Lock, Synchronization Orientation

sanghyuck.na@lge.com

1

Race condition

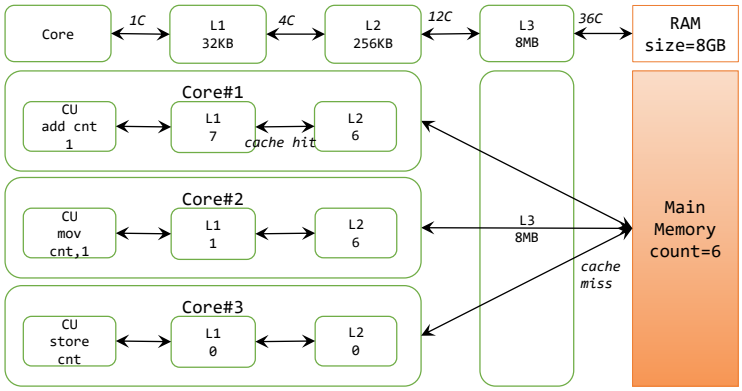
- 두 개 이상 Thread가 한 공유자원을 동시에 접근하고 수정할 있는 상태
  - Thread scheduling algorithm과 그 전략에 따라 미정의순서로 실행될 수 있음
  - 그 최종 결과는 예측될 수 없고, 데이터 무결성도 지켜지지 않음

```
Integer counter = Integer.valueOf(0);

public void run() {
    synchronized(counter) {
        ++counter;
    }

    System.out.println( "p=" +  counter    );
…
p=9552
a=10000
```

2

Modern CPU Model

- "check-and-act"부터 출발
  - 기억장치 간 속도차로 인한 성능지연을 해결하기 위해 CPU와 Storage 사이에 Memory와 Cache L1, L2가 있습니다
  - 현 시점에 그 값이 예상된 값인지 '확인'작업과 그로 인해 발생하는 '동작' 은 Multithread환경에서는 필수적입니다



4

JAVA Assembler

- istore_과 iload
  - JAVA Bytecode수준의 Store/Load이지만 JVM실행 시 실제 Register로 바꾸고 기계어 수준으로 Shared Resource의 Consistency와 Integrity를 고려해야 합니다.
- 최적화 배치결과는 실제 코드와 다를수 있습니다
  - Control unit 사용율을 향상시키기 위한 Pipelining 을 위함
  - Fetch > Decode > Execute > write-back

```
main:
.LFB0:
        .cfi_startproc
        movl    b(%rip), %eax
        movl    $2, b(%rip)
        addl    $1, %eax
        movl    %eax, a(%rip)
        ret
        .cfi_endproc
b:
        .zero   4
a:
        .zero   4
```

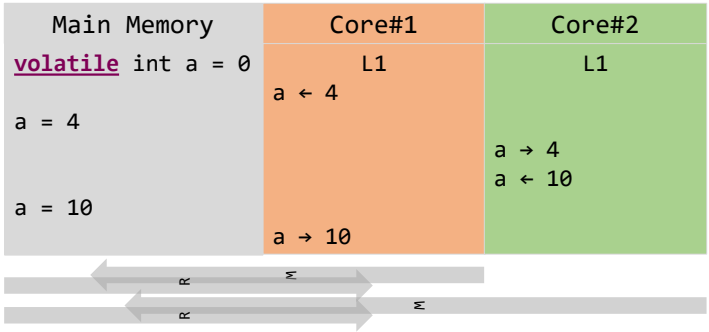
```
Main {
  int a;
  int b;
  void testMain();
    Code:
       0: aload_0
       1: aload_0
       2: getfield      #15  // Field b:I
       5: iconst_1
       6: iadd
       7: putfield      #13  // Field a:I
      10: aload_0
      11: iconst_1
      12: putfield      #15  // Field b:I
      15: return
}
```

```
int a = 0;
int b = 0;
int main(){
    a = b + 1;
    b = 2;
}
```

