] arr = new Point[N];
```

SoA : 필드별로 배열 따로 둠

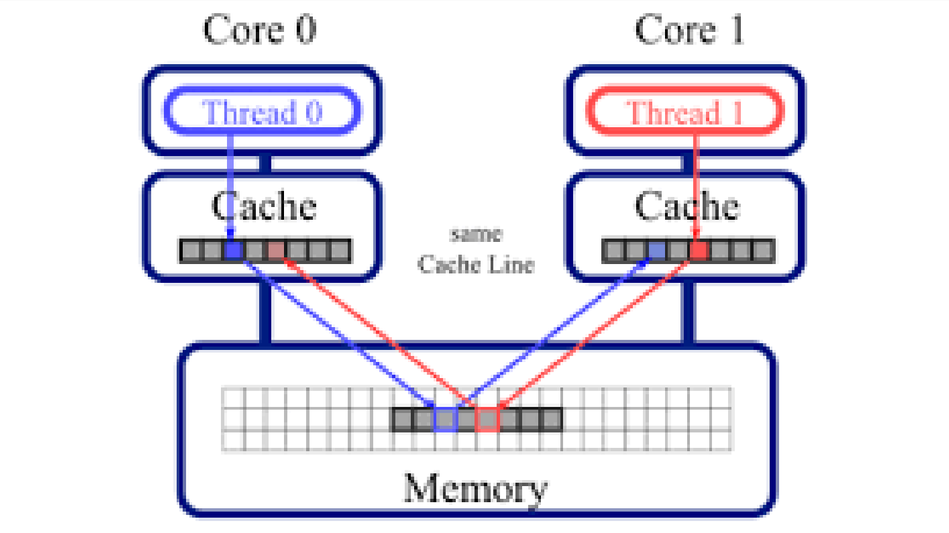
```
float[] xs = new float[N];
float[] ys = new float[N];
float[] zs = new float[N];
```

07 캐시 미스

C) False Sharing 방지

False Sharing

여러 스레드가 서로 다른 변수 수정 (같은 캐시 라인)



line bouncing 발생

1) 변수 사이 패딩

```
class PaddedCounter {
    volatile long a = 0L;
    long p1,p2,p3,p4,p5,p6,p7;
    volatile long b = 0L;
}
```

-> 메모리 낭비

2) @Contended

```
class Counter {
    @Contended
    public volatile long value = 0;
}
```

3) Long Adder / Striped Counter

```
LongAdder adder = new LongAdder();
adder.increment();
```

07 캐시 미스

D) 캐시 친화적 알고리즘

1) 블로킹

큰 데이터를 작은 블록으로 나눠 작업

→ 각 블록이 캐시에 fit

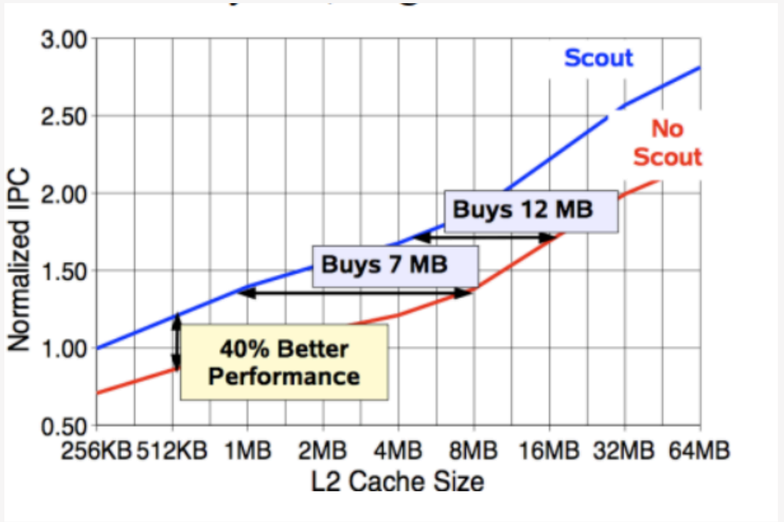
```
for (int ii = 0; ii < N; ii += B) {
    for (int jj = 0; jj < N; jj += B) {
        for (int kk = 0; kk < N; kk += B) {
            for (int i = ii; i < ii+B; i++)
                for (int j = jj; j < jj+B; j++)
                    for (int k = kk; k < kk+B; k++)
                        C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

멀티코어 활용 좋음, 캐시 히트율 증가

but 블록 크기 튜닝 필요

2) 프리페칭

CPU가 실제 접근하기 전 미리 데이터를 캐시에 불러오는 작업



메모리 대기시간 줆어듦 (성능 향상)

but 캐시 오염 가능성, 낭비 발생 가능

감사합니다