

COMP 321 Lecture 08: Concurrency Lab

Dan Licata

Fall 2026

In this lab, you will adapt the operational semantics we have written for sequential programs to handle concurrent threads. The meta-operations of *update*, *lookup*, and substitution, and the rules for stepping expressions will stay unchanged from before. We will start from the rules for stepping statements in Figure 1.

1 Thread pool

A thread pool is a list of threads, where each thread is a statement. We write a thread pool as $s_1, \dots, s_n$ or write $\vec{s}$ for a whole list.

Here is a first attempt at a rule for stepping threads:

$$\frac{\mu \mid s_i \mapsto \mu' \mid s'_i}{\mu \mid s_1 \parallel \dots \parallel s_i \parallel \dots \parallel s_n \mapsto \mu \mid s_1 \parallel \dots \parallel s'_i \parallel \dots \parallel s_n} \text{ step-threads-1}$$

Since i can index any thread s_1 through s_n , this means that the thread pool steps by choosing any thread s_i and stepping that thread. This is *non-deterministic* in the sense that a thread pool can have many different next steps. A list of steps with each thread pool stepping to the next $(\mu_1 \mid \vec{s}_1) \mapsto (\mu_2 \mid \vec{s}_2) \mapsto (\mu_3 \mid \vec{s}_3) \dots$ is called a *trace*.

We will write a_0, a_1 , etc. for memory addresses (which could concretely just be Num(0), Num(1), etc.)

Task 1. Consider the thread pool

```
[(a0, 0)] | (a0 := get(a0) + 1;) || (a0 := get(a0) + 2;)
```

- Give a trace in which `a0` contains 3 at the end, and contains 1 at some point during the trace.
- Give a trace in which `a0` contains 3 at the end, and contains 2 at some point during the trace.
- Give a trace in which `a0` contains 1 at the end.
- Give a trace in which `a0` contains 2 at the end.

We call the first two *good interleavings* and the second two *bad interleavings*.

Have us check your work!

$$\begin{array}{c}
\frac{\mu \mid e_1 \mapsto e'_1}{\mu \mid \text{Set}(e_1, e_2) \mapsto \mu \mid \text{Set}(e'_1, e_2)} \quad \text{Set-search-1} \\
\\
\frac{\mu \mid e_2 \mapsto e'_2 \quad e_1 \text{ done}}{\mu \mid \text{Set}(e_1, e_2) \mapsto \mu \mid \text{Set}(e_1, e'_2)} \quad \text{Set-search-2} \\
\\
\frac{}{(\mu \mid \text{Set}(\text{Num}(a), \text{Num}(v))) \mapsto (\text{update}(\mu, a, v) \mid \text{Noop})} \quad \text{Set-instr} \\
\\
\frac{\mu \mid e \mapsto e'}{\mu \mid \text{If}(e, s_1, s_2) \mapsto \mu \mid \text{If}(e', s_1, s_2)} \quad \text{If-search} \\
\\
\frac{k \neq 0}{\mu \mid \text{If}(\text{Num}(k), s_1, s_2) \mapsto \mu \mid s_1} \quad \text{If-true} \\
\\
\frac{}{\mu \mid \text{If}(\text{Num}(0), s_1, s_2) \mapsto \mu \mid s_2} \quad \text{If-false} \\
\\
\frac{\mu \mid s_1 \mapsto \mu' \mid s'_1}{\mu \mid \text{Sequence}(s_1, s_2) \mapsto \mu' \mid \text{Sequence}(s'_1, s_2)} \quad \text{Sequence-search} \\
\\
\frac{s_1 \text{ done}}{\mu \mid \text{Sequence}(s_1, s_2) \mapsto \mu \mid s_2} \quad \text{Sequence-done} \\
\\
\frac{}{\mu \mid \text{While}(e, s) \mapsto \mu \mid \text{If}(e, \text{Sequence}(s, \text{While}(e, s)), \text{Noop})} \quad \text{While-expand} \\
\\
\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{LocalVar}(x.s) \mapsto \text{update}(\mu, a, 0) \mid s[\text{Num}(a)/x]} \quad \text{Declare} \\
\\
\frac{\text{Func}(f, x.s) \in p}{\mu \mid \text{Call}(f, e) \mapsto \mu \mid \text{LocalVar}(x, \text{Sequence}(\text{Set}(x, e), s))} \quad \text{Call}
\end{array}$$

Figure 1: Step rules

Next, we will add a `fork{s};` statement to the language, with abstract syntax `Fork(s)` where s is a statement. The idea is that `fork` adds a new thread to the thread pool, and returns immediately in the thread in which it occurred. For example

should step to the thread pool

where the original thread continues with the code after the fork, while the new thread runs `x := get(x) + 2;`

which is

In this statement, we only find the fork by doing the search rule Sequence-search several times.

