Concurrent Algorithms and Data Structures (Sheet #2)

Marius Gavrilescu

1. (a) At the very beginning there might be few or a single task in the queue. The first thread selects this single task, the other threads then see the queue is empty and terminate, and then the first thread proceeds alone.

   (b) If the token made a complete circuit then we know that each thread at some point had the token (because the token got back to the controller), and at that point the thread called canTerm (so there was no work to do) and at the time of calling canTerm the thread had not called `noTerm` since the last time it had the token (because the thread did not become the new controller). Therefore this thread must be waiting in the while loop for a new task.

   We therefore know that every thread is waiting in the while loop for a new task, and so there is no way for any progress to be made. Therefore we can terminate.

   (c) The token can be just an AtomicInteger, meaning the ID of the thread currently holding the token. We can also have an AtomicBoolean in the Termination class which the controller sets to true when we can terminate, and canTerm reads.

```
class Termination(p: Int) {
  val terminate = new AtomicBoolean(false)
  val controller = new AtomicInteger(0)
  val token = new AtomicInteger(0)
  val calledNoTerm = new AtomicIntegerArray(p)

  var firstRound = false;
  private def next(me: Int) = (me+1)%p

  def canTerm(me: Int): Boolean = {
    /* did the controller tell us to terminate? */
    if(terminate) return true
    /* do we have the token? */
    if(token.get != me) return false
    /* did we call noTerm since we last had the token? */
    if(calledNoTerm.get(me) > 0) {
      calledNoTerm.set(me, 0)
      token.set(next(me))
      return false
    }
    /* are we the controller? */
    if(controller.get != me) {
      token.set(next(me))
      return false
    }
    /* is this the very first round?
```

```
        i.e. is this thread 0 calling canTerm for the first time? */
      if(!firstRound)
        terminate.set(true)
      firstRound = true;
      return false;
    }

    def noTerm(me: Int) = {
      calledNoTerm.set(me, 1)
    }
  }
```

3. (a)
```
object SavingsAccount {
  val lock = Lock()
  val condition = lock.newCondition

  var balance = 0

  def deposit(amount: Int) = lock.mutex {
    balance += amount
    condition.signalAll
  }

  def withdraw(amount: Int) = lock.mutex {
    while(balance < amount) condition.await
    balance -= amount
  }
}
```

   (b)
```
object BetterSavingsAccount {
  val lock = Lock()
  val condition = lock.newCondition

  var balance = 0
  var pendingPrefWithdrawals = 0

  def deposit(amount: Int) = lock.mutex {
    balance += amount
    condition.signalAll
  }

  def withdraw(amount: Int) = lock.mutex {
    while(balance < amount && !pendingPrefWithdrawals) condition.await
    balance -= amount
  }

  def prefWithdraw(amount: Int) = lock.mutex {
    pendingPrefWithdrawals += 1
    while(balance < amount) condition.await
    balance -= amount
    pendingPrefWithdrawals -= 1
```

```
    }
  }
```

(c) `transfer` locks the lock, then calls `withdraw` which locks the lock again. If the lock is not reentrant this would cause a deadlock.

(d) If `transfer` is called with insufficient funds, then `withdraw` will release the lock and wait, but the lock will still be held by `transfer`. So no thread will be able to deposit to the account.

5.
```
class Rooms(n: Int) {
  val arrived = new Array[Int](n)
  val conditions = new Array[Condition](n)
  var currentRoom = -1

  val lock = new BakeryLock()
  for(i <- 0 until n) conditions(i) <- lock.newCondition

  def enter(r: Int) = lock.mutex {
    arrived(r) += 1
    if(currentRoom >= 0) /* if at least one room is in use */
      do {
        conditions(r).await()
      } while (currentRoom >= 0 && currentRoom != r);
    currentRoom = r
  }

  def exit(r: Int) = lock.mutex {
    assert (currentRoom == r)
    arrived(r) -= 1
    if(!arrived(r)) { /* r is now empty */
      var newRoom = r+1
      /* find next room with threads waiting */
      while(arrived(newRoom%n) == 0 && newRoom < r+n) newRoom += 1
      if(arrived(newRoom%n)) {
        currentRoom = newRoom%n
        conditions(currentRoom).signalAll
      } else { /* nobody is waiting */
        currentRoom = -1
      }
  }
}
```

Here we keep the current room that contains at least one thread (or -1 if all rooms are empty) and an array of threads that arrived at the entrance of room i (that is, they called `enter` but not `exit`), and one condition for each room.

