CS5510: Multiprocessor Programming : Homework III

Sergio Sainz

October 2019

1 Part I : Theory

1.1 [7 Points] Java memory model

Consider the class shown in listing 1. According to what you have been told about the Java memory model, will the reader method ever divide by zero?

```
1  class VolatileExample{
2      int x = 0 ;
3      volatile boolean v = false;
4      public void writer (){
5          x = 42;
6          v = true;
7      }
8      public void reader(){
9          if(v == true){
10             int y = 100/x;
11         }
12     }
13 }
```

Answer:
The java object model does not warranty sequential consistency (much less linearizability). Reasons is that supporting such consistencies would prohibit compiler optimizations. Still, Java Memory Model allows for sequential consistency in three ways: volatile fields, final fields and **synchronized** code section. In the listing above we see that v is volatile, this means that every read is read from main memory (and not from thread's working memory) and every write is written to main memory (and not to thread's working memory).
But, x, is not volatile. And because a thread calling writer is not warrantied sequential consistency in the instructions $x = 42$, $v = true$. It could be that one thread, A, calls writer method and the JVM chooses to execute $v = true$ before $x = 42$. Then another thread, B, could take over after A executes $v = true$ and

executes reader method all the way until the division of $100/x$, because x is still having value of 0, then division between zero is possible. More formal:

$$write_A(v = true) \rightarrow read_B(v = true)$$
$$divide_B(100/x, \mathrm{x} = 0)$$

1.2 Sequential Consistency (1) [7 points]

For the following executions, please indicate if they are sequentially consistent. All variables are intially set to 0.

Sequential consistency complies with two principles: (1) Method calls should appear to occur in a one at the time sequential order. (2) Method call should appear to take effect in program order.
In other works sequential execution requires that method calls act as if they occurred in a sequential order consistent with program order. That is, in any concurrent execution there is a way to order the method calls sequentially so that they (1) are consistent with program order, and (2) meet the object's sequential specification.

1. P1: W(x,1);
   P2: R(x,0); R(x,1);
   Answer they are sequential consistent: R(x,0); → P1: W(x,1); → R(x,1);

2. P1: W(x,1);
   P2: R(x,1); R(x,0);
   Answer NOT sequential consistent.

3. P1: W(x,1);
   P2: W(x,2);
   P3: R(x,1); R(x,2);
   Answer Sequential consistent. W(x,1); → R(x,1); → W(x,2); → R(x,2);

4. P1: W(x,1);
   P2: W(x,2);
   P3: R(x,2); R(x,1);
   Answer Sequential consistent. W(x,2); → R(x,2); → W(x,1); → R(x,1);

5. P1: W(x,1);
   P2: W(x,2);
   P3: R(x,2); R(x,1);
   P4: R(x,1); R(x,2);
   Answer NOT sequential consistent

6. P1: W(x,1); R(x,1); R(y,0);
   P2: W(y,1); R(y,1); R(x,1);
   P3: R(x,1); R(y,0);
   P4: R(y,0); R(x,0);
   Answer Sequential consistent. R(y,0); → R(x,0); → W(x,1); → R(x,1); → R(y,0); → R(x,1); → R(y,0); → W(y,1); → R(y,1); → R(x,1);

7. P1: W(x,1); R(x,1); R(y,0);
   P2: W(y,1); R(y,1); R(x,1);
   P3: R(y,1); R(x,0);
   Answer NOT sequential consistent.

1.3 Sequential Consistency (2) [6 points]

- P1
  x = 1;
  print(y,z);

- P2
  y = 1;
  print(x,z);

- P3
  z = 1;
  print(x,y);

All variables are stored in a memory system which offers sequential consistency. All operations, even the print statements, are atomic (no operation can overlap with other operations). Are the following sequences legal outputs?

1. 001011
   Answer Valid output. $x = 1 \rightarrow print(y, z); \rightarrow y = 1 \rightarrow print(x, z); \rightarrow z = 1 \rightarrow print(x, y);$

2. 001111
   Answer Valid output. $x = 1 \rightarrow print(y, z); \rightarrow y = 1 \rightarrow z = 1 \rightarrow print(x, z); \rightarrow print(x, y);$

3. 001110
   Answer Not valid. We cannot produce valid sequence for this output because in last print statement of this output we see all other thread's output to have been printed. And because the system offers sequential consistency that means that program order is respected. The set to memory operations are done before the print operations. Thus, by the time the last print operation executes, all set operations should have been set. All x, y, z, are already set to 1. Therefore last print (10) is not possible.

