Distributed System Homework2 Report

2017-11394 김형모

2017년 3월 27일

1 Exersices

1.1 Exersice 15

스레드 0과 스레드 1이 다음과 같이 concurrent하게 `while` 스핀락을 통과할 수 있다.

$$read_0(y == -1) \rightarrow read_1(y == -1) \rightarrow ...$$

이 경우 $x == i$일 때, 스레드 i는 `lock.lock()`을 수행하지 않고 Critical-Section으로 진행하고, 스레드 $1-i$는 `lock.lock()`을 수행하고 Critical-Section으로 진행하게 되므로 두 스레드가 Critical-Section을 동시해 실행할 수 있게 된다. 따라서 틀린 알고리즘이다.

1.2 Exersice 27

스레드 A, B, C가 다음과 같은 실시간 순서로 각각 `enq(a)`, `enq(b)`, `deq()`를 수행한다고 하자.[1]

A, B...

A : `slot = tail.get()` : `tail == 0` $\rightarrow$

A : `tail.compareAndSet(slot, slot+1)` : `tail == 1` $\rightarrow$

B : `slot = tail.get()` : `tail == 1` $\rightarrow$

B : `tail.compareAndSet(slot, slot+1)` : `tail == 2` $\rightarrow$

B : `items[slot] = b` : `slot == 1` $\rightarrow$

B : `return enq` $\rightarrow$

[1] 초기값은 `head == 0`, `tail == 0`, `items = { null, ...}`이다.

C : `call deq` →

C : ...

C : `slot = head.get()` : `head == 0` →

C : `value = items[slot]` : `slot == 0` →

C : `if (value == null)` : `value == null` →

C : `throw new EmptyException()`

이때 B의 `enq()`가 완전히 종료된 후 C의 `deq()`에서 `EmptyException()`을 호출하는 legal sequential subhistory가 존재하지 않는다. Sequential이고 `deq()`가 단 한 번만 있는 경우, B의 `enq()`가 끝난 후에 C의 `deq()`를 호출했을 때 반드시 올바른 값을 반환하기 때문이다. 따라서 linearizable하지 않다.

1.3 Exersice 28

스레드 A가 `writer()` 메서드를, 스레드 B가 `reader()` 메서드를 호출하는 상황에서 $write_A(x = 42)$와 $write_A(v = \text{true})$가 스레드 A와 스레드 B의 메모리에 반영되는 순서를 보장할 수 없다.

1. v는 `volatile`로 선언되었으나 x는 `volatile`로 선언되지 않았으므로, 스레드 B의 메모리에서 v의 값이 true가 된 시점에서 x의 값이 스레드 A의 캐시와 동기화되었는지는 알 수 없다.

2. x와 v는 서로 의존성이 없는 변수이므로, 스레드 A가 실행될 때 x와 v의 값이 어떤 순서로 확정될 지 알 수 없다. 따라서 스레드 B에서도 x와 v의 값이 어떤 순서로 확정될 지 알 수 없다.

그러므로, $write_A(v = \text{true}) \rightarrow read_B(v == \text{true}) \rightarrow read_B(x == 0)$의 순서로 실행되면 `reader()` 메서드에서 `devided-by-zero` 에러가 발생할 수 있다.

1.4 Exersice 31

i-th 메서드 `m`은 invocation이 일어난지 2^i step 후에 반드시 responce가 일어나므로, `wait-free`라고 할 수 있다.

1.5 Exersice 32

1.5.1 Give an example execution showing that the linearization point for `enq()` cannot occur at Line 15.

Line 15: `tail.getAndIncrement()`가 linearization point라고 가정하면,

A : `tail.getAndIncrement()` →

B : `tail.getAndIncrement()` →

B : `items[i].set(b)` →

C : `deq() == b`

따라서 C의 `deq() == a`가 아니므로, linearization point가 아니다.

1.5.2 Give another example execution showing that the linearization point for `enq()` cannot occur at Line 16.

Line 16: `items[i].set(x)`가 linearization point라고 가정하면,

A : `tail.getAndIncrement()` →

B : `tail.getAndIncrement()` →

B : `items[i].set(b)` →

A : `items[i].set(a)` →

C : `deq() == a`

따라서 C의 `deq() == b`가 아니므로, linearization point가 아니다.

1.5.3 Since these are the only two memory accesses in `enq()`, we must conclude that `enq()` has no single linearization point. Does this mean `enq()` is not linearizable?

`tail.getAndIncrement()`와 `items[i].set(x)`는 의존성이 있어 한 스레드 내에서의 순서는 보장되지만, atomic 연산이 아니다.

- `tail.getAndIncrement()`
- `items[i].set(x)`
- `value = items[i].getAndSet(null)`

서로 다른 스레드에서 위의 Line들이 예시에서처럼 나열되면, `enq()` 메서드내의 임의의 객체 상태를 변화시키는 Line이 linearization point이 되지 않는 예외가 항상 존재하므로 linearizable하지 않다.