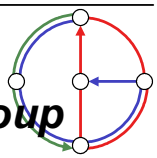
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group



HS 2009 Prof. Dr. Roger Wattenhofer, Thomas Locher, Remo Meier, Benjamin Sigg

Distributed Systems
Theory exercise 6

Assigned: December 11, 2009
Discussion: none

1 ALock2

Have a look at the source code below. It is a modified version of the ALock (slides 8/42 ff).

```
public class ALock2 implements Lock {
  int[] flags = {true,true,false,...,false};
  AtomicInteger next = new AtomicInteger(0);
  ThreadLocal<Integer> mySlot;

  public void lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {}
    flags[mySlot % n] = false;
  }

  public void unlock() {
    flags[(mySlot+2) % n] = true;
  }
}
```

a) What was the intention of the author of "ALock2"?

b) Will ALock2 work properly? Why (not)?

c) Give an idea how to repair ALock2.

Hint: don't bother about performance.

Solution

a) The author wants that two processes can acquire the lock simultaneously.

b) The lock is seriously flawed. An example shows how the lock will fail: Assume there are n processes, all processes try to acquire the lock. The first two processes (p_1, p_2) get the lock, the others have to wait. Process p_1 keeps the lock a very long time, while p_2 releases the lock almost immediately. Afterwards every second process (p_4, p_6, ...) acquires and releases the lock. One half of all process are waiting on the lock (p_3, p_5, ...), the others continues to work (p_4, p_6, ...). If the working process now start to acquire the lock again, then they wait in slots that are already in use.

c) A solution would be to increase the size of the array to $2 * n$. With this size there are simply not enough processes to wrap around too early.

Or we mark any occupied slot. If a process tries to obtain an occupied slot, it just tries again with the next slot.

Unfortunately one process could still block all the other processes, and FIFO (first in, first out) is still not guaranteed. In a second step one could make the `unlock` method more intelligent: instead of jumping two slots, the next slot is first checked. If there is a process waiting there, then this process gets the lock. If the process there already acquired the lock, the next but one slot gets the lock. We can use a lock to protect the `unlock` method against race conditions.

```
// some simple lock, e.g. a spin lock
Lock somelock = ...

void unlock(){
  somelock.lock();

  if( flags[ (mySlot+1) % n] ){
    // the next slot already has acquired the lock
    flags[(mySlot+2) % n] = true;
  }
  else{
    // the next slot did not already acquire the lock
    flags[(mySlot+1) % n] = true;
  }

  somelock.unlock();
}
```

2 MCS Queue Lock

See slides 8/56 ff.

a) A developer suggests to add an `abort` flag to each node: if a process no longer wants to wait it sets this `abort` flag to `true`. If a process unlocks the lock, it may see the `abort` flag of the next node, jump over the aborted node, and check the successor's successor node. Modify the basic algorithm to support aborts.

Optional: sketch a proof for your answer.

Hint: Be aware of race-conditions!

b) Assuming many processes may abort concurrently, does your answer from a) still work? Explain why. If it does not work: modify your algorithm to allow concurrent aborts.

Optional: sketch a proof for your answer.

c) Instead of a `locked` and an `aborted` flag one could use an integer, and modify the integer with the CAS operation. What do you think about this idea? How is the algorithm affected? How is performance affected?

d) The CLH lock (slide 8/49) is basically the same as an MCS lock. Conceptually the only difference is, that a process spins on the `locked` field of the predecessor node, not on its own node. What could be an advantage of CLH over MCS and what could be a disadvantage?

Solution

a) There is more than one solution, but we can solve this problem without using RMW registers or other locks. It is important to set and read the flags in the right order: The `unlock` method first sets `locked`, then reads `aborted`. The `abort` method on the other hand first sets `aborted`, then reads `locked`. This way if `unlock` and `abort` run in parallel, one of them must already have written its flag before the other can read it. In the worst case `unlock` is called twice for some process, but that is not a problem. Unlocking an already unlocked lock results in no action.

```
public void unlock(){
  if( ... missing successor ... )
    ... wait for missing successor

  qnode.next.locked = false;
  if( qnode.next.aborted ){
    if( ... qnode.next misses successor ... ){
      if( ... really no successor ... )
        return;
    }
    else{
      ... wait for missing successor ...
    }
    qnode.next.next.locked = false;
  }
}

public void abort(){
  qnode.aborted = true;
  if( !qnode.locked ){
    unlock();
  }
}
```