6

Volatile – Write through

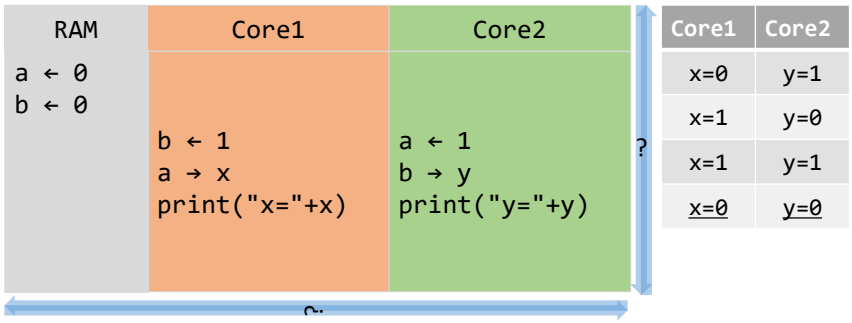
- 한 변수의 데이터를 모든 스레드가 일관되게 바라볼 수 있음
  - Store는 항상 메모리로 바로 작성된다
  - Load는 항상 메모리를 참조한다



7

Opaque(or Releax)

- 가장 느슨한 자유로운 모드 memory_order_relaxed[c++14]
  - 빠른 연산속도를 위한 인스트럭션재배치 모드
  - Store, Load의 명령어 재정렬제약이나 Multithread간 동기화가 없는 자유로운 모드
  - 연산자(S/L)의 원자성만 보장합니다. 아래 왼쪽 코드는 오른쪽결과가 모두 나타날 수 있습니다.
- Opaque mode



8

Data Consistency/Integrity 보장 기법

- Atomic
- Lock, Volatile
- VarHandle: Fine grain instruction 배치를 통한 성능향상 기법
  - Fence in Variable Handle: JAVA8 JEP 171
  - VarHandle (Memory barrier): in Variable Handle: JAVA9 JEP 193

9

Atomic type

sanghyuck.na@lge.com

10

Atomic class[5]

- 원자적인 VarHandle[9]연산들로 단일 값을 유지하는 class
- Primitive type
  - AtomicBoolean
  - AtomicInteger
  - AtomicInteger**Array**
  - AtomicInteger**FieldUpdater**: Reflection기반으로 volatile 동작
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
- Reference type
  - Atomic**MarkableReference**: Mark bit를 함께 유지
  - AtomicReference
  - AtomicReferenceArray
  - AtomicReferenceFieldUpdater
  - Atomic**StampedReference**: "stamp"를 함께 유지

11

Atomic 의미

- 원자적인 Store, Load
- Race condition에서 값의 일관성, 무결성을 유지
  - 데이터 정확성처리
  - 최적의 성능을 위한 내부 동기화, 명령 재정렬같은 세밀한 튜닝 메소드 제공
- Value Wrapper
  - Anonymous instance나 Lambda expression 내부에서 effective final임에도 수정가능

```
AtomicLong()
AtomicLong(long initialValue)
AtomicInteger(int initialValue)
```

```
int pCnt = 0;
AtomicInteger aCnt = new AtomicInteger(0);

IntStream.range(0, 100_000).parallel().forEach(e -> {
    pCnt += 1;
    aCnt.incrementAndGet();
});
```

12

Atomic Integer

- Plus, Minus operators
  - 더 분해 할 수 없는 Read Write으로 실행 성공 또는 실패 상태만 존재
  - 증감, 차감, Sequence number로 주로사용. 단, long 값을 대체할 목적으로 사용 안함

```
int incrementAndGet()
int addAndGet(int delta)
int getAndUpdate(IntUnaryOperator updateFunction)
```

```
AtomicInteger i = new AtomicInteger(0);

i.intValue(); i.floatValue(); i.doubleValue();

i.incrementAndGet(); i.getAndIncrement();
i.decrementAndGet(); i.getAndDecrement();

i.addAndGet(2);
i.getAndAdd(5);

i.getAndUpdate((cur) -> cur + 7);
```

