Concurrent Programming with CXL

RVR and possibly others

June 4, 2020

Chapter 1

On Concurrent Programming

Programming with concurrency is hard. On the one hand concurrency can make programs faster than sequential ones, but having multiple processes concurrently read and update shared variables and synchronize with one another makes programs more complicated than programs where only one thing happens at a time. Why is this? There are, at least, two reasons:

- The execution of a sequential program is mostly *deterministic*. If you run it twice with the same input, the same things will happen. Bugs are typically easily reproducible.
- Each statement and each function can be thought of as happening *atomically* because there is no other activity interfering with their execution.

The lack of determinism and atomicity in concurrent programs make them not only hard to reason about, but also hard to test. Running the same test of concurrent code twice is likely to produce two different results. More problematically, a test may trigger a bug only for certain "lucky" executions. Due to the probabilistic nature of concurrent code, some bugs may be highly unlikely to get triggered even when running a test millions of times. And even if a bug does get triggered, the source of the bug may be hard to find because it is hard to reproduce.

This book is intended to help people with understanding and developing concurrent code. In particular, it uses a new tool called CXL that help with *testing* concurrent code. The approach is based on *model checking*: instead of relying on luck, CXL will run *all possible executions* of a particular test program. So even if a bug is unlikely to occur, if the test *can* expose the bug it *will*. Importantly, if the bug is found, the model checker precisely shows how to trigger it in the smallest number of steps.

Model checking is not a replacement for formal verification. Formal verification proves that a program is correct. Model checking only verifies that a program is correct for some *model*. Think of a model as a test program. Because model checking tries every possible execution, this means that the test program needs to be relatively simple. In particular, it needs to have a relatively small number of reachable states. If the model is too large, it may run for longer than we may care to wait for or even run out of memory.

So why is this useful? Consider, for example, a sorting algorithm. Now suppose we create a test program, a model, that tries sorting *all* lists of up to five numbers chosen from the set { 1, 2, 3, 4, 5 }. Model checking proves that for those particular scenarios the sorting algorithm works: the output is a sorted permutation of the input. In some sense it is an excellent test: it will have considered all corner cases, including lists where all numbers are the same, lists that are already sorted or reversely sorted, etc. If there is a bug in the sorting algorithm, most likely it would be triggered and the model checker would produce a scenario that would make it easy to find the source of the bug. However, if the model checker does not find any bugs, we do not know necessarily that the algorithm works for lists with more than five numbers or for lists that have values other than the numbers 1 through 5. Still, we would expect that the likelihood that there are bugs remaining in the sorting algorithm is small.

But model checking can also help with proving an algorithm correct. The reason is that the correctness of many programs is built around *invariants*: predicates that must hold for every state in the execution of a program. A model checker can find violations of those invariants when evaluating a model and provide valuable early feedback to somebody who is trying to construct a proof. Throughout this book we will include, as much as possible, examples of such invariants as they often provide excellent insight into why a particular algorithm works.

So what is CXL? CXL is a concurrent programming language. It was designed to teach the basics of concurrent programming, but it is also useful for testing new concurrent algorithms or even sequential and distributed algorithms. CXL programs are not intended to be "run" like programs in most other programming languages—instead CXL programs are model checked to verify that the program has certain desirable properties and does not suffer from bugs.

The syntax and semantics of CXL is similar to that of Python. Python is familiar to many programmers and is easy to learn and use. We will assume that the reader is familiar with the basics of Python programming. We also will assume that the reader understand some basics of machine architecture and how programs are executed. For example, we assume that the reader is familiar with the concepts of CPU, memory, register, and stack.

Chapter 2

Introduction to Programming with CXL

Like Python, CXL is an imperative, dynamically typed, and garbage collected programming language. There are also some important differences:

- Every statement in CXL must be terminated by a semicolon (even `while` statements and `def` statements); in Python semicolons are optional and rarely used;
- Correct indentation in CXL is encouraged but optional;
- Python is object-oriented; CXL is not;
- CXL does not (currently) support floating point;
- CXL does not support I/O of any kind;
- CXL only supports basic operator precedence or associativity. Use parentheses liberally to remove ambiguity.

There are also many unimportant ones that you will discover as you get more familiar with programming in CXL.

Figure 2.1 gives a simple example of a CXL program. It is sequential and has a method `square` that takes an integer number as argument. Each method has a variable called `result` that eventually contains

```
const N = 10;

def triangle(n):   # computes the n'th triangle number
    result = 0;
    for i in 1..n:
        result = result + i;
    ;
;
x = choose(0..N);
assert triangle(x) == ((x * (x + 1)) / 2);
```

Figure 2.1: Computing triangle numbers.

the result of the method (there is no `return` statement in CXL). The method also has a variable called `n` containing the value of the argument. The `x..y` notation generates a set containing the numbers from `x` to `y` (inclusive). The last two lines in the program are the most interesting. They assign to `x` some unspecified value in the range `0..N` and then verifies that `triangle`(x) equals $x(x+1)/2$.

"Running" this CXL program will try all possible executions, which includes all possible values for x. Try it out:

```
$ ./cxl triangle.cxl
#states = 13
no issues found
$
```

Essentially, the `choose`(S) operator provides the input to the program by selecting some value from the set S, while the `assert` statement checks that the output is correct. If the program is correct, the output of CXL is the size of the "state graph" (13 states in this case). If not, CXL also reports what went wrong, typically by displaying a summary of an execution in which something went wrong.

Experiment!

- See what happens if, instead of initializing `result` to 0, you initialize it to 1.
- Write a similar program that computes squares or factorials.

