

COMP 321 Lecture 07: Shared-memory Concurrency

Dan Licata

Fall 2026

1 Digression: 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.

2 Introduction to Concurrency

This week, we will extend the language we have developed so far with *shared-memory concurrency*.

The idea of concurrency is that a program is explicitly structured using multiple *threads*, which each independently take steps of execution. At any point, the programming language can freely choose which thread to step, ideally in a “fair” way where every thread gets to step as often as the others. Concurrency is also called *multithreading*. One typical use of concurrency is for programs that have a graphical user interface (GUI). The program might have one (or more than one) thread for the GUI, which is responsible for receiving user input from the keyboard/mouse/etc. and for displaying things, and another thread for the “logic” of the program. With multithreading, both the logic and GUI will interleave taking steps. This way, if the logic thread runs a long computation, the GUI thread will also continue to run, so that the user interface doesn't freeze up. If the logic and GUI were in a single thread, then the long computation would need to finish before the GUI is run again. Another common use of threads is when a program has multiple “clients” and wants to alternate between working for each of them — as a running example, we will think about processing transactions for a bank.

It is important to distinguish concurrency from *parallelism*, which is when computations actually happen simultaneously on multiple processors. One can run a concurrent program without parallelism on a single processor, by having the threads take turns stepping on that one processor. One can also have parallelism without explicit concurrency in the program, as some of you saw in COMP 212, by stepping expressions to values on multiple processors, in such a way that the result is the same as if the program was run on a single processor.

There are many different ways that programming languages allow threads to communicate with each other. For today, we will look at the simplest one to define, which is called *shared-memory concurrency*: the threads can read and write the same memory. A single thread will be represented by a statement. A concurrent program is represented by a *thread pool*, which is a list of statements that we write as $s_1 \parallel s_2 \dots \parallel s_n$. Intuitively, to step the thread pool, we choose a single thread and step it according to the rules we have developed so far (which may update the shared memory), and then put the next step back in the thread pool.

3 Examples

Suppose `account1` and `account2` are two addresses (e.g. 1 and 2)¹ containing 100 and 50 in a memory $\mu = [(account1, 100), (account2, 50)]$. Here is a thread pool:

¹for readability, we write these addresses like variables, but formally some specific numeric memory address would be substituted for the variables before stepping the program

```
account1 := (get(account1) + 10); || account2 := (get(account2) + 20);
```

One way this can step is

```
mu | account1 := (get(account1) + 10); || account2 := (get(account2) + 20);
|-> mu | account1 := (100 + 10); || account2 := (get(account2) + 20);
|-> mu | account1 := 110; || account2 := (get(account2) + 20);
|-> [(1,110), (2,50)] | noop; || account2 := (get(account2) + 20);
|-> [(1,110), (2,50)] | noop; || account2 := (50 + 20);
|-> [(1,110), (2,50)] | noop; || account2 := 70;
|-> [(1,110), (2,70)] | noop; || noop;
```

A sequence of steps is called a *trace*. In this trace, the first thread runs first and adds 10 to the first account, then the second thread runs and adds 10 to the second account.

Here is a different trace in which the second thread goes first:

```
mu | account1 := (get(account1) + 10); || account2 := (get(account2) + 20);
|-> mu | account1 := (get(account1) + 10); || account2 := (50 + 20);
|-> mu | account1 := (get(account1) + 10); || account2 := 70;
|-> [(1,100), (2,70)] | account1 := (get(account1) + 10); || noop;
|-> [(1,100), (2,70)] | account1 := (100 + 10); || noop;
|-> [(1,100), (2,70)] | account1 := (110); || noop;
|-> [(1,110), (2,70)] | noop; || noop;
```

However, in a multithreaded program, it is not the case that one thread runs to completion and then the other thread is run; we can also have traces where the steps of the two threads are *interleaved*.

```
mu | account1 := (get(account1) + 10); || account2 := (get(account2) + 20);
|-> mu | account1 := (100 + 10); || account2 := (get(account2) + 20);
|-> mu | account1 := (100 + 10); || account2 := (500 + 20);
|-> mu | account1 := 110; || account2 := (50 + 20);
|-> [(1,110), (2,50)] | noop; || account2 := (50 + 20);
|-> [(1,110), (2,50)] | noop; || account2 := 70;
|-> [(1,110), (2,70)] | noop; || noop;
```

In this trace, the first thread runs for 1 step, then the second thread for 1 step, then the first thread for 2 steps, then the second thread for 2 steps.

The *thread scheduler* in the programming language is formally free to choose any thread at any point, based on considerations like fairness (each thread should get to step every so often, rather than one thread getting all the attention) and maximizing overall efficiency (sometimes a thread will not be able to step, especially considering the fact that some instructions (like reading/writing memory) take much longer than others (like arithmetic)).