13

Atomic Set

- Memory effect: volatile*
  - 원자적 Reads Writes
  - 모든 스레드는 변수의 값의 현재시점의 최신사항을 동일하게 바라봅니다
- Set호출의 Argument내부에서 표현식 및 추가메소드호출은 피하세요

thread#1	thread#2
Math.max() … l.set(20)	l.set(40)

```
void set(int newValue)
```

```
AtomicLong l = new AtomicLong();
l.set(20);
l.set(Math.max(l.get(), 30));
// undefined behavior
```

15

Atomic Update

- Memory Effects: volatile
  - Atomic read를 보장하기 위해서 항상 main memory에서 읽어와야 함.
  - CPU architecture가 Cache level이 높을 수록, Mutithread 개수가 증가할 수록 비용은 커짐

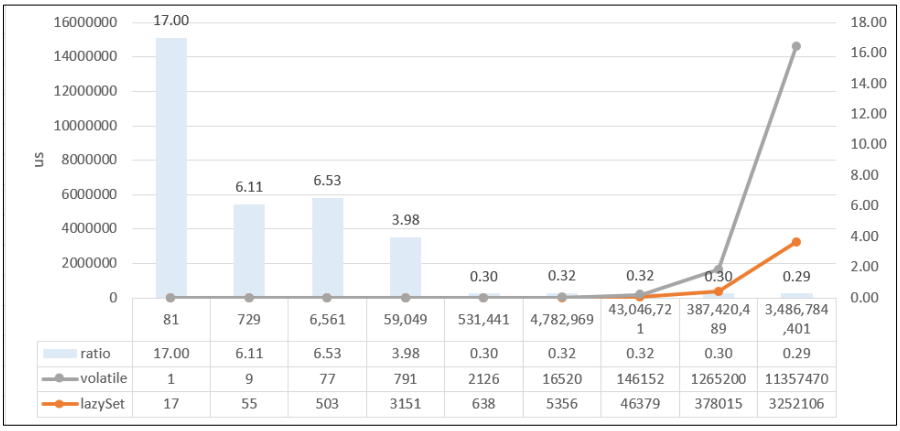
```
long getAndUpdate[8](LongUnaryOperator updateFunction)
long updateAndGet[8](LongUnaryOperator updateFunction)
long accumulateAndGet[8](long x, LongBinaryOperator accumulatorFunction)
long getAndAccumulate[8](long x, LongBinaryOperator accumulatorFunction)
```

```
AtomicLong l = new AtomicLong(50);
long now = l.updateAndGet((x) -> Math.max(x, 20));

LongBinaryOperator longb = (x, new_) -> Math.max(x, new_);
long new_ = l.accumulateAndGet(20, longb);
long old_ = l.getAndAccumulate(20, longb);
```

16

Performance volatile vs lazySet

- 초기 volatile은 lazySet보다 17배 빠른 성능을 보임[1]
- Read and writes는 증가할 수록 약 3배 더 나은성능이 나타남



	81	729	6,561	59,049	531,441	4,782,969	43,046,721	387,420,489	3,486,784,401
ratio	17.00	6.11	6.53	3.98	0.30	0.32	0.32	0.30	0.29
volatile	1	9	77	791	2126	16520	146152	1265200	11357470
lazySet	17	55	503	3151	638	5356	46379	378015	3252106

17

Atomic CAS

- Compare And Set[JSR133]
  - 현재값이 기대값이 동일한 경우만 새로운값 작성(Atomic Update)
  - 메모리경합 발생 시 경쟁하지 않고 실패하는 Weak메소드 제공

```
boolean compareAndSet5(int expect, int update)

boolean weakCompareAndSetPlain9(int e, int newValue)
boolean weakCompareAndSetVolatile9(int e, int newValue)
boolean weakCompareAndSetRelease9(int e, int newValue)
boolean weakCompareAndSetAcquire9(int e, int newValue)
```