Chapter 3

The Problem of Concurrent Programming

Concurrent programming, aka multithreaded programming, involves multiple processes running in parallel while sharing variables. Figure 3.1 presents a simple example. The program initializes two shared variables: an integer `count` and an array `done` with two booleans. Method `incrementer` takes a parameter called `self`. It increments `count` and sets `done[self]` to `True`. Method `main` waits for the processes to finish by polling both `done` flags. After that, it verifies that the value of `count` equals 2. If not, it reports the actual value of `count`. The program spawns three processes. The first runs `incrementer(0)`, the second runs `incrementer(1)`, and the last runs `main()`.

What will happen?

- Before you run the program, what do you think will happen? Is the program correct in that `count` will always end up being 2?
- What are the possible values of `count` at the time of the `assert` statement? (You may assume that `load` and `store` instructions of the underlying architecture are atomic—in fact they are.)

What is going on is that the CXL program is compiled to machine instructions, and it is the machine instructions that are executed by the underlying CXL machine. The details of this appear in Chapter 5, but suffice it to say that the machine has instructions that load values from memory and store values into memory. Importantly, it does not have instructions to atomically increment or decrement values in memory locations. So to increment a value in memory, the machine must do at least three machine instructions. Conceptually:

1. load the value from the memory location;
2. add 1 to the value;
3. store the value to the memory location.

When running multiple processes, each essentially runs an instantiation of the machine, and they do so in parallel. As they execute, their machine instructions are interleaved in unspecified and often random ways. A program is correct if it works for any interleaving. In fact, CXL will try all possible interleavings of the processes executing machine instructions.

The following is a possible interleaving of incrementers 0 and 1:

1. incrementer 0 loads the value of `count`, which is 0;

```
def incrementer(self):
    count = count + 1;
    done[self] = True;
;
def main(self):
    while not (done[0] and done[1]):
        pass;
    ;
    assert count == 2, count;
;
count = 0;
done = [ False, False ];
spawn incrementer(0);
spawn incrementer(1);
spawn main();
```

Figure 3.1: Incrementing twice in parallel.

2. incrementer 1 loads the value of `count`, which is still 0;
3. incrementer 1 adds one to the value that it loaded (0), and stores 1 into `count`;
4. incrementer 0 adds one to the value that it loaded (0), and stores 1 into `count`;
5. incrementer 0 sets `upDone` to `True`;
6. incrementer 1 sets `upDone` to `True`.

The result in this particular interleaving is that `count` ends up being 1. When running CXL, it will report violations of assertions. It also provides an example of an interleaving, like the one above, in which an assertion fails.

If one thinks of the assertion as providing the specification of the program, then clearly its implementation does not satisfy its specification. Either the specification or the implementation (or both) must have a bug. We could change the specification by changing the assertion as follows:

```
    assert (count == 1) or (count == 2);
```

This would fix the issue, but more likely it is the program that must be fixed.

Can you think of a fix to the program? You can try it out and see if it works or not.

Chapter 4

CXL Values

CXL programs manipulates CXL values. CXL values are recursively defined: they include booleans (`False` and `True`), integers (but not floating point numbers), strings (enclosed by double quotes), sets of CXL values, and dictionaries that map CXL values to other CXL values.

Another type of CXL value is the *atom*. It is essentially just a name. An atom is denoted using a period followed by the name. For example, `.main` is an atom.

CXL makes extensive use of dictionaries. Their syntax and properties are a little different from Python. A dictionary maps CXL values, known as *keys*, to CXL values. Unlike Python, any CXL value can be a key, including another dictionary. CXL dictionaries are written as `dict`$\{k_0 : v_0,\ k_1 : v_1,\ ...\}$. If `d` is a dictionary, and `k` is a key, then the following expression retrieves the CXL value that k maps to:

```
d k
```

This is unfamiliar, but in CXL square brackets can be used in the same way as parentheses, so you can express it in the more familiar form:

```
d[k]
```

However, if `d = dict{ .count: 3 }`, then you can write can write `d.count` (which has value 3) instead of having to write `d[.count]` (although both will work). The meaning of `d a b c ...` is `((((d a) b) c) ...)`.

Tuples are special forms of dictionaries where the keys are the indexes into the tuple. For example, the tuple `(5, False)` is the same CXL value as `dict{ 0: 5, 1: False }`. The empty tuple `()` is the same value as `dict{}`. Note that this is different from the empty set, which is {}. As in Python, you can create singleton tuples by adding an extra ",". For example, `(1,)`.

Again, square brackets and parentheses work the same in CXL, so `[a, b, c]` (which looks like a Python list) is the same CXL value as `(a, b, c)` (which looks like a Python tuple), which in turn is the same CXL value as `dict{ 0:a, 1:b, 2:c }`. So if `x == [ False, True ]`, then `x[0] == False` and `x[1] == True`, just like in Python. However, when creating a singleton list, make sure you include the comma, as in `[False,]`.

CXL is not an object-oriented language, so objects don't have built-in methods. However, CXL does have some powerful operators to make up for some of that. For example, dictionaries have two handy unary operators. If `d` is a dictionary, then `keys d` (or equivalently `keys(d)`) returns the set of keys and `len d` returns the size of this set.

Chapter 5

The CXL Machine

Before we delve into how to solve synchronization problems, it is important to know a bit about the underlying machine. A CXL program is translated into a list of machine instructions that the CXL machine executes. The CXL machine is not an ordinary virtual machine, but its architecture is nonetheless representative of conventional machines such as hardware with memory and CPUs and virtual machines such as the Java Virtual Machine.

Instead of bits and bytes, a CXL machine manipulates CXL values. A CXL machine has the following components:

- Code: This is an immutable and finite list of CXL machine instructions, generated from a CXL program. The types of instructions will be described later.
- Shared memory: A CXL machine has just one memory location containing an CXL value.
- Processes: Any process can spawn an unbounded number of other processes and processes may terminate. Each process has an immutable (but not necessarily unique) *name tag*, a program counter, a stack of CXL values, and a single mutable general purpose *register* that contains a CXL value.

