习题答疑

第一、二、三、四、五章

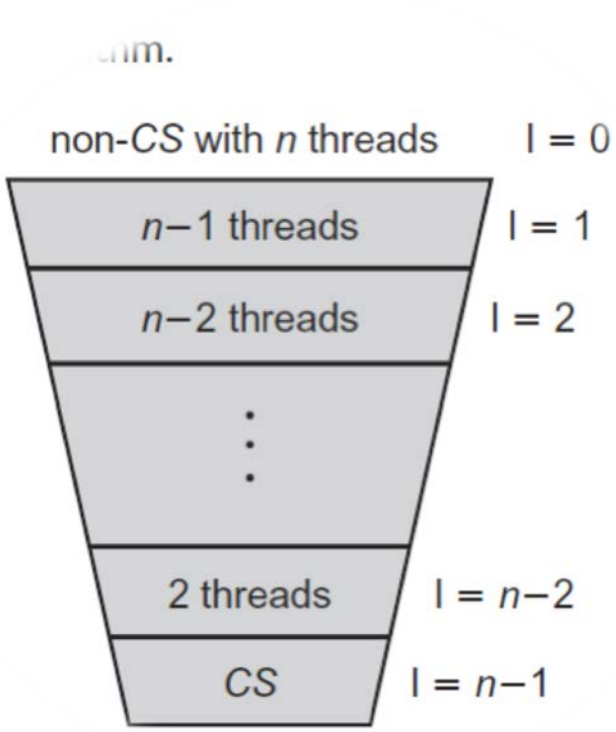
第一章 Safety & Liveness

- Patrons are served in the order they arrive.
- If an interrupt occurs, then a message is printed within one second.
- The cost of living never decreases.
- You can always tell a Harvard man.

- What goes up must come down.
- If two or more processes are waiting to enter their critical sections, at least one succeeds.
- If an interrupt occurs, then a message is printed.
- Two things are certain: death and taxes.

第二章 Filter Lock

- **证明**过滤器锁允许一些线程超越其它线程任意多次
  - 证明方法：举例说明
- 超越（overtake）：反复获得锁、再释放锁
- level的“跨越”




```
public void lock() {
  int me = ThreadID.get();
  for(int i=1; i<n; i++) { //attempt level i
    level[me] = i;
    victim[i] = me;
    // spin while conflicts exist
    while ((∃k != me) (level[k] >= i && victim[i] == me)) { };
  }
}
```

第二章 Filter Lock

Given threads A, B, and C, C can overtake A an arbitrary number of times.

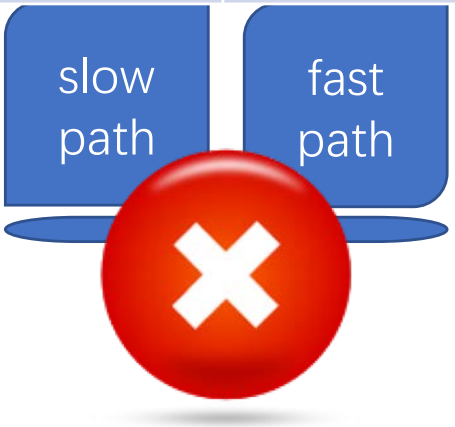
1. C acquires the lock.
2. A calls lock(), becoming the victim at level 1.
3. B calls lock(), becoming the victim at level 1.
4. C releases the lock, calls lock() again, and becomes the victim at level 1.
5. B acquires and releases the lock.
6. B calls lock(), becoming the victim at level 1.
7. C acquires the lock.

Repeat 4-7 until A wakes up and moves on.

第二章 FastPath

```
1  class FastPath implements Lock {
2    private static ThreadLocal<Integer> myIndex;
3    private Lock lock;
4    private int x, y = -1;
5    public void lock() {
6      int i = myIndex.get();
7      x = i;                    // I'm here
8      while (y != -1) {}        // is the lock free?
9      y = i;                    // me again?
10     if (x != i)               // Am I still here?
11       lock.lock();            // slow path
12   }
13   public void unlock() {
14     y = -1;
15     lock.unlock();
16   }
17 }
```

Thread 1 (myIndex=1)	Thread 2 (myIndex=2)
i = 1	i = 2
x = 1	
	x = 2
<y == -1>	<y == -1>
y = 1	
	y = 2
<x != i>	<x == i>
lock.lock()	
return	return



第三章

Wait-Freedom

- A concurrent object implementation is *wait-free* if each method call finishes in a finite number of steps.

- A method is *lock-free* if it guarantees that **infinitely often** *some* method call finishes in a finite number of steps.

Lock-Freedom

- Lock-free algorithms admit the possibility that some threads could starve.

History

- An execution of a concurrent system is modeled by a *history*, a **finite** sequence of method *invocation* and *response events*.
- An *infinite history* is an **infinite** sequence of method invocation and response events.
- Pending invocations may be included in a (finite) history or an infinite history.

