Department of Information and Computer Science
Utrecht University

INFOB3CC: Threads

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)?
   (b) What does it mean for a program to be lock-free?
   (c) What does it mean for a program to be wait-free?
   (d) What is meant by the assumption of fairness?
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)
   (c) Which of these properties does a CAS-lock have? Explain why:
      - Mutual exclusion
      - Deadlock freedom
      - No starvation
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?
   (b) Is the lock starvation-free if we assume fairness of the scheduler?
   (c) Is the lock deadlock-free if the scheduler is fair?
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?
   (b) Which of these requires are met/are not met for attempt 1 (the simple turn-based lock)?
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.

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

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;
}
```

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) );
}
```

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:

```
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:

[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] = 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.

(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.

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