A name tag consists of the name of the main method of the process, along with an optional tag specified in the `spawn` statement. The default tag is `()`. In Figure 3.1, the created processes have name tags `incrementer/0`, `incrementer/1`, and `main/()`.

The state of a process is called a *context*: it contains the value of its name tag, program counter, stack, and register. The state of the CXL machine consists of the value of its memory and the multiset (or *bag*) of contexts. It is a multiset of contexts because two different processes can be in the same state. The initial state of the CXL memory is the empty tuple, `()`. The context bag has an initial context in it with name tag `__main__/()`, pc 0, register `()`, and an empty stack. Each machine instruction updates the state in some way. Figure **??** shows an example of a reachable state for the program in Figure 3.1.

It may seem strange that there is only one memory location and that each process has only one register. However, this is not a limitation because CXL values are unbounded trees. Both the memory and the register of a process always contain a special type of CXL value in which the root is a dictionary that maps atoms to CXL values. We call this a *directory*. A directory represents the state of a collection of variables named by the atoms.

Because directories are CXL values themselves, directories can be organized into a tree. Each node in the tree is then identified by a sequence of atoms, like a path name in the file system hierarchy. We call such a sequence the *address* of a CXL value.

Compiling the code in Figure 3.1 results in the CXL machine code listed in Figure 5.1. The CXL machine is predominantly a *stack machine*. Most instructions pop values from the stack or push values onto the stack. Initially there is one process that starts executing at instruction 0. In this case, there is a `JUMP` instruction that sets the program counter to 22. At program counter 1 is the code for the `up` method. All methods start with a `Frame` instruction and end with a `Return` instruction.

```
Up.cxl:1 def incrementer(self):
    0 Jump 39
    1 Frame incrementer ['self'] 12
Up.cxl:2     count = count + 1;
    2 Push 1
    3 PushAddress count
    4 Load 1
    5 2-ary +
    6 PushAddress count
    7 Store 1
Up.cxl:3     done[self] = True;
    8 Push True
    9 PushVar self
   10 PushAddress done
   11 Store 2
   12 Return
```

Figure 5.1: The first part of the machine code corresponding to Figure 3.1.

```
CXL Assertion failed ('assert', 'Up.cxl', 9, 5) 1
#states = 79
==== Safety violation ====
__init__/() [0,39-56]       57 dict{ .count:0, .done:dict{ 0:False, 1:False } }
incrementer/0 [1-6]          7 dict{ .count:0, .done:dict{ 0:False, 1:False } }
incrementer/1 [1-11]        12 dict{ .count:1, .done:dict{ 0:False, 1:True  } }
incrementer/0 [7-11]        12 dict{ .count:1, .done:dict{ 0:True,  1:True  } }
main/() [14-19,22-27,29-36] 37 dict{ .count:1, .done:dict{ 0:True,  1:True  } }
```

Figure 5.2: The output of running Figure 3.1.

The `Frame` instruction has three arguments:

1. The name of the method;
2. A list containing the names of the arguments of the method;
3. The program counter of the method's `Return` instruction.

The code generated from `count := count + 1` in line 2 of `Up.cxl` is as follows:

2. the `Push` instruction pushes the constant 1 onto the stack of the process.
3. the `PushAddress` instruction pushes the address of the shared variable `count` onto the stack.
4. The `Load` instruction pops the address of the `count` variable and then pushes the value of the `count` variable onto the stack.
5. `2-ary` is a `+` operation with 2 arguments. It pops two values from the stack (1 and the value of `count`), adds them, and pushes the result back onto the stack.
6. The `PushAddress` instruction pushes the address of the `count` variable onto the stack.
7. The `Store` instruction pops the address of a variable and pops a CXL value (the sum of the `count` variable and 1), and updates the `count` variable.

Figure 5.2 shows the output produced by running the `Up.cxl` program. It starts by reporting that the assertion on line 12 failed and that `count` has value 1 instead of 0: `Assertion failed ('assert', 'Up.cxl', 12, 1) 1`. Next it reports an execution that failed this assertion. The output has four columns:

1. The name tag of the process;
2. The sequence of program counters of the CXL machine instructions that the process executed;
3. The current program counter of the process;
4. The contents of the shared memory.

If we look at the middle three rows, we see that:

1. Process 0 executed instructions 1 through 7, loading the value of `count` but stopping just before storing 1 into `count`;
2. Process 1 executed instructions 1 through 12, storing 1 into into `count` and storing `True` into `done[2]`;
3. Process 0 continues execution, storing value 1 into `count` and storing `True` into `done[1]`.

This makes precise the concurrency problem that we encountered.

Chapter 6

Critical Sections

Hopefully you have started thinking of how to solve the concurrency problem and you may already have prototyped some solutions. In this chapter we will go through a few reasonable but broken attempts. At the heart of the problem is that we would like make sure that when the `count` variable is being updated that no other process is trying to do the same thing. We call this a *critical section*: a set of instructions where only one process is allowed to execute at a time.

Critical sections are useful when accessing a shared data structure, particularly when that access requires multiple underlying machine instructions. A simple counter is a very simple example of a data structure. A more involved one would be access to a binary tree. Adding a node to a binary tree or re-balancing a tree often requires multiple operations. Maintaining consistency is certainly much easier (although not necessarily impossible) if during this time no other process also tries to access the binary tree.

A critical section is often modeled as processes in an infinite loop entering and exiting the critical section. In CXL:

```
while True:
    @ncs: pass; # in non-critical section
    # enter critical section
    @cs: pass;  # in critical section
    # exit critical section
;
```

```
def process(self):
    while True:
        @ncs: pass;
        #enter critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        #exit critical section
    ;
;
spawn process(0), 0;
spawn process(1), 1;
```

Figure 6.1: A barebones critical section.

```
def process(self):
    while choose({ False, True }):
        #enter critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        #exit critical section
    ;
;
spawn process(0), 0;
spawn process(1), 1;
```

Figure 6.2: CXL model of a critical section.