3.1 Fork

How do we get a thread pool in the first place? We will add a new statement called `fork{s}`; that adds the thread `s` to the thread pool. In the thread that runs the `fork`, the fork returns immediately, in the sense that it does not first run `s` and then return. Here is a complete program that can step to the thread pool above

```

main() {
    {var account1;
      account1 := 100;
      {var account2;
        account2 := 50;
        fork { account1 := (get(account1) + 10); };
        fork { account2 := (get(account2) + 20); };
      }}
};

```

3.2 Race Conditions

Here is a slightly more involved example. Instead of just setting a variable in each thread, we call a function that transfers money from one bank account to another:

```

transfer(var account_from, var account_to, var amount) {
    get(account_from) := (get(get(account_from)) - get(amount));
    get(account_to) := (get(get(account_to)) + get(amount));
}

main() {
    {var account1;
      account1 := 100;
      {var account2;
        account2 := 10;
        {var account3;
          account3 := 30;
          fork { transfer(account1, account2, 10); };
          fork { transfer(account1, account3, 10); };
        }}}
};

```

(If you're confused by the gets: In main, account1 is substituted with the memory address allocated to the local variable account1. In transfer, account_from stands for the memory address allocated for that local variable, get(account_from) stands for the address allocated for account1, and get(get(account_from)) stands for the contents of that address, which starts as 100.)

This is supposed to run and produce the memory [(account1, 80), (account2, 20), (account3, 40)]. There are many traces in which it does do this. Moreover, it shouldn't matter which transaction is processed first, as long as both transactions get processed correctly in some order: withdrawing 10 from account 1 and then withdrawing 10 again will leave 80 in account 1, no matter which transaction's withdrawal happens first. Depositing 20 into account 2 and then into account 3 will leave 20 in account 2 and 40 in account 3, as will doing the deposits in the opposite order. So this seems like a good candidate for concurrency to structure a program in which the bank can interleave transactions on behalf of multiple customers.

However, there are also incorrect traces in which we get the wrong value in account 1! Here is the key part of one:

```
memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->1
```

```

statement
    3 := (get (3) - get (12)); get (8) := (get (get (8)) + get (12));
|| 3 := (get (3) - get (15)); get (13) := (get (get (13)) + get (15));

memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
    3 := (100 - get (12)); get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := (get (3) - get (15)); get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    3 := (100 - 10); get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := (get (3) - get (15)); get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    3 := (100 - 10); get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := (100 - get (15)); get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    3 := 90; get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := (100 - get (15)); get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    3 := 90; get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := (100 - 10); get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->100 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    3 := 90; get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := 90; get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->90 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    noop; get (8) := (get (get (8)) + get (12)); noop; noop;
|| 3 := 90; get (13) := (get (get (13)) + get (15)); noop; noop;

memory [16] 0->0 1->0 2->0 3->90 4->10 5->30 6->3 7->0 8->4 9->3 10->2 11->0 12->10
statement
    get (8) := (get (get (8)) + get (12)); noop; noop;
|| noop; get (13) := (get (get (13)) + get (15)); noop; noop;

```

The issue is that both threads step `get (3)` to get the value 100 from location 3 before either thread steps the `set (:=)` to update the value in location 3. Because of this, the final value in location 3 will incorrectly be 90 instead of 80 (yay, free money?).

This is called a *race condition*: two threads are using the same memory location, with at least one of them writing to the memory, so the order in which the threads are scheduled influences the result.

Here are some simpler examples of programs that have race conditions. Both threads write:

```
x := 20; || x := 30;
```

If thread 1 goes first, x will contain 30 at the end; if thread 2 goes first, 20.

One writes and the other reads:

```
x := 20; || y := get(x);
```

If thread 1 goes first, y will get 20; if thread 2 goes first, it will get the old value of x.

3.3 Using Locks

One way to fix this problem is using a *lock*, which is a way for a thread to (hopefully) prevent anyone else from using some memory while it modifies it. (The “hopefully” is because, the language we will implement this week does not *guarantee* that all programs use the lock appropriately; the programmer must make sure to code this correctly.) A lock will be represented by a memory location that contains 0 (unlocked) or 1 (locked):

- 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 thread 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.

Here is how to fix the above example using locks:

```
acquire(var lock) { ... }
release(var lock) { ... }

transfer(var account_from, var lock_from, var account_to, var lock_to, var amount)
  acquire(get(lock_from));
  get(account_from) := (get(get(account_from)) - get(amount));
  release(get(lock_from));

  acquire(get(lock_to));
  get(account_to) := (get(get(account_to)) + get(amount));
  release(get(lock_to));
}

main() {
  {var lock1;
   lock1 := 0;
   {var lock2;
    lock2 := 0;
```

```

    {var lock3;
      lock3 := 0;
    {var account1;
      account1 := 100;
      {var account2;
        account2 := 10;
        {var account3;
          account3 := 30;
          fork { transfer(account1, lock1, account2, lock2, 10); };
          fork { transfer(account1, lock1, account3, lock3, 10); };
        }}}}}}
  };

```

The program creates 3 locks, one for each account. The `transfer` function acquires the corresponding lock before updating the value in an account, and then releases it after. This means that while the first transfer is getting and then setting `account1`, the second thread cannot also get account 1, preventing the problem trace we saw above.