Explain your answer by showing one possible interleaving of the instructions that might lead to the legal output. In case of an illegal output, explain why no possible interleaving exists.

1.4 Consistency (1) [8 points]

Consider a memory object that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory object. Does the converse hold? If the memory object is quiescently consistent, are the individual registers quiescently consistent? Outline a proof or give a counterexample.

Answer Does not hold.

Consider the following history for stack s ($H|s$) (notation: (thread, method, argument)):

$$\{inv(1, s.push, 1), inv(2, s.push, 2), inv(3, s.pop, 3), return(2, s.push, 2),$$
$$return(3, s.pop, 3), inv(3, s.pop, 2), inv(2, s.push, 3), return(3, s.pop, 2),$$
$$return(2, s.push, 3), return(1, s.push, 1)\}$$

History $H|s$ is quiescent consistent.

Now consider that stack method push sets the counter c to the length of the stack (by increasing one to the existing counter c), and the register p is set to the last thread number accessing the stack.

And method pop also sets the counter c to the length of the stack after removal (by decreasing previous value of c) and register p also is set to the last thread accessing the stack.

Then, we have $H|p$:

$$\{inv(1, s.push(1) : s.p = 1), inv(2, s.push(2) : s.p = 2),$$
$$inv(3, s.pop(3) : s.p = 3), return(2, s.push(2) : s.p = 2),$$
$$return(3, s.pop(3) : s.p = 3), inv(3, s.pop(2) : s.p = 3),$$
$$inv(2, s.push(3) : s.p = 2), return(3, s.pop(2) : s.p = 3),$$
$$return(2, s.push(3) : s.p = 2), return(1, s.push(1) : s.p = 1)\}$$

History $H|p$ is quiescently consistent.

Now let us look at $H|c$:

$$\{inv(1, s.push(1) : s.c = 1), return(1, s.push(1) : s.c = 1),$$
$$inv(2, s.push(2) : s.c = 2), inv(3, s.pop(3) : s.c = 2),$$
$$return(2, s.push(2) : s.c = 2), return(3, s.pop(3) : s.c = 2),$$
$$\text{quiet},$$
$$inv(3, s.pop(2) : s.c = 1), inv(2, s.push(3) : s.c = 3),$$
$$return(3, s.pop(2) : s.c = 1), return(2, s.push(3) : s.c = 3)\}$$

$H|c$ is NOT quiescently consistent. We can explain reason in the part of history after the *quiet* segment. After quiet segment, we can see two calls: thread 3 calls s.pop() to pop integer 2 from the stack then decreases the count to 1 (length of stack minus one, previously stack length was 2 because thread 3 made a pop of integer 3 and decreased length from 3 to 2 before quiet period). Then a second call by thread 2 to push integer 3 to the stack and increase counter to 3 (length of stack previously was 2 after push of integer 2 by thread 2 before the quiet period). As we can see the length of stack (c) cannot be both 3 and 1 at the end of both calls, therefore , one of them must be wrong, breaking the quiescence of the registry (or if we allow registry c to take 1 or 3, then the legal sequence specification for *Stack* is violated).

1.5 Consistency (2) [7 points]

Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

Answer, example of history that is quiescence consistent but not sequential consistent:

$$\{inv(1, dequeue, x), inv(2, enqueue, y),$$
$$return(1, dequeue, x), inv(1, enqueue, x),$$
$$return(2, enqueue, y), return(1, enqueue, x)\}$$

As we can see enqueue of x happens after dequeuing of x in program order. But because quiescence does not require to follow program order as long as there is activity in the memory being modified then we can find a sequence that breaks program order that complies with queue object and then history still complies with quiescence as long as there is no quiet periods between enqueue and dequeue.

Answer example of history that is sequential consistent but no quiescence consistent:

$$\{inv(1, enqueue, x), return(1, enqueue, x),$$
$$quiet$$
$$inv(1, dequeue, y), inv(2, enqueue, y),$$
$$return(1, dequeue, y), return(2, enqueue, y)\}$$

As we can see the history above is sequential consistent because we can move the enqueueing of y by thread 2 before the enqueueing of x by thread 1 without breaking program order and still having a FIFO queue valid sequence.
Meanwhile history is not quiescence consistent because the enqueueing of x must happen *before* enqueuing of y due to requirement of quiescence consistency that method calls separated by period of quiescence respect real-time-order. Thus the sequence should be $enq(x) \rightarrow enq(y) \rightarrow deq(y)$ which is invalid sequence according with FIFO queue.