Here `@ncs` and `@cs` are *labels*, each identifying a location in the CXL machine code. The first thing we need to ensure is that there cannot be two processes at the critical section. We would like to place an assertion at the `@cs` label that specifies that only the current process can be there. CXL in fact supports this. It has an operator `atLabel` L, where L is the atom of the label (in this case, `.cs`). The operator returns a bag of name tags of processes executing at that label. The bag is represented by a dictionary that maps each element in the bag to the number of times the element appears in the bag. Method `atLabel` only exists for specification purposes—do not use it in normal code.

Figure 6.1 shows the code. The assertion also makes use of the `nametag()` operator that returns the name tag of the current process. If you run the code through CXL, the assertion should fail because there is no code yet for entering and exiting the critical section.

However, mutual exclusion by itself easy to ensure. For example, we could insert the following code to enter the critical section:

```
while True:
    pass;
;
```

This code will surely prevent two or more processes from being at label `cs` at the same time. But it does so by preventing *any* process from reaching the critical section. We clearly need another property besides mutual exclusion.

Mutual exclusion is an example of a *safety property*, a property that ensures that *nothing bad will happen*, in this case two processes being in the critical section. What we need now a *liveness property*: we want to ensure that *eventually something good will happen*. There are various possible liveness properties we could use, but here we will propose the following informally: if some set of processes S are trying to enter the critical section and any process already in the intersection eventually leaves, then eventually one process in S will enter the critical section. We call this *progress*.

In order to detect violations of progress, and other liveness problems in algorithms in general, CXL requires that every execution must be able to reach a state in which all processes have terminated. Clearly, even if mutual exclusion holds in Figure 6.1, the spawned processes never terminate. In order to resolve this, we will model processes in critical sections using the framework in Figure 6.2: a process can *choose* to enter a critical section more than once, but it can also choose to terminate, even without entering the critical section ever. Note also that we have dropped the `@ncs` label: it would serve no specific purpose. Figure 6.3 show a high-level state diagram of the code in Figure 6.2.

Try it out!

- Plug `while True: pass;;` for entering the critical section in Figure 6.2 and run CXL. It should print a trace to a state from which a terminating state cannot be reached.

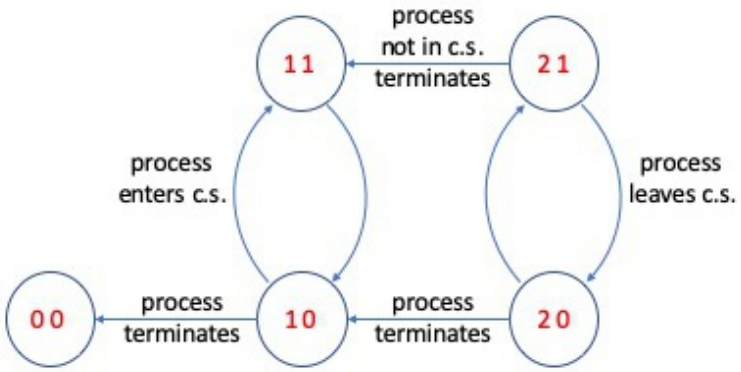

Figure 6.3: High-level state diagram specification of mutual exclusion with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes in the critical section.

Chapter 7

Naïve Attempts at Implementing Critical Sections

In order to have at most one process in a critical section at a time, you probably are thinking that one might use a *lock*. That is a good thought, but how does one implement one? Figure 7.1 presents a mutual exclusion attempt based on a naïve implementation of a lock. The idea is that the lock is like a baton that at most one process can own (or hold) at a time. Initially the lock is not owned, indicated by lock being `False`. To enter the critical section, a process waits until `lock` is `False` and then sets it to `True` to indicate that the lock has been taken. The process then executes the critical section. Finally the process releases the lock by setting it back to `False`.

Unfortunately, if we run the program, we find that the assertion still fails. Diagnosing the problem, we see that the `lock` variable suffers from the same problem as the `count` variable in Figure 3.1: operations on it consist of several instructions. It is thus possible for both processes to believe the lock is available and to obtain the lock at the same time.

Figure 7.2 presents a solution based on each process having a flag indicating that it is trying to enter the critical section. A process can write its own flag and read the flag of its peer. After setting its flag, the process waits until the other process (`1 - self`) is not trying to enter the critical section. If we run this program, the assertion does not fail. In fact, this solution does prevent both processes being in the critical section at the same time.

To see why, first note the invariant that if process i is in the critical section, then flags[i] == True. Without loss of generality, suppose that process 0 sets `flags[0]` at time t_0. Process 0 can only reach the critical section if at some time t_1, $t_1 > t_0$, it finds that `flags[1] == False`. Because of the invariant, `flags[1] == False` implies that process 1 is not at the critical section at time t_1. Let t_2 be the time at which process 0 sets `flags[0]` to `False`. Process 0 is in the critical section sometime between t_1 and t_2. It is easy to see that process 1 cannot enter the critical section between t_1 and t_2, because `flags[1] == False` at time t_1. To reach the critical section between t_1 and t_2, it would first have to set `flags[1]` to `True` and then wait until `flags[0] == False`. But that does not happen until time t_2.

However, the solution has a different problem called *livelock*: if both try to enter the critical section at the same time, they may end up waiting for one another indefinitely. Thus the solution violates *progress*.

