Exam in Concurrent Algorithms and Data Structures

Department of Information Technology
Uppsala University
2023–01-04
Lecturer: Parosh Aziz Abdulla
Location: Fyrislundsgatan 80, sal 1
Time: 8:00 - 13:00

No books or lecture notes allowed
The use of mobile phones, calculators, and computers is not allowed
Directions:

1. If you are asked to start your answer with one of the words, *yes* or *no*, please do so. Otherwise, you will not get any credit for the answer.

2. All explanations should be at most five lines. Lengthy explanations might yield a lower grade (and will not give extra points).

Good Luck!

Problem 1 We define the `MultiSet` abstract data type over the alphabet $\Sigma = \{a, b\}$ that is a generalization of sets over Σ. In contrast to sets which can store at most one occurrence of each symbol in the alphabet, a multiset can keep *multiple occurrences* of the alphabet's symbols. For instance, in our case, a multiset may contain three copies of a and two copies of b. Furthermore, we require that `MultiSet` may contain at most 50 elements at any time. `MultiSet` allows two operations, each taking one of the alphabet members, a or b, as input and returning a Boolean value indicating the operation's success. The operations are `add` that adds one more element of the given type to the multiset, and `rmv` that removes one element from the multiset.
Your task is to define the abstract data type `MultiSet`, i.e., the set S of states, the initial state s_{init}, the set M of method names, and the set R of rules. In the rules, you are only allowed to use the following operations, all of which are interpreted over the natural numbers: increment (by one), decrement (by one), equality predicate, and the larger-than predicate.

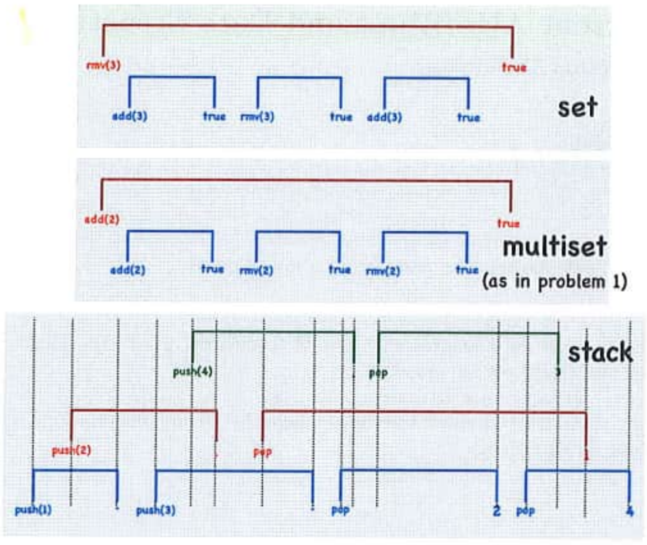
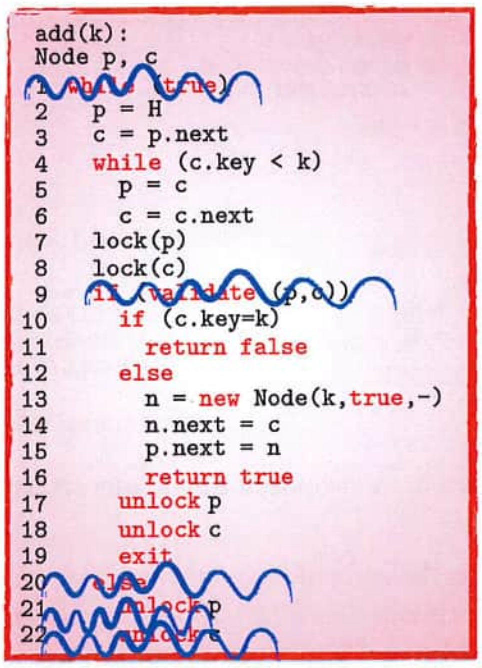


Figure 1: A set of histories.