1.6 Linearizability (1) [7 points]

For the following history of a shared register with the operations write(x)/void and read()/x, answer the questions below.
B: r.write(1)
A: r.read()
C: r.write(2)
A: r:1
B: r:void
C: r:void
B: r.read()
B: r:1
A: q.write(3)
C: r.read()
A: q:void

1. What is $H|B$?
   B: r.write(1)
   B: r:void
   B: r.read()
   B: r:1

2. What is $H|r$?
   B: r.write(1)
   A: r.read()
   C: r.write(2)

A: r:1
B: r:void
C: r:void
B: r.read()
B: r:1
C: r.read()

3. Turn H into a complete subhistory H'.
   We remove the pending invocation that has not yet taken effect. (C: r.read()). In that way the new history, G, is complete (all invocations have response events):
   B: r.write(1)
   A: r.read()
   C: r.write(2)
   A: r:1
   B: r:void
   C: r:void
   B: r.read()
   B: r:1
   A: q.write(3)
   A: q:void

4. Is H' sequential?
   No, it is not. Sequential history means that each invocation event is immediately followed by response event. In history, H, the first invocation event is not immediately followed by the response event. Thus not sequential

5. Is H' well-formed? Yes it is. For every thread, x, the thread subhistory, $H|x$, is sequential.

6. Is H' linearizable? If yes, prove it!
   Following class' method of proving history , H , is linearizable. Step (1) make the history, H, complete, such as H' (remove unapplied invocations, or extend with response events at end). Step (2) Transform the complete history, H', into linearizable history , S, such that $S \equiv H'$, and S is (a) legal, (b) sequential, (c) precedence in H' must be true in S.
   Step 1: reorder H' to S:
   C: r.write(2)
   C: r:void
   B: r.write(1)
   B: r:void
   A: r.read()
   A: r:1
   B: r.read()

B: r:1
A: q.write(3)
A: q:void

Step 2: check that **equivalence** is maintained from H' to S. Because each thread's histories are the same then equivalence is maintained ($H' \equiv S$).
Sequential : History S is sequential, each invokation event is followed by response event.
Legal: Registry preconditions and postconditions are maintained.
Precedence: Every pair of methods representing precedence relationship are maintained from H' to S. These are $Br.write(1) \rightarrow Br.read()$, $Br.write(1) \rightarrow Aq.write(3)$, $Ar.read() \rightarrow Br.read()$, $Ar.read() \rightarrow Aq.write(3)$, $Cr.write(2) \rightarrow Br.read()$, $Cr.write(2) \rightarrow Aq.write(3)$.
All four constrains are covered (equivalent, sequential, legal and precedence), thus H' is linearizable.

7. If the first two events are swapped, is the resulting history equivalent to H?.
Answer: Here I take it you mean the original history , H , if so, then yes, resulting history is equivalent (the history of the threads remains unchanged).

1.7 Linearizability (2) [7 points]

Is the following history of a FIFO queue with the operations enq(x)/void deq()/x linearizable? If yes, prove it! Is it sequentially consistent?
A: r.enq(x)
A: r:void
B: r.enq(y)
A: r.deq()
B: r:void
A: r:y

Answer History above is not linearizable because event, A: r.enq(x), has precedence with , B r.enq(y). Also event A: r.enq(x), has precedence with , A r.deq()/y. For the history to be legal event B r.enq(y) must have happened before A: r.enq(x). But because of precedence relationship $A : r.enq(x) \rightarrow B : r.enq(y)$ exists it is not possible for event B: r.enq(y) must have happened before A: r.enq(x). Thus a we cannot generate a legal history that maintains precedence with H.
History , H, is sequentially consistent. We can assume thread B enqueues y before thread A enqueues x without breaking program order for neither thread A nor B. Thus producing legal history that is sequentially consistent.