CAS Operations	Memory Effects
compareAndSet	Volatile
weakCompareAndSetPlain	Plain
weakCompareAndSetVolatile	Volatile
weakCompareAndSetRelease|Acquire	Release-Acquire

18

Atomic CAS

```
var v = new AtomicLong(0);
IntConsumer action = () -> {
    boolean status = false;
    do {
        long now = v.get();
        long new_ = now + 1;
        status = v.compareAndSet(now, expected);
        status = v.weakCompareAndSetPlain(now, new_);
        if (!status) {
            System.out.println(
                "failed " + Thread.currentThread());
        }
    } while (!status);
};

IntStream.range(0, 100).parallel().unordered()
        .forEach(action);
```

19

Memory Effects

- Reordering
  - Thread1 [1] 과 [2]는 절차적 순서로 실행됩니다. 다만 최적화 옵션과 Architecture에 따라 코드순서는 바뀔 수 있으며 그 실행결과는 실행시점에 따라 다를 수 있습니다.
- Memory Barrier[link]
  - Shared resource인 mIsReady로 [1] 과 [3]의 코드순서는 동기화 됩니다. 하지만, Memory Barrier에 따라 동기화 시점을 기준으로 상하코드의 순서는 바뀔 수 있으며 그 실행결과는 실행시점에 따라 다를 수 있습니다.

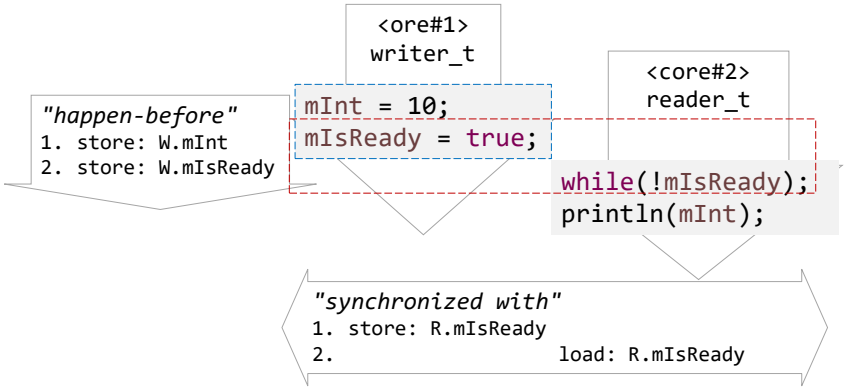
```
int mInt = 0;
boolean mIsReady = false;

// Thread1
void reader_t() {
   while(!mIsReady); [1]
    System.out.println(mInt);[2]

}
```

```
//Thread2
void writer_t() {
    mInt = 10;
    mIsReady = true; [3]
}
```

20

Memory Effects

- Reordering: Happen-before
  - 한 스레드내 Store, Load의 결정된 순서
- Memory Barrier: Synchronized-with
  - 한 스레드 내 Store, Load는 다른 스레드 Store, Load의 의존해서 순서가 결정



21

Memory Effects[kernel]

- Atomic: One cycle로 처리되는 연산으로 분할 될 수 없는 연산
- Volatile: 변수를 접근하는 순간 모든 스레드가 바라보는 값은 동일함

Atomic{E}	Atomic	Happen-Before	Synchronized-with	C++14
set(E) E get()	X			volatile
setPlain(E)[9] E getPlain()[9]	△ Reference, 32bit 이하 원시 자료형 값만	X	X	
void setOpaque(E n)[9] E getOpaque()[9]	△ Same variable	X	X	memory_order_relaxed
void setRelease(E n)[9]	O	O	△ After	memory_order_release
E getAcquire()[9]	O	O	△ Before	memory_order_acquire

22

Opaque[9]