When a thread arrives, we note its arrival. If all rooms are empty, we let the thread in. Otherwise, we make the thread wait. When a waiting thread is woken up, it checks whether it can enter (that is, all rooms are empty or it is trying to enter the same room that is currently in use), and enters if yes or waits again if not.

When a thread leaves, we note that it left. If the room is now empty, then we look for the next room that has threads waiting for it, and we wake up all of those threads. If nobody is

waiting, then we note that all rooms are empty.

This algorithm satisfies mutual exclusion, because a thread can only enter if all rooms are empty or it is trying to enter the roomcurrently in use.

To ensure starvation freedom we make all threads (except the thread that enters an empty building) wait at least once (even if they would be allowed to enter!), and when a room becomes empty we look for the next room with threads waiting and let those in.

Because we only let in the threads waiting right now for room i (and not any threads that arrive for room i later), we can see that `currentRoom` will always increase (wrapping around at n). Therefore if a thread is waiting, `currentRoom` will eventually be this thread's room and this thread will be woken up, at which point it will enter the room. So every thread that is waiting will eventually enter.

6. The implementation is safe, because if a read does not overlap a write operation then at the time of the read the last write operation has finished, so the message passed through every node, and every node set its local variable to the value in the message. So any thread reading now would see the last written value. If a read overlaps a write, then the read will return the value written by that write, or a value written by an earlier write, or the initial value (all of these are allowed values).

   The implementation is also atomic, because it is wait-free (neither reading nor writing involve waiting) and if a read overlaps with N writes $w_1, w_2, w_3, \ldots, w_n$, then by assumption no two of these writes overlap so we read either the value before w_1 (if the message from w_1 did not reach us), or the value written by w_1 (if the message from w_1 reached us but not the message from w_2), or the value written by w_2, ... or the value written by w_n. If we do two non-overlapping reads one after the other, then since messages are received in FIFO order the second read will return either the same value or a value written later. So the register is linearizable.

   With overlapping writes the register is no longer safe. Suppose we have 3 nodes, and thread 1 starts to write 1 and thread 3 starts to write 3 before the first write finishes. We have the following events:

   Thread 1 writes 1
   Thread 2 receives "Thread 1 writes 1"
   Thread 3 receives "Thread 1 writes 1"
   Thread 3 writes 3
   Thread 1 receives "Thread 3 writes 3"
   Thread 2 receives "Thread 3 writes 3"
   Thread 3 receives "Thread 3 writes 3"
   Thread 1 receives "Thread 1 writes 1"

   Now both writes are finished, and the local values are: 1 in thread 1, 3 in thread 2, 3 in thread 3. So a read made by thread 1 followed by a read made by thread 2 return different values, even though there was no write in between (so it is not the case that a read that does not overlap with a write will return the last value written, because reads by different threads return different values).

7.
```
def read : Int = {
  val arr = new Array[Int](N);
  for(i <- 0 until N) arr(i) = b(i).read
  val left = booleanArrayToInt(arr)
  for(i <- 0 until N) arr(i) = b(N + i).read
  val middle = booleanArrayToInt(arr)
  for(i <- 0 until N) arr(i) = b(N * 2 + i).read
  val right = booleanArrayToInt(arr)

  /* write has not begun OR has already written left and middle. */
  if(left == middle)
    return left
  /* write has begun but has not yet started to write middle. */
  else if(middle == right)
    return right;
  /* write has started to write middle. Left must be clean. */
  else {
    /* note register is write-once, so reading left again will not
       interfere with another write. */
    for(i <- 0 until N) arr(i) = b(i).read
    return booleanArrayToInt(arr)
  }
}
```

10. All we need to do is to take 32 n-thread binary consensus objects, then use the first one to decide the first bit of the 32-bit int, the second one to decide the second bit, and so on. On the ith step, each thread can submit the last bit of its thread ID to the ith consensus object, and get back a bit. At the end each thread will know 32 bits in order, and all threads will have the same bits. Therefore all threads share a 32-bit integer.