The final naïve solution is based on a variable called `turn` that alternates between 0 and 1. When `turn` = i, process i can enter the critical section, while process $1 - i$ has to wait. When done, process i sets `turn` to $1 - i$ to give the other process an opportunity to enter. An invariant of this solution is that while process i is in the critical section, `turn` == i. (If you do not believe that, replace the assertion at the critical section with `assert turn == self` and run CXL.) Since `turn` cannot be 0 and 1 at the same time, mutual exclusion is satisfied. The solution also has the nice property that processes 0 and 1 alternate entering the critical section. The problem with the solution is that it also violates the *progress* property: if process i terminates

```
def process():
    while choose({ False, True }):
        # Enter critical section
        while lock:
            pass;
        ;
        lock = True;

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        lock = False;
    ;
;
lock = False;
spawn process(), 0;
spawn process(), 1;
```

Figure 7.1: Naïve implementation of a shared lock.

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        while flags[1 - self]:
            pass;
        ;

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
spawn process(0), 0;
spawn process(1), 1;
```

Figure 7.2: Naïve use of flags to solve mutual exclusion.

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        while turn == (1 - self):
            pass;
        ;

        # Critical section
        @cs: assert atLabel.cs == dict{ nametag(): 1 };

        # Leave critical section
        turn = 1 - self;
    ;
;
turn = 0;
spawn process(0), 0;
spawn process(1), 1;
```

Figure 7.3: Naïve use of turn variable to solve mutual exclusion.

instead of entering the critical section when it is process i's turn, process $1 - i$ ends up waiting indefinitely for its turn.

Chapter 8

Peterson's Algorithm

In 1981, Peterson came up with a beautiful solution to the mutual exclusion problem, now known as "Peterson's Algorithm" [**?**]. The algorithm is an amalgam of the (incorrect) algorithms in Figures 7.2 and 7.3, and is presented in Figure 8.1. A process first indicates its interest in entering the critical section by setting its flag. It then politely gives way to the other process should it also want to enter the critical section—if both do so at the same time one will win because writes to memory in CXL are atomic. The process continues to be polite, waiting in a `while` loop until either the other process is no where near the critical section or has given way. Running the algorithm with CXL shows that it satisfies both mutual exclusion and progress.

Why does it work? We will focus here on how one might go about proving mutual exclusion for an algorithm such as Peterson's. For that, we have to understand a little bit more about how the CXL machine works. In Chapter 5 we talked about the concept of *state*: at any point of time the CXL machine is in a specific state. Everytime a process executes a CXL machine instruction, the state changes (if only because the program counter of the process is updated). We call that a *step*. Steps in CXL are atomic.

The CXL machine starts in an initial state in which there is only one process and its program counter is 0. A *trace* is a sequence of steps starting from the initial state. When making a step, there are two sources of non-determinism in CXL. One is when in some state more than one process can make a step. The other is when a process executes a `choose` operation and there is more than one choice.

A good way to prove mutual exclusion is through induction on the number of steps. In such a proof, we need a so-called *inductive invariant* I, which is some predicate that holds over the state. We need I to satisfy the following:

- (ME1) I should imply mutual exclusion.
- (ME2) I should hold in the initial state.
- (ME3) If I holds over some state s, and some process in the state takes a step, then I should also hold in the resulting state.

One candidate for such an invariant is mutual exclusion itself. After all, it certainly satisfies ME1 and ME2. And as CXL has already determined, mutual exclusion is an invariant: it holds over every *reachable state*. Unfortunately, it is not an *inductive* invariant and therefore does not satisfy ME3. To see why, we need to consider an *unreachable* state. It is easy to construct one: let process 0 be at label `@cs` and process 1 at the start of the `while` loop. Also imagine that in this particular state `turn` = 1. Now let process process 1 make a sequence of steps. Because `turn` = 1, process 1 will break out of the while loop and also enter the critical section, violating mutual exclusion. So mutual exclusion is an invariant, but not an inductive invariant. Doing a inductive proof with an invariant that is not inductive is usually much harder than doing one with an invariant that is.

Let $C(\texttt{self}) = \texttt{flags}[1-\texttt{self}] \wedge \texttt{turn} = 1-\texttt{self}$, that is, the condition on the `while` loop. For Peterson's Algorithm, an inductive invariant that works well is: if process i is label `@cs` (i.e., process i is in the critical

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        turn = 1 - self;
        while flags[1 - self] and (turn == (1 - self)):
            pass;
        ;

        # critical section is here
        @cs: assert atLabel.cs == dict{ nametag(): 1 }, atLabel.cs;

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = 0;
spawn process(0);
spawn process(1);
```

Figure 8.1: Peterson's Algorithm

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        @gate: turn = 1 - self;
        while flags[1 - self] and (turn == (1 - self)):
            pass;
        ;

        # Critical section
        @cs: assert (not (flags[1 - self] and (turn == (1 - self))))
                    or (atLabel.gate == dict{nametags[1 - self] : 1})
             ;

        # Leave critical section
        flags[self] = False;
    ;
;
flags = [ False, False ];
turn = 0;
nametags = [ dict{ .name: .process, .tag: tag } for tag in 0..1 ];
spawn process(0), 0;
spawn process(1), 1;
```

Figure 8.2: Peterson's Algorithm with Inductive Invariant

section), then $C(i)$ does not hold or process $1-i$ is executing after setting `flags`$[i]$ but still before setting `turn` to $1-i$. Let's call it $I_p(i)$.

Figure 8.2 formalizes $I_p(i)$ in CXL. The label `@gate` refers to the instruction that sets `turn` to $1-i$. You can run Figure 8.2 to determine that $I_p(i)$ is indeed an invariant for $i = 0, 1$.

To see that $I_p(i)$ implies mutual exclusion, suppose not. Then when both process 0 and process 1 are in the critical section, the following must hold: $(\neg C(0) \vee \texttt{process}(1)@gate) \wedge (\neg C(1) \vee \texttt{process}(0)@gate)$. We know that both flags are set. We also know that neither process 0 nor process 1 is at label `@gate` (because they are both at label `@cs`), so this simplifies to $(\neg C(0)) \wedge (\neg C(1))$. So we conclude that $\texttt{turn} = 0 \wedge \texttt{turn} = 1$, a logical impossibility. Thus $I_p(i)$ satisfies ME1.

