Department of Information and Computer Science
Utrecht University

INFOB3CC: Threads (solutions)

Trevor L. McDonell

November 21, 2022

Introduction

This tutorial set is intended to be done after the Threads 3 lecture.
Some further reading on the terms used in this tutorial set:

- `https://en.wikipedia.org/wiki/Test-and-set`
- `https://en.wikipedia.org/wiki/Compare-and-swap`
- `https://en.wikipedia.org/wiki/Non-blocking_algorithm`

Questions

1. Properties of non-blocking algorithms.
   (a) What does it mean for an operation to be atomic (linearisable)?

   > Operations are atomic if their outcome appears to take place instantaneously, regardless of other operations executing at the same time.

   (b) What does it mean for a program to be lock-free?

   > Lock-freedom allows individual threads to starve, but guarantees that at least one thread will make progress.

   (c) What does it mean for a program to be wait-free?

   > Wait freedom guarantees that all threads of the system will make progress and complete in a bounded number of steps.

   (d) What is meant by the assumption of fairness?

   > Fairness is the likelihood that at every step of the program execution, each of the threads have a chance to advance. In a completely fair (100%) system all threads advance their work in equal proportions; in a completely unfair (0%) system, only one thread is advancing and all other threads never make progress.

2. With a single `IORef` you can create a multi-threaded lock.
   (a) Which operation(s) in Haskell can you use to implement this operation?
   (b) Give the code for this compare-and-swap-lock (lock and unlock method)

Example implementation:

```
import Data.Atomics (readForCAS, casIORef, peekTicket)

spinLock :: IORef Int -> IO ()
spinLock lock = do
  ticket <- readForCAS ref
  go ticket
  where
    go :: Ticket Int -> IO ()
    go ticket
      | peekTicket ticket == 0 = do
        -- if it was unlocked, try locking
        (success, ticket') <- casIORef ref ticket 1
        if success
          then return ()  -- success, we took the lock
          else go ticket' -- try again
      | otherwise = spinLock lock  -- was already locked, spin

spinUnlock :: IORef Int -> IO ()
spinUnlock lock = atomicWriteIORef lock 0
```

(c) Which of these properties does a CAS-lock have? Explain why:

- Mutual exclusion
- Deadlock freedom
- No starvation

A CAS lock provides mutual exclusion and deadlock-freedom. However, it suffers from starvation; a thread can have endless bad luck as different threads might always be scheduled before it whenever the lock is released.

3. The CAS lock uses a repetition of the compare-and-swap instruction (in Haskell we use `casIORef`).

(a) If we do *not* assume fairness of the scheduler, is a CAS lock starvation-free?

If the scheduler cannot be assumed to provide fairness, there is very little one can assume. In particular, a thread may not be scheduled at all in some circumstances. (In practice, of course, real schedulers don't do such stupid things as "never schedule a thread", and even if they are not strictly speaking "fair", the issues are usually more subtle.)

(b) Is the lock starvation-free if we assume fairness of the scheduler?

No, a thread can be passed by arbitrarily often. Suppose that threads 0 and 1 alternate in the critical section: each time one thread leaves the critical section, the other one wants in and asks for and gets the lock. The value of the lock often randomly changes between 0 and 1. In the case of unfortunate (but possible!) scheduling, it is possible that every time thread 2 tries the CAS, the lock is taken by some other thread.

(c) Is the lock deadlock-free if the scheduler is fair?

Yes. Suppose that one or more threads attempt to acquire the lock, which is currently unoccupied. The last thread to have exited the critical section returns the lock back to the unlocked state, and due to fairness all the waiting threads then have a chance to perform the CAS operation.

4. Consider the different (attempted) implementations of software-based mutual exclusion we discussed in the *Threads 1* lecture, culminating in Peterson's algorithm.
   (a) What are the requirements that a lock implementation must fulfil?

A lock should comply with

- Mutual exclusion: at most 1 thread is in the critical section at any time
- No deadlock: if threads want to get the lock, and it is free, they can acquire it
- No starvation: every thread that asks for the lock, will eventually get it

   (b) Which of these requires are met/are not met for attempt 1 (the simple turn-based lock)?

Attempt 1 satisfies mutual exclusion because processes always check whether it is their turn before entering the critical section. It does not comply with deadlock freedom, because if a thread fails anywhere in the program, the other thread is permanently blocked. This also implies starvation, however in the working scenario a thread can not be overtaken arbitrarily often; processes must alternate access to the critical section.