1.8 Linearizability (3) [7 points]

Is the following history of a fo queue with the operations enq(x)/void deq()/x linearizable? If yes, prove it!
A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y

Answer It is not linearizable because even initial history is not legal (two threads, A and C, both dequeue y . Each object in the queue is dequeued once is part of the queue sequential specification). Therefore we cannot expect there will be a legal linearization from initial history.

1.9 Compositional Linearizability [7 points]

Prove the "only if" part of Theorem 3.6.1, reprinted below. H is linearizable if, and only if, for each object x, $H|x$ is linearizable.
Answer Assume then the contrary: A history H is linearizable even if the history of one invalid object $H|i$ is not linearizable.
History of invalid object $H|i$ is not linearizable in a couple of cases: (1) When the sequential history $S|i$ derived from $H|i$ does not complies with legal sequential specification of the object once the events are sequential and equivalent with original history $H|i$. (2) When the legal sequential history $S|i$ derived from $H|i$ does not follow precedence in $H|i$.
Then, let $H|\neg i$ the set of histories of all other , linearizable, objects. Then assume $S|\neg i$ is the sequential, equivalent, precedent and legal history produced from $H|\neg i$. Then we have two cases: if history $H|i$ has a sequential history that is legal but does not follow precedence in $H|i$. And second case when $S|i$ is sequential and follow precedence in $H|i$ but is not legal.

In both of these cases , append $H|i$ at the end of $H|\neg i$. Let H be the union of both $H|\neg i$ and $H|i$. H cannot be linearizable because either sequential history S won't follow precedence of H (in object i) or S won't have a legal sequence in object i. Thus a contradiction.

1.10 More Histories (1) [7 points]

Is this a linearizable execution?

Answer Yes it is. Valid linearizable history S (equivalent to original history H) is as follows:
A inv(q.enq(x))
A res(q.enq(x))

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	
2	Work on `q.enq(x)`	
3	Return from `q.enq(x)`	
4		Invoke `q.enq(y)`
5		Work on `q.enq(y)`
6		Work on `q.enq(y)`
7		Return from `q.enq(y)`
8		Invoke `q.deq()`
9		Return `x` from `q.deq()`

Figure 1: Exercise 1.10

B inv(q.enq(y))
B res(q.enq(y))
B inv(q.deq())
B res(q.deq():x)

This history S is already sequential, legal, equivalent to the original H one and maintains all precedence relationships.

1.11 More Histories (2) [7 points]

Is this a linearizable execution?

Answer Yes it is. Valid linearizable history S (equivalent to original history H) is as follows:
A inv(q.enq(x))
A res(q.enq(x))
B inv(q.enq(y))
B res(q.enq(y))
B inv(q.deq())
B res(q.deq():x)

This history S is already sequential, legal, equivalent to the original H one and maintains all precedence relationships.

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

Figure 2: Exercise 1.11

1.12 More Histories (3) [7 points]

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Return from `q.enq(x)`	
2		Invoke `q.enq(y)`
3	Invoke `q.deq()`	Work on `q.enq(y)`
4	Work on `q.deq()`	Return from `q.enq(y)`
5	Return `y` from `q.deq()`	

Figure 3: Exercise 1.11

This one cannot be linearizable. Because there is a precedence relationship between the method $A : enq(x) \rightarrow B : enq(y)$ and between $A : enq(x) \rightarrow A : deq(y)$. This creates problem that for the history to be legal and indeed dequeue y at the last method , $B : enq(y)$ must have happened before $A : enq(x)$. But, because precedence relationship $A : enq(x) \rightarrow B : enq(y)$ exists then this is not possible. Thus not linearizable.

1.13 AtomicInteger [7 points]

The AtomicInteger class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is boolean compareAndSet (int expect, int update).

This method compares the object's current value to expect. If the values are equal, then it atomically replaces the object's value with update and returns true. Otherwise, it leaves the object's value unchanged and returns false. This class also provides int get(), which returns the object's actual value.
Consider the FIFO queue implementation shown in listing 2. It stores its items in an array items, which, for simplicity, we will assume has unbounded size. It has two AtomicInteger fields: head is the index of the next slot from which to remove an item, and tail is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

```
1  class IQueue <T> {
2      AtomicInteger head = new AtomicInteger(0);
3      AtomicInteger tail = new AtomicInteger(0);
4      T[] items = (T[]) new Object[Integer.MAX_VALUE];
5      public void enq(T x) {
6          int slot;
7          do{
8              slot = tail.get();
9          } while (!tail.compareAndSet(slot, slot+1));
10         items[slot] = x;
11     }
12     public T deq() throws EmptyException {
13         T value;
14         int slot;
15         do {
16             slot = head.get();
17             value = items[slot];
18             if (value == null)
19                 throw new EmptyException();
20         } while (!head.compareAndSet(slot, slot+1));
21         return value;
22     }
23 }
```