$I_p(i)$ satisfies ME2 trivially, because in the initial state no process is in the critical section.

Finally we have to show that $I_p(i)$ satisfies ME3. Without loss of generality, suppose $i = 0$ (a benefit from the fact that the algorithm is symmetric for both processes). We have to show that if we are in a state in which $I_p(0)$ holds, then any step will result in a state in which $I_p(0)$ continues to hold. Since $I_p(0)$ holds, process 0 is at label `@cs`. If process 0 were to take a step, then in the next state process 0 would be no longer at that label and $I_p(0)$ would hold trivially over the next state. Therefore we only need to consider a step by process 1.

From $I_p(0)$ we know that one of the following cases must hold before process 1 takes a step:

- `flags[1] = False`;
- `turn = 0`;
- process 1 is at label `@gate`.

Let us consider each of these cases. In the first case, if process 1 takes a step, there are two possibilities: either $flags[1]$ will still be `False` (in which case the first case continues to hold), or $flags[1]$ will be `True` and process 1 will be at label `@gate` (in which case the third case will hold). We know that process 1 never sets `turn` to 1, so if the second case holds before the step, it will also hold after the step. Finally, if process 1 is at label `@gate` before the step, then after the step `turn` will equal 0, and therefore the second case will hold after the step.

We have now demonstrated mutual exclusion in Peterson's Algorithm in two different ways: one by letting CXL explore all possible executions, the other using an inductive invariant and proof by induction. The former is certainly easier, but it does not provide intuition for why the algorithm works. The second provides much more insight. We therefore encourage to include inductive invariants in your CXL code.

A cool anecdote is the following. When the author of CXL had to teach Peterson's Algorithm, he refreshed his memory by looking at the Wikipedia page. The page claimed that the following predicate was invariant: if process i is in the critical section, then $\neg C(i)$ (i.e., $I_p(i)$ without the disjunct that process $1-i$ is at label `gate`). To demonstrate that this predicate is not invariant, you can remove the disjunct from Figure 8.2 and run it to get a counterexample.

This anecdote suggests the following. If you need to do an induction proof of an algorithm, you have to come up with an inductive invariant. Before trying to prove the algorithm, you can check that the predicate is at least invariant by testing it using CXL. Doing so could potentially avoid wasting your time on a proof that will not work because the predicate is not invariant, and therefore not an inductive invariant either.

To finish the story, the author of CXL fixed the Wikipedia page, but there is another cool story. Years later a colleague of the author, teaching the same course, asked if the first two assignments (setting `flags[self]`) to `True` and `turn` to `1 - self`) can be reversed. After all, they are different variables assigned independent values—in a sequential program one could surely swap the two assignments.

What do you think?

- See if you can figure out for yourself if the two assignments can be reversed. Then run the program in Figure 8.1 after reversing the two assignments and see what happens.

Bonus question!

- Can you generalize Peterson's algorithm to more than two processes?

Chapter 9

CXL Methods and Pointers

A method `m` with argument `a` is invoked in its most basic form as follows (assigning the result to `r`).

```
r = m a;
```

That's right, no parentheses are required. In fact, if you invoke `m(a)`, the argument is `(a)`, which is the same as `a`. If you invoke `m()`, the argument is `()`, which is the empty tuple. If you invoke `m(a, b)`, the argument is `(a, b)`, the tuple consisting of values `a` and `b`. You have already seen examples of this: the expression `atLabel.cs` invokes method `atLabel` with argument `.cs`.

You may note that all this looks familiar. Indeed, the syntax is the same as that for dictionaries (see Chapter 4). Both dictionaries and methods map CXL values to CXL values, and their syntax is indistinguishable. If `f` is either a method or a dictionary, and `x` is an arbitrary CXL value, then `f x`, `f(x)`, and `f[x]` are all the same expression in CXL.

CXL is not an object-oriented language like Python is. In Python you can pass a reference to an object to a method, and that method can then update the object. In CXL, it is also sometimes convenient to have a method update a shared variable specified as an argument. For this (as well as some other uses), CXL supports *pointers* to shared variables. If `x` is a shared variable, then the expression `&(x)` (the parentheses are mandatory) is a pointer to `x` (also known as the *address* of `x`. Conversely, if `p` is a pointer to a shared variable, then the expression `^p` is the value of the shared variable.

It is often the case that methods that update shared variables through pointers do not actually have to return a result to be useful. In CXL, if you want to invoke a method that does not return a result or if you are simply not interested in it, you have to use a `call` statement like so:

```
call m a;
```

Figure 9.1 again shows Peterson's algorithm, but this time with methods defined to enter and exit the critical section. You can put the first three methods in its own CXL source file and include it using the CXL `import` statement. This would make the code re-usable by other applications. For those who are extra adventurous: you can add the `P_enter` and `P_exit` methods to the `P_mutex` dictionary like so:

```
dict{ .turn: 0, .flags: [ False, False ], .enter: P_enter, .exit: P_exit }
```

That would allow you to simulate object methods.

```
def P_enter(pm, pid):
    (^pm).flags[pid] = True;
    (^pm).turn = 1 - pid;
    while (^pm).flags[1 - pid] and ((^pm).turn == (1 - pid)):
        pass;
    ;
;
def P_exit(pm, pid):
    (^pm).flags[pid] = False;
;
def P_mutex():
    result = dict{ .turn: 0, .flags: [ False, False ] };
;

#### The code above can go into its own CXL module ####

def process(self, pm):
    while choose({ False, True }):
        call P_enter(pm, self);
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        call P_exit(pm, self);
    ;
;
mutex = P_mutex();
spawn process(0, &(mutex)), 0;
spawn process(1, &(mutex)), 1;
```

Figure 9.1: Peterson's Algorithm accessed through methods.