Problem 2 For each of the concurrent histories shown in Fig. 1, answer whether the given history is linearizable for the given abstract data type. If *yes* redraw the history and put the linearization points in the correct places (no explanation is required in this case). If *no* explain in no more than five lines the reason. Assume that all three histories are initiated from an empty configuration, i.e., the empty set, the empty multiset, and the empty the stack , respectively.



```
add(k):
Node p, c
~~while (true)~~
  p = H
  c = p.next
  while (c.key < k)
    p = c
    c = c.next
  lock(p)
  lock(c)
  ~~if (validate(p,c))~~
    if (c.key=k)
      return false
    else
      n = new Node(k,true,-)
      n.next = c
      p.next = n
      return true
    unlock p
    unlock c
    exit
  ~~else~~
    ~~unlock p~~
    ~~unlock c~~
```

Figure 2: The add module optimistic list algorithm.

Problem 3 Recall the `Optimistic List` algorithm. Suppose that the threads only perform `add` operations, i.e., no thread ever performs a `delete` or a `ctn` operation. However, we are still interested in guaranteeing two properties, namely that (i) we cannot add multiple copies of the same element to the list, and that (ii) we allow adding an element if we have not previously added the element to the list.

Assume that we remove the `validate` procedure form the code of the `add` operation, so we obtain the code depicted in Fig. 2. Is the resulting algorithm linearizable wrt. the two guarantees mentioned above? First, answer *yes* or *no*. If *yes* motivate your answer in no more than five lines. If *no* give a concurrent history that shows non-linearizability.

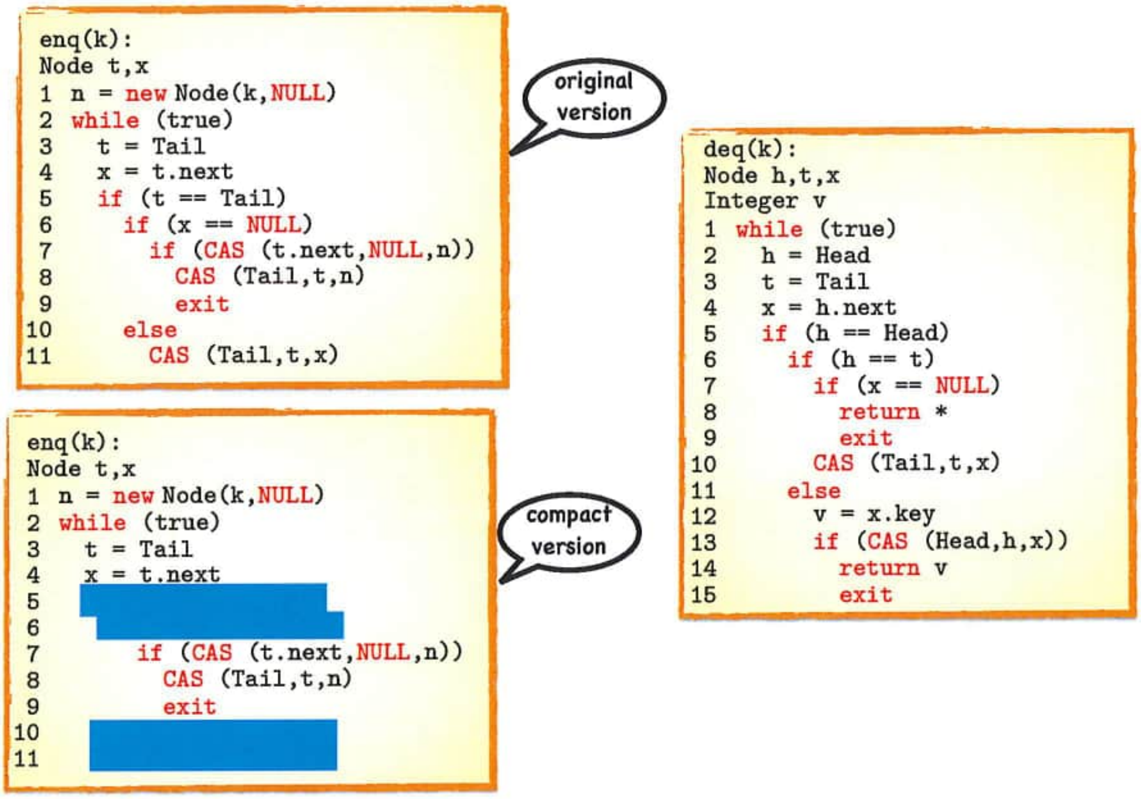
```
1: var q.back : int ← 0
2: var q.items : array of val
        ← {NULL, NULL, ...}
```

```
3: procedure enq(x : val)
4:     ⟨i ← INC(q.back)⟩          ▷ E1
5:     ⟨q.items[i] ← x⟩           ▷ E2 atomic
```

```
6: procedure deq() : val
7:     while true do
8:         ⟨range ← q.back − 1⟩                      ▷ D1 atomic
9:         for i = 0 to range do
10:            ⟨x ← SWAP(q.items[i], NULL)⟩          ▷ D2 atomic
11:            if x ≠ NULL then return x
```

Figure 3: A proposed queue algorithm.