b) The solution of a) does not yet work for concurrent aborts. Making the `unlock` method recursive will help.

```
public void unlock(){
  unlock( qnode );
}
private void unlock( QNode qnode ){
  // as before...
  if( ... missing successor ... )
    ... wait for missing successor

  qnode.next.locked = false;
  if( qnode.next.aborted ){

    // wait for successor of qnode.next
    if( ... ){ ... } else{ ... }

    unlock( qnode.next );
  }
}
```

c) There are four combinations of values the `locked` and `aborted` flag can have. We can easily encode these combinations in an integer. We would not need too worry about the order in which we read and write to the flags, as we could do this atomically. So the algorithm would get easier. We could also ensure that `unlock` is called only once. Depending on the benchmark this could increase the performance. On the other hand, a `CAS` operation is quite expensive and could decrease performance.

d)
- \- There could be problems with caches: spinning on a value that "belongs" to another process can introduce additional load on the bus, and thus slow down the entire application.
- \+ The implementation is much easier: when releasing the lock one has only to set its own `locked` flag to `false`.
- \+ Also aborting is easier: a blocked process could read the state of its predecessor. If the predecessor is aborted, then the successor can just remove the node from the queue, and continue reading values from its predecessor's predecessor.

3 Linked-Lists

a) Write a proof for: a linked-list using fine-grained locking (as described on slide 8/76 ff) does not deadlock.

b) Lazy synchronization: can the `contains` method return a false answer when searching for x because processes concurrently try to add and remove x? Make a reasonable assumption how `add` works.

c) Optimistic synchronization: describe a scenario where a process is forever attempting to delete a node.

d) CAS: think about how to implement the `add` method. Write the interesting parts as pseudo-code.

Solution

a) We assign a number p to each node, according to their location in the list. The first node gets $p = 1.0$, the second node gets $p = 2.0$, etc... A new node gets the mean of its successor and predecessor, e.g. a new node between the first and second node would get the number $p = (1.0 + 2.0)/2 = 1.5$.

We observe: the first lock a process acquires never causes problems. If the process cannot acquire its first lock, then it cannot block any other process.

We further observer: if a process wants to acquire the lock of node n_a with p_a, then the process has not yet acquired the lock of a node n_b width $p_b > p_a$.

Assume a process locked n_a and wants to lock n_b, then $p_a < p_b$. If another process locked n_b and wants to lock n_a, then $p_a > p_b$. This is a contradiction, hence a deadlock never occurs.

This argument can be repeated for any number and combination of locks and processes.

b) The `add` method will lock the future predecessor and successor nodes as well. If `add` and `remove` work with the same item, they will never modify the list at the same time.

As a result it is always clear whether an item is in the list or not, and `contains` will never fail.

c) In order to delete a node, the `validate` method must return `true`. If we prevent `validate` from being `true`, the process will spin forever. For example we could have some process(es) which insert nodes between `pred` and `curr`. Since `validate` traverses the whole list, there is enough time to do the damage.

d) A solution could be to remove marked nodes on the fly when encountered. The pseudo code below is such an example.

```
// Node *node: "node" is a pointer to a "Node"-object.

Node* pointer( Node *node ){
  return node & ~1;
}

boolean marked( Node *node ){
  return (node & 1) == 1;
}


boolean add( Item item ){
  int key = item.hashCode();

  retry: while( true ){
```

```
    // traverse to predecessor

    Node *pred = head;
    Node *curr = pointer( pred->next );

    while( curr->key < key ){
      Node *succ = curr->next;
      // If curr->next (== succ) contains the marked-flag,
      // curr is marked as being removed.
      // We replace it on the fly
      if( marked( succ )){
        // try to replace pred.next. If pred is marked,
        //then this CAS fails.
        if( pred->next.CAS( curr | FALSE, succ | FALSE )){
          curr = succ;
          succ = pointer( curr )->next;
        }
        else{
          // on failure: just start over
          continue retry;
        }
      }
      pred = curr;
      curr = succ;
    }

    // add item

    if( curr->key == key ){
      return false;
    }
    else{
      Node *node = new Node( item );
      node->next = curr | FALSE;
      // try to insert the new node. If CAS fails,
      // the loop starts again.
      if( pred->next.CAS( curr | FALSE, node | FALSE )){
        return true;
      }
    }
  }
}
```