

COMP 321: Design of Programming Language, Fall 2026

Homework 4: Concurrency

1 C Syntax

Finish Tasks 5 and 6 from Homework 3 if you didn't do them last week. Hand in `elab.sml` and `reverse.impc` to a `hw04` folder.

2 Concurrency

Before doing this part of the homework, you might want to look at the lecture notes for this week for the solution to a couple of relevant problems from Wednesday's lab (the "give an operational semantics for fork" problem and the "definition of test-and-set" problem). Some of the problem below were on Wednesday's lab, but most people didn't get to them; if you did, feel free to just write up your solution from class. You should hand in this problem by uploading a file `hw04.pdf` to your handin folder in a `hw04` folder. You can either type your solution, or hand-write it and then take pictures or scan it in.

2.1 Test-and-set and Locks

In class on Monday (see the lecture notes), we gave a specification of a test-and-set operation that can be used to implement locks. In this task, your job is to give an operational semantics rule for adding test-and-set as a new built-in operation to the language. The new built-in should have the same effect on memory as the implementation in the lecture notes, but it should perform both memory updates in one step of evaluation, so no other threads can be interleaved between the two. This corresponds to how test-and-set or other concurrency primitives are implemented as a single instruction in hardware.

Suppose we have a statement `TestAndSet( $e_1, e_2$ )` where e_1 evaluates to the address of the flag and e_2 evaluates to the address for the return value.

Task 1 (15%). Define step rules for¹

$$\frac{}{\mu \mid \text{testAndSet}(e_1, e_2) \mapsto \text{testAndSet-instr}}$$

Task 2 (15%). Review the section of the lecture notes on locks, which writes up what we covered in class on Monday. Then, finish implementing a lock by implementing `acquire`. When the lock is 0, `acquire` should set the lock to 1 and then return. When the lock is 1, `acquire` should wait until the lock is released (this is called a *spin lock*). Use test-and-set to make sure that if two threads attempt to acquire the lock, only one at a time succeeds.

¹Unlike in lab, e_1 and e_2 are arbitrary expressions here.

```

acquire(var lock) {
    ...
}

```

Task 3 (10%). Using the operational semantics from the lab solutions in the lecture notes, and your operational semantics for test-and-set from above, write out a trace of the `main()` function below in which the first thread (the plus 1 thread) acquires the lock, and then the second thread (the plus 2 thread) attempts to acquire the lock but is stopped from doing at least once, and then does acquire the lock after the first thread releases it.

```

release(var lock) {
    get(lock) := 0;
}

main() {
    {var lock;
    lock := 0;
    {var x;
    x := 0;
    fork { acquire(lock); x := (get(x) + 1); release(lock); };
    fork { acquire(lock); x := (get(x) + 2); release(lock); };
    }
    }
}

```

2.2 Consensus

The *consensus problem* is a fundamental building block for distributed systems. In the consensus problem, multiple threads are trying to agree on a number (e.g., synchronizing clocks across different machines). However, the threads are not allowed to wait indefinitely long to proceed — for example, if one thread crashed, the others should be able to proceed with their work, having agreed amongst themselves. This rules out a simple implementation like “thread 1 writes the number to shared memory; thread 2 waits until thread 1 has done so and then reads it”.

Task 4 (20%). Write an IMP program that solves the consensus problem for 2 threads. That is, each thread should start off “thinking of” a fixed hard-coded number, and both threads should “return” (in the sense of setting a return variable) the same number. For concreteness, suppose thread 1’s number is 31 and thread 2’s number is 32. Then either both threads return 31 or both return 32.

Hint, use the following algorithm: allocate some shared memory where each thread can write its number, as well as some shared memory for determining who went first. Whichever thread goes first returns its own number, while whichever thread goes second returns the other thread that went first’s number.

In the code below, implement `threadOne` and `threadTwo` and show how to use them so that `return1` and `return2` are always the same number (either 31 or 32 depending on which thread is scheduled first). You can change what memory the `main()` function allocates if you want.

```

threadOne(...){
    ...
}

threadTwo(...){
    ...
}

main() {
    {var return1;
      return1 := 0;
      {var return2;
        return2 := 0;
        {var amIFirst;
          amIFirst := ...;
          {var threadOneAnswer;
            threadOneAnswer := ...;
            {var threadTwoAnswer;
              threadTwoAnswer := ...;
              fork { threadOne(...); };
              fork { threadTwo(...); };
            }}}}
    }
}

```