第29题：

无等待

wait-freedom

- **Read the questions carefully!**
- ***Exercise 29.*** Is the following property equivalent to saying that object x is wait-free?
- For every infinite history *H* of *x*, every thread that takes an infinite number of steps in *H* completes an infinite number of method calls.
- 一个执行无穷步的方法调用是没有返回的，因而是未决的（pending）。这样的方法调用在一个执行（历史）中不会发生无穷个（前提是线程数是确定的、有限的）。

第29题：

无等待

wait-freedom

- ***Exercise 29.*** Is the following property equivalent to saying that object x is wait-free?
- For every infinite history *H* of *x*, every thread that takes an infinite number of steps in *H* completes an infinite number of method calls.
- 对于wait-free的对象而言，上述命题成立，但反之不成立。
- 有穷的历史仍可能包含未决的方法调用，这些调用没有在有穷步内完成。上述命题未排除这种情况。

第30题：

无锁

lock-freedom

- ***Exercise 30.*** Is the following property **equivalent** to saying that object *x* is lock-free?
- For every infinite history *H* of *x*, an infinite number of method calls are completed.
- 命题含义：每个无穷历史都完成了无穷多个方法调用。
- 一个对象是**lock-free**的，如果它的任意执行都是**lock-free**的，即（**infinitely often**）总有线程能够在有穷步内完成一个方法调用。

第30题：

无锁

lock-freedom

- ***Exercise 30.*** Is the following property **equivalent** to saying that object *x* is lock-free?
- For every infinite history *H* of *x*, an infinite number of method calls are completed.
- 若对象x是lock-free的，则对对象x的任意无穷历史而言，总有线程能够在有穷步内完成一个方法调用，即能够完成无穷多个方法调用。所以，上述命题成立。但反之不成立。
- 对象x是lock-free的，意味着它不存在这样的有穷历史，其中每个线程都包含一个未决的方法调用。上述命题未排除这种情况。

Execution

- For every infinite execution *E* of *x*, every thread that takes an infinite number of steps in *E* completes an infinite number of method calls.
- (wait-free?)

- For every infinite execution *E* of *x*, an infinite number of method calls are completed.
- (lock-free?)

第32题

```
1  public class HWQueue<T> {
2    AtomicReference<T>[] items;
3    AtomicInteger tail;
4    static final int CAPACITY = 1024;
5
6    public HWQueue() {
7      items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8          CAPACITY);
9      for (int i = 0; i < items.length; i++) {
10       items[i] = new AtomicReference<T>(null);
11     }
12     tail = new AtomicInteger(0);
13   }
14   public void enq(T x) {
15     int i = tail.getAndIncrement();
16     items[i].set(x);
17   }
18   public T deq() {
19     while (true) {
20       int range = tail.get();
21       for (int i = 0; i < range; i++) {
22         T value = items[i].getAndSet(null);
23         if (value != null) {
24           return value;
25         }
26       }
27     }
28   }
29 }
```

- i是局部变量。
- 不同线程不会写到同一个位置。
- getAndIncrement 返回的是Increment之前的值。
- “合并成一个原子操作”、“设置为临界区”、加锁等都修改了enq方法本身。

Figure 3.17 Herlihy/Wing queue.

第32题： enq()

- FIFO定义了正确性要求。
- 一个方法是可线性化当且仅当这个方法的所有历史都是可线性化，而不是特定的某些历史。
- 可线性化一定有线性化点。但线性化点可以不固定在某一行，可能在15行，也可能在16行。
- 每个历史都是可线性化的，可以按照deq的顺序确定enq的线性化点。
- 线性化点指的是发生作用的点，不在于有多少个操作。

第33題：

Nonblocking

A pending invocation of a total method is never required to wait for another pending invocation to complete.

Theorem: Let *inv*(*m*) be an invocation of a total method m. If <*x inv P*> is a pending call of *m* in a sequential consistent history *H*, then there exists a response <*x res P*> such that *H*·<*x res P*> is sequential consistent.

- Proof:
- If method *m* has taken effect, but not yet returned a response in *H*, then since *H* is sequential consistent, there exists a sequential consistency sequence *S* for *H* such that *S* involves the response . Therefore, *S* is also a sequential consistent sequence for the history *H*·<*x res P*> .
- If method *m* has not yet taken effect in *H*. By definition of sequential consistency, there exists a sequential consistency sequence *S* for *H* such that *S* does not involve the pending invocation. Because method *m* is total, there exists a response such that *S*′ = *S*·<*x inv P*>·<*x res P*>. Therefore, *S*′ is a sequential consistency sequence for *H*·<*x res P*>.