Answer Please consider two threads in this example and also consider that Java Memory Model does not support sequential consistency. Therefore some instructions such as lines 16 and 17 can be executed in reverse order.
A inv(enq,x)
A res(enq):void
A inv(enq,y)
A res(enq):void
A inv(deq,x)
B inv(deq,x)
A working(deq,x):Read(slot=0);
B working(deq,x):Read(slot=0);
A working(deq,x):Read(value=x);
B working(deq,x):Read(value=x);

A working(deq,x):head.compareAndSet(0,1):Success;
A res(deq,x):x;
B working(deq,x):head.compareAndSet(0,1):FAIL;
B working(deq,x):Read(value=x; value=item[slot=0]);
B working(deq,x):Read(slot=1);
B working(deq,x):head.compareAndSet(1,2):Success;
B res(deq,x):x;

As we can see the problem is that the history is not legal according to queue sequential specification (the x item is returned twice and y is never going to be returned). This steams from the fact that algorithm assumes that line $slot = head.get()$; will execute before $value = items[slot]$. But because Java Memory Model does not support sequential consistency therefore program order not warrantied. It is true that head is atomic but is only atomic when accessing itself across different threads, not relative to other instructions before and after it.

1.14 Herlihy/Wing queue [9 points]

This exercise examines a queue implementation (listing 3) whose enq() method does not have a linearization point. The queue stores its items in an items array, which for simplicity we will assume never hits CAPACITY. The tail field is an AtomicInteger, initially zero. The enq() method reserves a slot by incrementing tail and then storing the item at that location. Note that these two steps are not atomic: there is an interval after tail has been incremented but before the item has been stored in the array. The deq() method reads the value of tail, then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps null with the current contents, returning the first non-null item it finds. If all slots are null, the procedure is restarted. Give an example execution showing that the linearization point for enq() cannot occur at line 14 (Hint: give an execution where two enq() calls are not linearized in the order they execute line 14.). Give another example execution showing that the linearization point for enq() cannot occur at line 15. 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?
Below is the Herlihy/Wing queue:

```
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,CAPACITY);
8          for (int i = 0; i < items.length; i++) {
9              items[i] = new AtomicReference <T>(null);
```

```
10          }
11          tail = new AtomicInteger(0);
12      }
13      public void enq(T x) {
14          int i = tail.getAndIncrement();
15          items[i].set(x);
16      }
17      public T deq () {
18          while (true) {
19              int range = tail.get();
20              for (int i=0;i<range;i++){
21                  T value = items[i].getAndSet(null);
22                  if (value != null) {
23                      return value;
24                  }
25              }
26          }
27      }
28  }
```

Answer As mentioned in the hint, below is example of execution where line 14 is executed by thread A before thread B execution of line 14. And nevertheless thread's B enqueue method is the one that gets applied first. Thus line 14 is not linearization point for enq method.
A inv(enq,x)
B inv(enq,y)
A working(enq,x):LINE 14:Write(i = tail.getAndIncrement())
B working(enq,y):LINE 14:Write(i = tail.getAndIncrement())
B working(enq,y):items[i=1].set(y)
B res(enq,y)
B inv(deq,y)
B res(deq,y)
A working(enq,x):items[i=0].set(x)
A res(enq,x)
A inv(deq,x)
A res(deq,x)

Next we explore whether Line 15 can be a linearization point: below please find example where thread A calls line 15 first before thread B and nevertheless thread B enq method gets applied first:
A inv(enq,x)
B inv(enq,y)
B working(enq,y):Write(i = tail.getAndIncrement())
A working(enq,x):Write(i = tail.getAndIncrement())
A working(enq,x):LINE 15:items[i=1].set(x)
A res(enq,x)
B working(enq,y):LINE 15:items[i=0].set(y)

B res(enq,y)
B inv(deq,y)
B res(deq,y)
A inv(deq,x)
A res(deq,x)