To handle this, we will change the operational semantics for a single statement from $\mu \mid s \mapsto \mu' \mid s'$ to have an extra output $\mu \mid s \mapsto \mu' \mid s' \mid \vec{s}$, where the output thread pool $\vec{s}$ contains any threads that s forks.

μ	$\text{Fork}(s) \mapsto$		
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			
46			
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			
59			
60			
61			
62			
63			
64			
65			
66			
67			
68			
69			
70			
71			
72			
73			
74			
75			
76			
77			
78			
79			
80			
81			
82			
83			
84			
85			
86			
87			
88			
89			
90			
91			
92			
93			
94			
95			
96			
97			
98			
99			
100			

Task 2. Modify each of the rules in Figure 1 to output a (possibly empty) thread pool.

Task 4. Modify the rule

$$\frac{\mu \mid s_i \mapsto \mu' \mid s'_i \mid \vec{s}}{\mu \mid s_1 \parallel \dots \parallel s_i \parallel \dots \parallel s_n \mapsto \mu \mid s_1 \parallel \dots \parallel s'_i \parallel \dots \parallel s_n} \text{ step-threads}$$

to incorporate the forked theads $\vec{s}$ output by the premise.

Task 3. Give a trace in which

```
{var x; fork(x := get(x) + 1); fork(x := get(x) + 2); }
```

steps to the thread pool

```
a0 := get(a0) + 1; || a0 := get(a0) + 2;
```

Have us check your work!

3 Locks

3.1 Return addresses

In C, it is not entirely necessary for functions to return a value, because one can simulate a return value by assigning to a memory location provided by the caller of the function. For example,

```
int f () {
    return 1;
}
int main () {
    int x;
    x = f();
}
```

is morally the same as

```
void f (int* return_address) {
    *return_address = 1;
}

int main () {
    int x;

    int ret;
    f(&ret);
    x = ret;
}
```

which is the same as

```
f (var return_address) {
    get(return_address) := 1;
}

main () {
    {var x;
    {var ret;
    f(ret);
    x := get(ret);
    }
    }
}
```

Since we haven't implemented functions with return values yet, when we say a function "returns", we mean it in this sense.

3.2 Test and Set

One way to avoid the bad interleavings from Section 1 is to *lock* the variable x in each thread before using it. While a thread holds the lock, no other thread is supposed to read or write x . To implement locks, we will need a new operation called *test and set*.

First, we will describe an implementation of test and set in our existing language IMP and then we will add it as a new primitive.

Morally, test-and-set takes a memory location a , which should contain 0 or 1; we call this a *flag*. Test-and-set always sets the flag to 1. The test-and-set also returns the value of the flag a at the *beginning* of the call to test-and-set, so that the caller can see if the bit was changed (beginning value was 0) or wasn't changed (beginning value is 1).

Task 1. Implement test-and-set. Morally, a and return have type int^* , so a is the address of a local variable, $\text{get}(a)$ is the address of the flag, and $\text{get}(\text{get}(a))$ is the current value of the flag.

```
testAndSet(var a, var return) {
```

```
}
```

Task 2. Suppose two threads are running test-and-set on the same flag at the same time:

```
main() {
  {var a;
   a := 0;
  {var return1;
   {var return2;
    fork{testandset(a, return1);};
    fork{testandset(a, return2);};
   }}}
};
```

The intention is that one of return1 will be 0, for the thread that goes first and sets the flag) and the other will be 1, for the thread that goes second and sees the flag is already set. Explain why this implementation allows bad interleavings in which both threads see the flag as unset and both return 0.

Task 3. We can fix this problem by adding *test-and-set* as a new statement, which does both the get and the set in a single step, so that no other threads can be interleaved in between them. This corresponds to how test-and-set or other concurrency primitives are implemented as a single instruction in hardware. We have a statement $\text{TestAndSet}(e_1, e_2)$ where e_1 evaluates to the address of the flag and e_2 evaluates to the address for the return value.

Add a rule for

$$\frac{}{\mu \mid \text{testAndSet}(\text{Num}(a), \text{Num}(r)) \mapsto \text{testAndSet-instr}}$$

(We will also need the usual search rules for evaluating e_1 and e_2 .)

Task 4. Finally, we use test-and-set to implement a *lock*, with the following interface:

- The lock is an int variable, which is either 0 or 1.
- The lock starts off unlocked (0).
- If the lock is unlocked (0), the lock can be *acquired* by setting the lock to 1. If another process has already locked the lock, then a thread that tries to acquire the lock should wait until the lock is unlocked.
- If the lock is locked by a thread (1), it can be *released* by setting it to 0.

Task 4. Finish implementing the lock by implementing `acquire`.

```
acquire(var lock) {
    {var before;

}

}

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); };
      }
    }
}

;
```

Adding the lock to the code from Section 1 should make the bad interleavings impossible.

If you finish all of that, have us check your work and ask me for a bonus problem.