5. Give a simple example program where each lock individually complies with deadlock freedom, but where it can still happen that the program as a whole deadlocks.

Some possibilites:

- A program with lock priority inversion (taking locks in inconsistent orders)
- Example: dining philosophers, where each philosopher first grabs the left fork, and then the right fork. Deadlock can happen at the level of the philosopher, even though taking each individual fork is deadlock free.

6. Give an example of a problem/computation which is concurrent but not (necessarily) parallel. Explain why.

There are many examples in this category, e.g.:

(a) A web server where each client is handled by a separate thread. Concurrency is used to simplify the program structure so that each client is a logically sequential piece of code (no explicit switching between user requests in the main loop).

(b) A GUI, where different threads are delegated to process different tasks, such as waiting on different user inputs, rendering the display, etc.

7. What is the progress guarantee of the following piece of C code?[1]

```
typedef struct {
  int count;
} obj;

void increment_reference_count(obj* this)
{
  atomic_increment(this->count);
}

void decrement_reference_count(obj* this)
{
  if ( 0 == atomic_decrement(this->count) )
    delete this;
}
```

> The code is wait-free: different threads can call the atomic increment/decrement functions concurrently and they will all complete in a finite number of steps (implementation handled by the hardware).

8. What is the progress guarantee of the following piece of C code?

```
void stack_push(stack* s, node* n)
{
  node *head;
  do
  {
    head    = s->head;
    n->next = head;
  }
  while ( ! atomic_cas(s->head, head, n) );
}
```

> The code is lock free: one thread will successfully complete the CAS each time (and therefore complete in a finite number of steps), but it is possible that a thread is infinitely unlucky and never succeed to push its element onto the stack.

9. The CAS lock does not guarantee absence of starvation. To overcome this limitation, we consider an attempt on creating a lock consisting of a boolean array `flags`, denoting which threads want to acquire the lock, and a variable `lock`, denoting the index of the thread which currently has the lock or -1 if no thread has the lock. Each thread has a unique positive thread index $0 \leq$ `thread_index` $<$ `thread_count`.

   To acquire the lock, we write to `flags` and then try to take the lock using a compare-and-swap instruction:

[1] For those not familiar with C syntax: a struct such as `obj` is a chunk of memory large enough to hold the fields of the struct next to each other. The syntax `struct->field` accesses the value of `field` inside the referenced (by pointer) `struct`. An object in an object-oriented language is thus just a struct, and member functions are just regular functions which take a hidden first argument (usually called `this`) which is a pointer (reference) to the struct containing the member variables of that object. Thus you can read this example as how you could implement reference counting for objects in an OOP language.

```
flags[thread_index] = true
while (true) {
  old = atomic_cas(lock, -1, thread_index)
  if (old == -1 || old == thread_index) { return }
}
```

To release the lock, we look for a next thread which is waiting to acquire the lock, and assign the lock to that thread, or write -1 if no other thread needs the lock:

```
flags[thread_index] = false
for (i = thread_index+1; i < thread_index+thread_count; i++) {
  if (flags[i mod thread_count]) {
    lock = i mod thread_count
    return
  }
}
lock = -1
```

Note that `mod` denotes modulo, the remainder of integer division.

(a) Does this lock guarantee mutual exclusion? Motivate your response.

> Yes. The procedure to acquire the lock ends when the CAS write succeeded (`old = 1`) or when the old value already was `thread_index`. In both cases, `lock` will now be `thread_index`. This thread won't change `lock` until it releases the lock. Any other threads trying to acquire the lock cannot exit the lock, as the CAS fails and `old` isn't equal to their thread index.

(b) Recall that deadlock freedom of a lock means that if threads want to get the lock, and it is free, they can acquire it. Does this attempt of creating a lock guarantee deadlock freedom? Explain why.

> Yes. If variable lock is -1, then a thread can take the lock. If the variable lock is not -1, and no thread has entered the critical section, then one of the locks was assigned the index of a thread who requested the lock. That thread can thus proceed in the CAS-loop of the acquire procedure.

(c) Does this attempt of creating a lock guarantee the absence of starvation? Explain why.

> Yes. Threads request the lock with the `flags` array. When a thread releases the lock, it finds the next thread (in a cyclic order) who requested the lock. Any thread requesting the lock will thus eventually get the lock.