Question is *Does the fact that both line 14 and 15 are not linearization points this mean enq() is not linearizable?*
Definition of linearization does not really talk about linearization points. It talks about whether history of system, H, can be transformed to a history, S, such that S is legal to sequential specification of the object (in this case a queue), sequential (each invocation followed by response immediately), equivalent to H (each thread history order is maintained) and precedences are maintained between S and H (for every pair of methods in the history H, (A, B), where method A finishes before method B starts, such relation remains in S from H).
Also, from **Compositional Linearizability**, we know as long as all and only if all components of a system are linearizable, then, system is linearizable.
In Herlihy/Wing queue, both array *items* and integer *tail* are linearizable. These two components are the only *shared memory* components in enq method (and in the whole queue). Thus, **Herlihy/Wing queue's enq method is linearizable**.

1.15 Practice [Extra Credit 15 points]: Using a Linearizability Checker

A linearizability checker, as described by Jepsen Knossos, works as follows: Given a history of operations by a set of clients and some single-threaded model, attempts to show whether the history is linearizable with respect to that model. Use the Knossos linearizability checker to evaluate the histories provided in Problems 1.6–1.8 above. To receive full-credit, you should run the checker as described in Knossosas-a-library, and provide screenshots showing your inputs and the output of the tool. A README file has been attached to the assignment page that describes how to setup the library. Compare your answers to problems 1.6–1.8 and the checker's output; is the outcome of the checker correct? Remember, the author of the checker doesn't claim the checker to be correct and thus, you must verify the outcome by hand (which you should have already done in the Theory section). Don't submit any code as part of this assignment. Include only screenshots in your submission PDF document.

1. Exercise 1.6. Please find below the screenshot for exercise 1.6 in knossos. The result is that the history is linearizable , just as the manual analysis also reported:

2. Exercise 1.7. Please find below the screenshot for exercise 1.7 in knossos. The result is that the history is NOT linearizable , just as the manual analysis also reported:

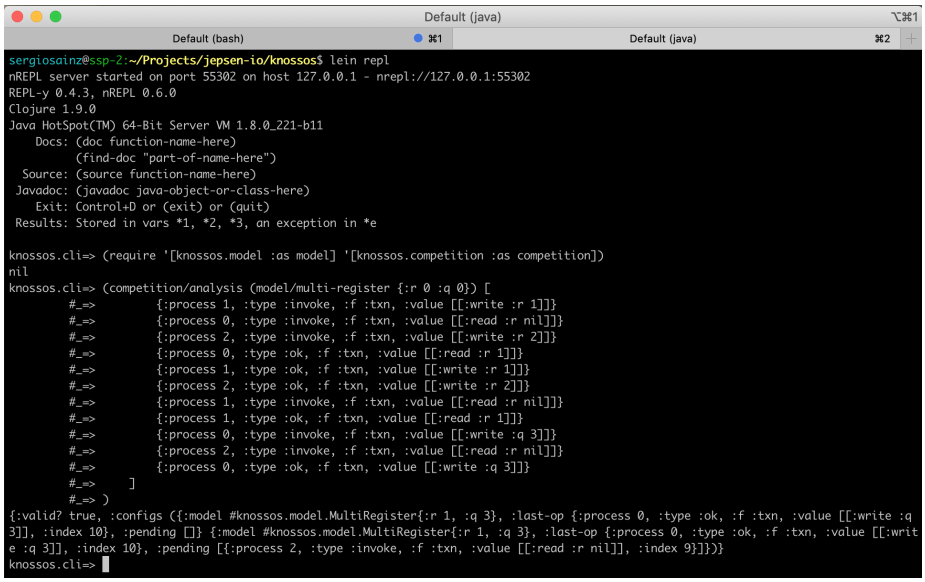

Figure 4: Exercise 1.6 in knossos

3. Exercise 1.8. Please find below the screenshot for exercise 1.8 in knossos. The result is that the history is NOT linearizable , just as the manual analysis also reported:

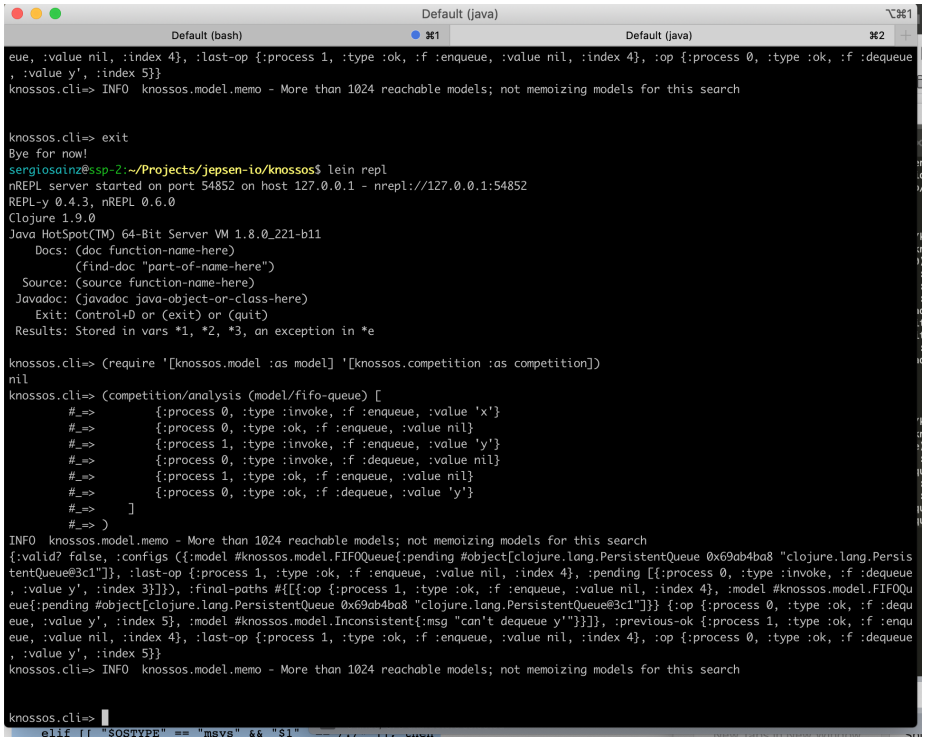

Figure 5: Exercise 1.7 in knossos

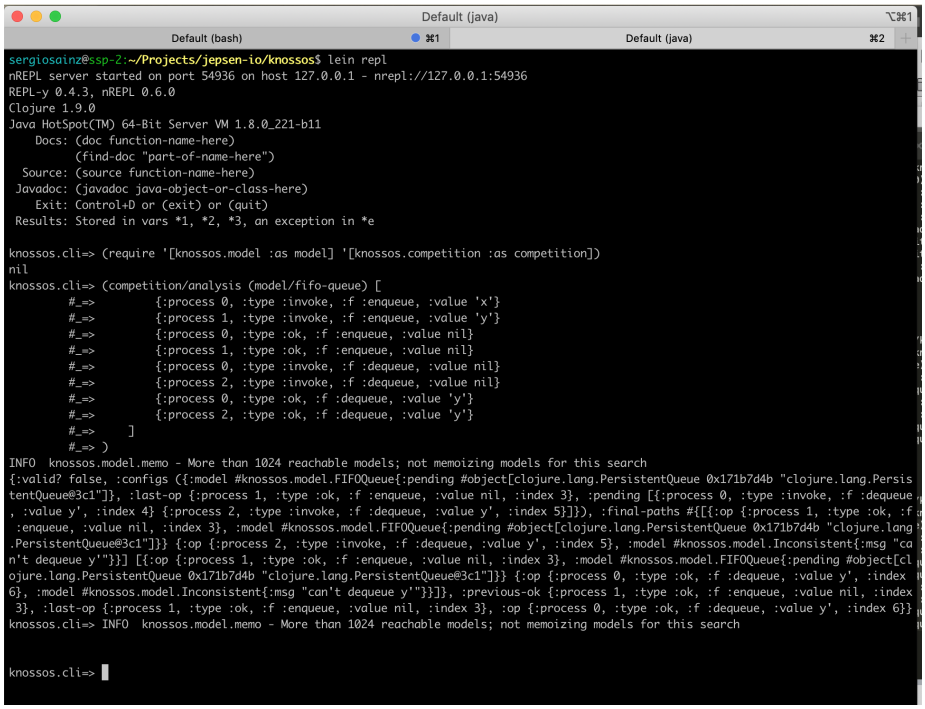

Figure 6: Exercise 1.8 in knossos