第四章

Exercise 42. You learn that your competitor, the Acme Atomic Register Company, has developed a way to use Boolean (single-bit) atomic registers to construct an efficient *write-once* single-reader single-writer atomic register. Through your spies, you acquire the code fragment shown in Fig. 4.23, which is unfortunately missing the code for `read()`. Your job is to devise a `read()` method that works for this class, and to justify (informally) why it works. (Remember that the register is write-once, meaning that your read will overlap at most one write.)

```
1  class AcmeRegister implements Register{
2    // N is the length of the register
3    // Atomic multi-reader single-writer registers
4    private BoolRegister[] b = new BoolMRSWRegister[3 * N];
5    public void write(int x) {
6      boolean[] v = intToBooleanArray(x);
7      // copy v[i] to b[i] in ascending order of i
8      for (int i = 0; i < N; i++)
9        b[i].write(v[i]);
10     // copy v[i] to b[N+i] in ascending order of i
11     for (int i = 0; i < N; i++)
12       b[N+i].write(v[i]);
13     // copy v[i] to b[2N+i] in ascending order of i
14     for (int i = 0; i < N; i++)
15       b[(2*N)+i].write(v[i]);
16   }
17   public int read() {
18     // missing code
19   }
20 }
```

left

middle

right

overlap on only one of the three copies.

N 是寄存器的长度，不是线程数。

Figure 4.23 Partial acme register implementation.

第四章 第42题

Idea I:

- **If** the left and middle copies are the same, then the overlap occurred on the right copy, so take the left copy.
- **If** the right and middle copies are the same, then the overlap occurred on the left copy, so take the right copy.
- **Otherwise**, the overlap occurred on the middle copy, so take the right copy.

第四章 第42题

Idea II: b[N..2N-1] and b[0..N-1] are repeatedly read, until the outcomes coincide.

- Because a read can only have clashed with the write while reading either b[N..2N-1] or b[0..N-1].
- So if the outcomes coincide, we can be certain the outcome represents either the original or the written value in the register.

第五章

Exercise 53. The `Stack` class provides two methods: `push`(x) pushes a value onto the top of the stack, and `pop`() removes and returns the most recently pushed value. Prove that the `Stack` class has consensus number *exactly* two.

We need to show that the Stack class can solve consensus for two threads, but not three.

1. Both threads announcing their values in an array.
2. The stack is initialized with two items: first we push the value LOSE, and then the value WIN.
3. In the decide() method, each thread pops an item from the stack.
4. The thread that pops WIN decides its own value, and the thread that pops LOSE decides **the other** thread's value.

第五章 第53题

- The proof that consensus is impossible with three threads is a valence argument like the one used for the Queue class.
- Suppose, towards a contradiction, that there is a wait-free consensus protocol for three threads.
- By trying all the combinations of pop() and push(), one can show that the third thread could not tell which method call came first.

第五章 第53题

Given threads A, B and C. Consider a critical state s. Let a move from A lead to decision 0, and a move from B to decision 1.

1. Let A and B perform methods on different stacks. ……
2. Let A and B do pops on the same stack. ……
3. Let A push *a* onto and B pop from the same stack. ……
4. Let A pop and B pushes *b* onto the same stack. ……
5. Let A push *a* and B push *b* onto the same stack. ……

✓ *All cases contradict the fact that s is critical.*

第五章

Exercise 54. Suppose we augment the FIFO `Queue` class with a `peek()` method that returns but does not remove the first element in the queue. Show that the augmented queue has infinite consensus number.

- Threads share a public announce[] array.
- In the propose() method, each thread with its index A writes its proposed value to announce[A].
- In the decide() method, each thread first calls enq(A), and then calls peek(), which returns some B.
- The thread then decides the value in announce[B].

第五章 第54题

This is correct because

1. If B is the first index to be enqueued, later enq() calls will not change B's position,
2. B stored its value in announce[B] before calling enq(B).

第五章

Exercise 68. Fig. 5.17 shows a FIFO queue implemented with read, write, `getAndSet()` (that is, swap) and `getAndIncrement()` methods. You may assume

```
1  class Queue {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicReference items[] =
4      new AtomicReference[Integer.MAX_VALUE];
5    void enq(Object x){
6      int slot = head.getAndIncrement();
7      items[slot] = x;
8    }
9    Object deq() {
10     while (true) {
11       int limit = head.get();
12       for (int i = 0; i < limit; i++) {
13         Object y = items[i].getAndSet();
14         if (y != null)
15           return y;
16       }
17     }
18   }
19 }
```

Figure 5.17 Queue implementation.

this queue is linearizable, and wait-free as long as `deq()` is never applied to an empty queue. Consider the following sequence of statements.

- Both `getAndSet()` and `getAndIncrement()` methods have consensus number 2.
- We can add a `peek()` simply by taking a snapshot of the queue (using the methods studied earlier in the course) and returning the item at the head of the queue.
- Using the protocol devised for Exercise 54, we can use the resulting queue to solve n-consensus for any n.

We have just constructed an n-thread consensus protocol using only objects with consensus number 2. Identify the faulty step in this chain of reasoning, and explain what went wrong.

第五章 第68题

The problem is that the snapshot does not tell you the queue's actual state. Start with an empty queue.

1. The first thread does an *enq* and increments the head to make room for its value, but does not write its value, leaving that spot null.
2. A second thread runs and completely finishes enqueuing its value.
3. The second thread calls *peek* by taking a snapshot. It finds the space that the first thread made is still null, so it thinks that the second *enq'* s value is at the head of the queue and decides on this value.
4. The first thread wakes up, finishes writing its value and calls *peek* on a snapshot to see its value is the first and decides on it, a different value.