Chapter 10

Spinlock

Figure 7.1 showed a faulty attempt at solving mutual exclusion using a lock. The problem with the implementation of the lock is that checking the lock and setting it if it is available is not *atomic*. Thus multiple processes contending for the lock can all "grab the lock" at the same time. While Peterson's algorithm gets around the problem, it is not efficient, especially if generalized to multiple processes. Instead, processors provide so-called *interlock instructions*: special machine instructions that can read memory and then write it in an indivisible step.

While the CXL machine does not have any built-in interlock instructions, it does have support for executing multiple instructions atomically. This feature is available in the CXL language through its `atomic` statement. We can use `atomic` blocks to implement a wide variety of interlock operations. For example, we could fix the program in Figure 3.1 by constructing an atomic increment operation for a counter, like so:

```
def atomic_inc(ptr):
    atomic:
        ^ptr = ^ptr + 1;
    ;
;
count = 0;
call atomic_inc &(count);
```

Many CPUs have an atomic "test-and-set" operation. Method `tas` in Figure 10.1 shows its specification. Here `s` points to a shared boolean variable and `p` to a private boolean variable, belonging to some process. The operation copies the value of the shared variable to the private variable and then sets the shared variable to `True`.

Figure 10.1 goes on how to implement mutual exclusion for a set of N processes. It uses $N+1$ variables. Variable `shared` is initialized to `False` while `private[`i`]` for each process i is initialized to `True`. An important invariant, I_1, of the program is that at any time at most one of these variables is `False`. Another invariant, I_2, is that if process i is in the critical section, then `private[`i`] == False`. Between the two, it is clear that only one process can be in the critical section at the same time.

To see that invariant I_1 is maintained, note that `^p == True` upon entry of `tas`. So there are two cases:

1. `^s` is `False` upon entry to `tas`. Then upon exit `^p == False` and `^s == True`, maintaining the invariant.

2. `^s` is `True` upon entry to `tas`. Then upon exit nothing has changed, maintaining the invariant.

Invariant I_1 is also easy to verify for exiting the critical section. Invariant I_2 is obvious as (i) process i only proceeds to the critical section if `private[`i`] == False`, and (ii) no other process modifies `private[`i`]`.

```
const N = 3;

def tas(s, p):
    atomic:
        ^p = ^s;
        ^s = True;
    ;
;
def process(self):
    while choose({ False, True }):
        # Enter critical section
        while private[self]:
            call tas(&(shared), &(private[self]));
        ;

        # Critical section
        @cs: assert (not private[self]) and
                (atLabel.cs == dict{ nametag(): 1 })
             ;

        # Leave critical section
        private[self] = True;
        shared = False;
    ;
;
shared = False;
private = [ True for i in 0..(N-1) ];
for i in 0..(N-1):
    spawn process(i), i;
;
```

Figure 10.1: Mutual Exclusion using a “spinlock” based on test-and-set.

```
const N = 3;

def checkInvariant():
    let sum = 0:
        if not shared:
            sum = 1;
        ;
        for i in 0..(N-1):
            if not private[i]:
                sum = sum + 1;
            ;
        ;
        result = sum <= 1;
    ;
;
def invariantChecker():
    while choose({ False, True }):
        assert checkInvariant();
    ;
;

# tas() and process() code eliminated to save space

shared = False;
private = [ True for i in 0..(N-1) ];
spawn invariantChecker(), ();
for i in 0..(N-1):
    spawn process(i), i;
;
```

Figure 10.2: Checking invariants.

CXL can check these invariants as well. Figure 10.1 already has the code to check I_2. But how would one go about checking an invariant like I_1. Invariants must hold for every state. For I_2 we only need an assertion at label `@cs` because the premisse is that there is a process at that label. However, we would like to check I_1 in *every state* (after the variables have been initialized).

We can do this by adding another process that, in a loop, checks the invariant. Figure 10.2 shows the code. Method `checkInvariant()` checks to see if the invariant holds in a state. It introduces a new feature of CXL: the ability to have variables local to a method. In this case, the variable `sum` is used to compute the number of variables that have value `False`. The function is invoked by in a loop by a process that runs alongside the other processes. In CXL, `assert` statements are executed atomically, so the evaluation of the assertion is not interleaved with the execution of other processes. Because CXL tries every possible execution, the process is guaranteed to find violations of the invariant if it does not hold.

Try doing mutual exclusion with other interlock instructions!

- Find a description for `compare-and-swap` on the internet and update Figure 10.1, replacing `tas`.
- Other interesting interlock instructions include `swap` and `fetch-and-add`.

Chapter 11

Locks and the Synch Module

In Figure 10.1 we have shown a solution based on a shared variable and a private variable for each process. The private variables themselves are implemented as shared variables as well, but they are accessed only by their respective processes. There is no need to keep `private` as a shared variable—we only did so to be able to show and check the invariants. Figure 11.1 shows a more straightforward implementation. The solution is similar to the naïve solution of Figure 7.1, but uses test-and-set to check and set the lock variable atomically. Compared to Peterson's Algorithm, this approach is general for any number of processes.

It is important to appreciate the difference between an *atomic section* (the statements executed within an `atomic` statement) and a *critical section* (protected by a lock of some sort). The former ensures that while the atomic statement is executing no other process can execute. The latter allows multiple processes to run concurrently, just not within the critical section. The former is rarely available to a programmer, while the latter is very common.

In CXL, atomic statements allow you to *implement* your own low synchronization primitives like test-and-set. Atomic statements are not intended to *replace* locks or other synchonization primitives.