- 비트단위로 원자적 연산
- 동일한 변수에 대한 접근은 되도록 밀착되어 연산자 정렬배치

```
var executor = Executors.newFixedThreadPool(2);
var a = new AtomicBoolean(false);
var b = new AtomicBoolean(false);
try {
    var lst = executor.<Boolean>invokeAll(List.of(() -> {
        b.setOpaque(true);
        Boolean x = a.getOpaque();
        return x;
    }, () -> {
        a.setOpaque(true);
        Boolean y = b.getOpaque();
        return y;
    }));
    executor.shutdown();
    executor.awaitTermination(5l, TimeUnit.SECONDS);
    System.out.println(lst.get(0).get()
        + ", " + lst.get(1).get());
} catch (Exception e) { e.printStackTrace();}
```

```
E getOpaque()
void setOpaque(E n)
```

Core1x	Core2y
false	true
true	false
true	true
false	false

23

Release-Acquire[9]

- setRelease을 기준으로 이전 명령들은 해당명령 이후영역으로 배치 차단
- getAcquire를 기준으로 이후 명령들은 해당명령 이전영역으로 배치 차단

```
var executor = Executors.newFixedThreadPool(2);
var a = new AtomicBoolean(false);
var b = new AtomicBoolean(false);
try {
    var lst = executor.<Boolean>invokeAll(List.of(() -> {
        b.setRelease(true);
        Boolean x = a.getAcquire();
        return x;
    }, () -> {
        a.setRelease(true);
        Boolean y = b.getAcquire();
        return y;
    }));
    executor.shutdown();
    executor.awaitTermination(5l, TimeUnit.SECONDS);
    System.out.println(lst.get(0).get()
        + ", " + lst.get(1).get());
} catch (Exception e) { e.printStackTrace();}
```

```
void setRelease(E n)
E getAcquire()
```

Core1x	Core2y
false	true
true	false
true	true

24

Atomic LazySet[6]

- Memory effects "setRelease" JDK-6275329
  - happens-before, synchronized-with, Atomicity 를 보장
- 주로 Fine tuning code로 사용
  - Non-blocking구조체에서 GC 될 수 있도록 nulling out code에서 사용해서 주요 코드실행 성능에 최대한 영향을 주지 않기 위함

```
void lazySet(E newValue)
```

```
var i = new AtomicInteger(1);
i.lazySet(50);
var normal = i.getAcquire();
```

25

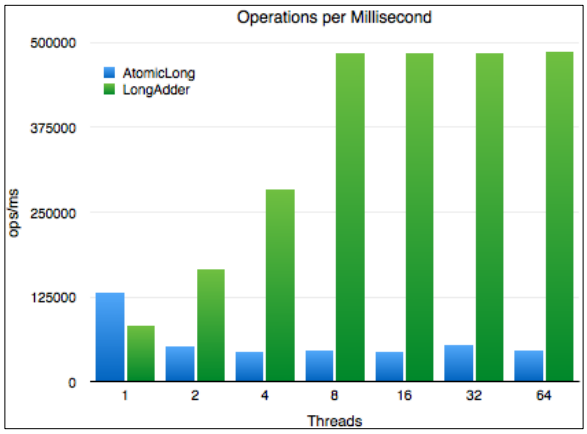
LongAdder[8]

- 덧셈전용 다중스레드 환경기반 Atomic class
  – 정밀 동기화제어는 불가하고 내부적으로 기본제어를 제공

```
var adder = new LongAdder();

adder.increment(); adder.decrement();

adder.intValue(); adder.longValue();
adder.floatValue(); adder.doubleValue();

adder.add(5);
adder.reset();

adder.sum();
adder.sumThenReset();
```

26

LongAdder



27

LongAccumulator[8]

- Customized 함수가 추가가능한 다중스레드 환경기반 Atomic class
  - AtomicLong과 비교해서 높은 update contention이 있는 환경에서 메모리 공간을 더 소비함으로써 높은 성능으로 동작

```
LongAccumulator(LongBinaryOperator accFunc, long identity)
```

```
var a = new LongAccumulator(Long::sum, 0);
a.get()
a.intValue(); a.longValue();
a.floatValue(); a.doubleValue();

a.accumulate(45);
a.getThenReset();
a.reset();
```