Problem 4 Fig. 3 gives the code of a concurrent program that implements a queue. The queue is represented as a pre-allocated unbounded array, *q.items*, initially filled with *NULL*s, and a marker, `q.back`, pointing to the end of the used part of the array.
Enqueuing an element is done in two steps: the marker to the end of the array is incremented (E1), thereby reserving a slot for storing the element, and then the element is stored at the reserved slot (E2).
Dequeue is more complex: it reads the marker (D1) and then searches from the beginning of the array up to the marker to see if it contains a non-NULL element. It removes and returns the first such element it finds (D2) (Line 11 returns the result and *exits* the search.) If no element is found, Dequeue starts afresh. Each of the three statements surrounded by brackets and annotated by E_2, D_1, and D_2 is assumed to execute atomically. However E_1 is not atomic, i.e., we can do the incrementation, wait, and later assign to i.
Is the algorithm linearizable? If *yes* motivate your answer in no more than five lines. If *no* give a concurrent history, with no more than three threads, that shows non-linearizability.


original version
compact version

```
enq(k):
Node t,x
n = new Node(k,NULL)
while (true)
  t = Tail
  x = t.next
  if (t == Tail)
    if (x == NULL)
      if (CAS (t.next,NULL,n))
        CAS (Tail,t,n)
        exit
    else
      CAS (Tail,t,x)
```

```
enq(k):
Node t,x
n = new Node(k,NULL)
while (true)
  t = Tail
  x = t.next
  [redacted]
      if (CAS (t.next,NULL,n))
        CAS (Tail,t,n)
        exit
  [redacted]
```

```
deq(k):
Node h,t,x
Integer v
while (true)
  h = Head
  t = Tail
  x = h.next
  if (h == Head)
    if (h == t)
      if (x == NULL)
        return *
        exit
      CAS (Tail,t,x)
    else
      v = x.key
      if (CAS (Head,h,x))
        return v
        exit
```

Figure 4: The new version of the Michel-Scott algorithm.

Problem 5 Conisder the new version of the Michael-Scott algorithm depicted in Fig. 4. In the new algorithm we keep the dequeue part, but replace the enqueue part by a more compact code where we have removed some lines of code.
Is the resulting algorithm linearizable? First, answer *yes* or *no*. If *yes*, give the linearization policy, and motivate your answer in no more than five lines. If *no* give a concurrent history that shows non-linearizability.

Algorithm 1: The code for an A-process.

```
update($a, 0, 1$)
$c := c + 1$
if $c = 1$ then RMW($b, 0, 1$)
$a := 0$
R
update($a, 0, 1$)
$c := c - 1$
if $c = 0$ then $b := 0$
$a := 0$
```

Algorithm 2: The code for a B-process.

```
RMW($b, 0, 1$)
R
$b := 0$
```

Algorithm 3: The code for the update operation.

```
$e :=$ false
repeat
    $z := x$
    if $z = k$ then
        $x := \ell$;    $e :=$ true
until $e =$ true
```

Problem 6 Consider a concurrent system in which we have a set of processes of two types A and B that share a common resource R. The protocol is supposed to satisfy a safety property which we refer to as `MutEx`. The `MutEx` property has the following interpretation:

- Any number of A-processes can access R at any give point of time.
- A B-process can access R only if no other process, whether of type A or type B, accesses R at the same time.

We depict the code for A-processes and B-processes in Algorithm 1 resp. Algorithm 2. The instruction RMW is a *Read-Modify-Write*. More precisely, RMW(x, k, ℓ) checks whether the value of the variable x is equal to k. If *yes*, it atomically *changes* the value of x to ℓ. If *no*, it waits until the value of x has become equal to k, and then performs the operation.
Does the protocol satisfy the `MutEx` property? If *yes*, motivate the reason in no more than five lines. If *no*, give a run of the program that violates the property.