3.4 Implementing Locks

How do we implement a lock? Release is easy – just set it to 0.

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

For acquiring, the main idea is to wait until the lock is available, which can be accomplished by looping as long as the lock is 1.

```

acquire(var lock) {
  while get(lock) { noop; }
  get(lock) := 1;
}

```

This code is a bit curious, because in a single-threaded program `while get(lock) { noop; }` would be an infinite loop. However, in a multi-threaded program, the idea is to wait until some other thread has set the lock to 0.

However, there is a problem with this code! The `acquire` function is vulnerable to exactly the same problem we saw above. If two threads try to acquire the lock at the same time, and the lock is unlocked, they might both proceed to get the value 0 before either sets it to 1, and so both threads will simultaneously think they have the lock, which defeats the purpose.

To correctly implement locks, we will need to add a new operation to the language called test-and-set, which simultaneously checks if the lock is available and sets it to 1 without other threads being interleaved. This is an example of a *hardware primitive* for implementing locking that a processor might have.

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).

Let `a` and `return` have type `int*`, so `a` is the address of a local variable, `get(a)` is the address of the flag, and `get(get(a))` is the current value of the flag. The memory-behavior of test-and-set is the same as the following code:

```
testAndSet(var a, var return) {
  get(return) := get(get(a));
  get(a) := 1;
}
```

This stores the current value of `a` in the pointer `return`, and then sets `a` to 1.

However, this implementation is still vulnerable to the usual problem, that two threads might independently do `get(get a)` before either sets it to 1. In the homework, you will add `testandset` to the language as a single operation.

4 Operational Semantics for Concurrency

Next, we will give a more precise treatment of the operational semantics of threads. Here are the solutions to Wednesday’s lab. The meta-operations of *update*, *lookup*, and substitution, and the rules for stepping expressions will stay unchanged from before. The updated operational semantics for stepping statements in Figure 1.

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*.

4.1 Fork

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

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

should step to the thread pool

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

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

One slightly tricky aspect is that the `fork` might be “deep” in a statement, in a left-nested sequence like

```
((fork(s); if e {s1} else {s2};) ; e1 := e2;) ...
```

which is

```
Sequence(Sequence(Sequence(Sequence(Fork(s), If(e, s1, s2)), Set(e1, e2)), ...))
```

$$\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) \mid \square} \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) \mid \square} \text{Set-search-2} \\
\\
\frac{}{(\mu \mid \text{Set}(\text{Num}(a), \text{Num}(v))) \mapsto (\text{update}(\mu, a, v) \mid \text{Noop}) \mid \square} \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) \mid \square} \text{If-search} \\
\\
\frac{k \neq 0}{\mu \mid \text{If}(\text{Num}(k), s_1, s_2) \mapsto \mu \mid s_1 \mid \square} \text{If-true} \\
\\
\frac{}{\mu \mid \text{If}(\text{Num}(0), s_1, s_2) \mapsto \mu \mid s_2 \mid \square} \text{If-false} \\
\\
\frac{\mu \mid s_1 \mapsto \mu' \mid s'_1 \mid \vec{s}}{\mu \mid \text{Sequence}(s_1, s_2) \mapsto \mu' \mid \text{Sequence}(s'_1, s_2) \mid \vec{s}} \text{Sequence-search} \\
\\
\frac{s_1 \text{ done}}{\mu \mid \text{Sequence}(s_1, s_2) \mapsto \mu \mid s_2 \mid \square} \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})} \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] \mid \square} \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)) \mid \square} \text{Call} \\
\\

\\
\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 \parallel \vec{s}} \text{step-threads}
\end{array}$$

Figure 1: Step rules

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.

Here is the rule for Fork:

$$\frac{}{\mu \mid \text{Fork}(s) \mapsto \mu \mid \text{Noop} \mid s}$$

This says that a fork statement doesn't change the memory μ , immediately steps to the done statement `Noop`, and signals that the single thread s should be added to the thread pool.

Next, we must modify each of the rules in Figure 1 to output a (possibly empty) thread pool. The modifications are in [blue](#). Each statement that does not involve a fork outputs the empty list of new threads. The search rule for sequencing outputs whatever threads are output by s_1 . This handles a situation like `Sequence(Fork( $s$ ),  $s_2$ )`, which should add s to the thread pool and run s_2 .

The modified rule `step-threads` in Figure 1 is mostly the same as before: choose some thread s_i and step it. The new ingredient is that whatever threads $\vec{s}$ are forked by the `step-a-single-thread-relation` $\mu \mid s_i \mapsto \mu' \mid s'_i \mid \vec{s}$ are added to the thread pool by appending them to make the list $s_1 \parallel \dots \parallel s_n \parallel \vec{s}$.