28

정리

- Atomic
- LongAdder
- LongAccumulator

29

Stamped Lock

sanghyuck.na@lge.com

+Phaser

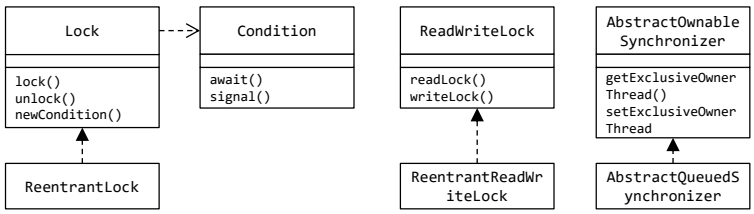
```
synchronized (this) {
}

Object obj = new Object();
synchronized (obj) {
}
```

```
private void shutdown(ExecutorService executor) {try
{executor.shutdown(); executor.awaitTermination(60,
TimeUnit.SECONDS); } catch (InterruptedException e)
{ e.printStackTrace(); } finally { if
(!executor.isTerminated()) { System.err.println("killing non-
finished tasks");} executor.shutdownNow(); } } public void
sleep(long t) { try { Thread.sleep(t); } catch
(InterruptedException e) { e.printStackTrace(); }}
```

30

JAVA Lock

- 특정한 조건을 잠그고 기다리는 클래스와 인터페이스 모음
  - Object class의 내장 동기화 및 모니터 wait()와 notify() 와는 별개로 Framework로 제공
- Interface Lock[5] ReentrantLock
  - 기본 Lock 인터페이스로 타스크재정렬 알고리즘이 포함되어 nonblock코드에서 사용
- Interface ReadWriteLock[5] ReentrantReadWriteLock
  - Reader간 공유되는 read lock, Writer는 독자적인 점유하는 write lock 제공
- Interface Condition[5]
  - Object class의 모니터메소드(wait, notify, notifyAll)를 객체로 분리한 것으로 객체 당 다중Wait-set을 구현할 경우 사용
- AbstractQueuedSynchronizer[5]
  - 대기큐(FIFO) 방식으로 동작하는 코드의 Lock과 Release 구현에 Code 제공



31

Stamped Lock[8]

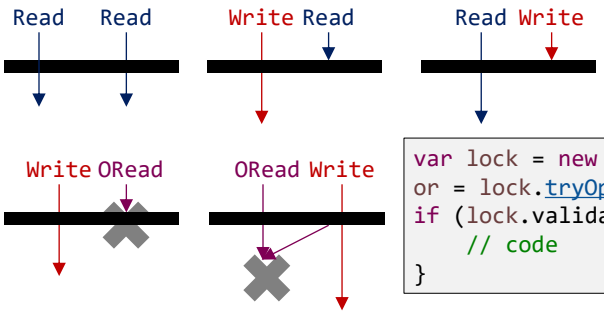
- Read, Write제어를 하기 위한 3가지 모드를 timestamp로 제어하는 Lock
  - 상태는 버전과 모드로 구성됩니다. Lock을 성공하면 timestamp를 반환합니다. 만약 try lock으로 시도하여 실패한다면 0값을 반환해서 '실패'를 나타낼 수 있습니다.
  - writeLock, readLock, tryOptimisticRead 3가지 모드, 성공적인 lock 획득은 non-zero반환
- Optimistic readLock
  - Write lock에 의해 언제라도 무효화 될 수 있는 Weak read lock
  - Readlock 가능여부를 확인을 위한 방법으로, validate()를 사용하여 lock유효성검증
  - contention에서 최대한 피해서 처리율 개선

```
long readLock(), void unlockRead(long stamp)
long writeLock(), void unlockWrite(long stamp)
long tryConvertToReadLock(long stamp)
long tryConvertToWriteLock(long stamp)

long tryOptimisticRead()
long tryConvertToOptimisticRead(long stamp)

boolean validate(long stamp)
```

32

Stamped Lock[8]

- Memory Synchronization
  - 가장 최근 write mode unlock의 전방코드는 그 뒤에 따라오는 OptimisticRead 획득부분에서 validate()가 true를 리턴 한 경우 그 코드가 실행 됐고 메모리 동기화도 완료된 것이 보장됩니다. JAVA는 lock과 unlock시점에 메모리동기화됨
- valdiate() false
  - tryOptimisticRead()와 validate()사이에 동일한 snapshot이 아닌 것으로 메모리동기화가 되지 않을 수 있기 때문에 항상 정확한 읽기는 readlock획득후직압




```
var lock = new StampedLock();
or = lock.tryOptimisticRead();
if (lock.validate(or)) {
    // code
}
```

33

Stamped Lock[8]

- 실제 데이터read접근은 readlock 획득한 다음 접근

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = 0L;

    stamp = lock.tryOptimisticRead();

    if (lock.validate(stamp)) {
        stamp = lock.readLock();
       // data access
        unlockRead(stamp);
    }
});
```

34

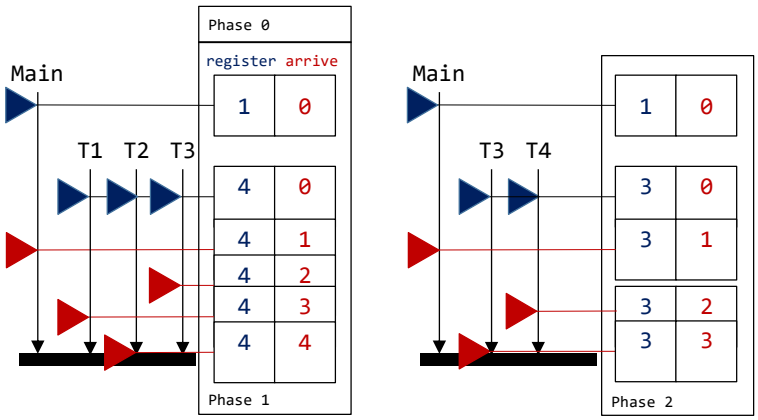
Synchronizer

- Semaphore[5]
  - acquire(), release()로 허용개수 permits 까지만 접근 허용
- CountDownLatch[5]
  - countdown()으로 주어진 count를 0으로 차감하여 await()로 대기중인 스레드 해제
- CyclicBarrier[5]
  - 주어진 조건(barrier)을 만족할 때까지 모든스레드를 대기시키고, 해제하는 기법
- Exchanger[5]
  - 두 스레드가 주어진 지점(exchange())에서 객체교환하는 기법
- Phaser[7]
  - 스레드의 등록갯수(register())와 도착갯수(arrive())가 일치 할 때 까지 대기하고, 개수가 일치하는 순간 그 모두 실행이 된다.
  - 내부적으로 동기화진입 단계 (phase) 유지. 0부터 시작

39

Phaser[7]

- register
  - 동기화 하려는 스레드 개수를 +1 증가시킴. bulkRegister(int)로는 2이상 한번에 등록
- arriveAndAwaitAdvance(= CyclicBarrier.await())
  - 모든 스레드의 도착을 대기하는 지점으로, 대기가 풀리면 phase는 1증가



40

Phaser[7]

- parties
  - 생성자의 parties갯수는 초기 register 개수
- register
  - 대기스레드에서 호출하여 등록갯수 증가
- arrive
  - Non-block메소드로 arriv갯수만 증가시키고 바로 다음코드를 실행 합니다.
  - 다른 모든 스래드와 동기화를 위해서는 arriveAndAwaitAdvance()를 호출합니다.
  - 호출 후 register와 arrive개수 리셋은 각 스레드에서 arriveAndDeregister()호출

```
Phaser(int parties)
int register()

void arrive()
void arriveAndAwaitAdvance()
void arriveAndDeregister()

protected boolean onAdvance(int phase, int registeredParties)
```

41

도전하세요!

- Phaser[7] 실습코드에서 register() 대신 bulkRegister()를 사용하도록 코드를 바꾸세요

42

정리

- StampedLock
- Phaser

45