Locks are probably the most popular and basic form of synchronization in concurrent programs. For this reason, CXL has a module called `synch` that includes support for locks. Figure 11.2 shows how they are implemented, and Figure 11.3 gives an example of how they may be used, in this case to fix the program of Figure 3.1. Notice that the module completely hides the implementation of the lock. The `synch` module includes a variety of other useful synchronization primitives, which will be discussed in later chapters.

```
def tas(s):
    atomic:
        result = ^s;
        ^s = True;
    ;
;
def process():
    while choose({ False, True }):
        while tas &(lock):
            pass;
        ;
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        lock = False;
    ;
;
lock = False;
for i in 1..10:
    spawn process();
;
```

Figure 11.1: Fixed version of Figure 7.1 using test-and-set.

```
def tas(lk):
    atomic:
        result = ^lk;
        ^lk = True;
    ;
;
def lock(lk):
    while tas(lk):
        pass;
    ;
;
def unlock(lk):
    ^lk = False;
;
def Lock():
    result = False;
;
```

Figure 11.2: The `Lock` interface in the `synch` module.

```
import synch;

def process(self):
    call lock &(countlock);
    count = count + 1;
    call unlock &(countlock);
    done[self] = True;
;
def main(self):
    while not (done[0] and done[1]):
        pass;
    ;
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0), 0;
spawn process(1), 1;
spawn main();
```

Figure 11.3: Program of Figure 3.1 fixed with a lock.

Chapter 12

Reader/Writer Locks

Locks are useful when accessing a shared data structure. By preventing more than one process from accessing the data structure at the same time, conflicting accesses are avoided. However, not all concurrent accesses conflict, and opportunities for concurrency may be lost, hurting performance. One important case is when multiple processes are simply reading the data structure. In many applications, reads are the majority of all accesses. Allowing reads to proceed concurrently can significantly improve performance.

What we want is a special kind of lock that allows either one writer or one or more readers to be in the critical section. This is called a *reader/writer lock* [**?**]. We will explore various ways of implementing reader/writer locks in this chapter and future ones.

Figure 12.1 presents a solution that uses a single (ordinary) lock and two counters: one that maintains the number of readers and one that maintains the number of writers. The lock is used to protect access to those counters. The program shows a process that executes in a loop. Each time, it decides whether to read or write. The critical section is spread between two labels: readers access `@rcs` and writers access `@wcs`. The specification is that if a reader is at label `@rcs`, no writer is allowed to be at label `@wcs`. Vice versa, if a writer is at label `@wcs`, no reader is allowed to be at label `@rcs` *and* there cannot be another writer at label `@wcs`. Figure 12.2 shows a high-level specification for two processes.

- Draw additional states and steps in Figure 12.2 to for three processes.

A process that wants to read first waits until there are no writers: `nwriters == 0`. If so, it increments the number of readers. Similarly, a process that wants to write waits until there are no readers *or* writers. If so, it increments the number of writers. The important invariants in this code are:

- n readers at `@rcs` $\Rightarrow$ `nreaders` $\geq n$,
- n writers at `@wcs` $\Rightarrow$ `nwriters` $\geq n$,
- (`nreaders` $\geq 0 \wedge$ `nwriters` $= 0) \vee$ (`nreaders` $= 0 \wedge 0 \leq$ `nwriters` ≤ 1).

It is easy to see that the invariants hold and imply the reader/writer specification. The solution also supports progress: if no process is in the critical section then any process can enter. Better still: if any reader is in the critical section, any other reader is also able to enter.

While correct, it is not considered a good solution. The solution is an example of what is called *busy-waiting*: processes spin in a loop until some desirable condition is met. The astute reader might wonder if obtaining an ordinary lock itself is an example of busy-waiting. After all, the CXL `synch` implementation of `lock()` spins in a loop until the lock is available (see Figure 11.2).

In most operating systems and programming language runtimes, however, when a process acquires a lock, the process is placed on a scheduling queue and stops using CPU cycles until the lock becomes available. Moreover, if there are multiple processes waiting for a lock, there is a scheduler that decides which of the processes gets the lock. Busy waiting disables all this.

```
import synch;

def process():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            let blocked = True:
                while blocked:
                    call lock &(rwlock);
                    if nwriters == 0:
                        nreaders = nreaders + 1;
                        blocked = False;
                    ;
                    call unlock &(rwlock);
                ;
            ;
            @rcs: assert atLabel.wcs == dict{}
                  ;
            call lock &(rwlock);
            nreaders = nreaders - 1;
            call unlock &(rwlock);
        else:                       # .write
            let blocked = True:
                while blocked:
                    call lock &(rwlock);
                    if (nreaders == 0) and (nwriters == 0):
                        nwriters = 1;
                        blocked = False;
                    ;
                    call unlock &(rwlock);
                ;
            ;
            @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                         (atLabel.rcs == dict{})
                  ;
            call lock &(rwlock);
            nwriters = 0;
            call unlock &(rwlock);
        ;
    ;
;
rwlock = Lock();
nreaders = 0; nwriters = 0;
for i in 1..4:
    spawn process();
;
```

Figure 12.1: Busy-Waiting Reader/Writer Lock implementation.

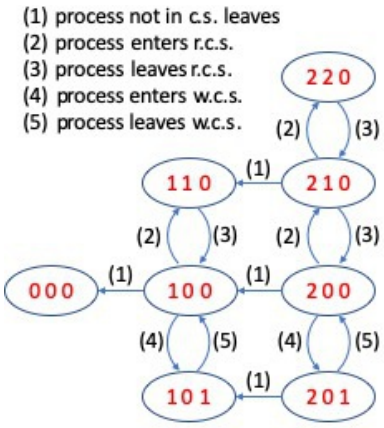

Figure 12.2: High-level state diagram specification of reader/writer locks with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes reading in the critical section; the third is the number of processes writing in the critical section.

But even when multiple processes each run on their own core, the solution is inefficient. After all, when the lock becomes available, the process has to check other variables to see if it can continue. In our case, the process may have to read `nreaders` and/or `nwriters` to decide if it can go on or has to try again, involving releasing the lock and re-acquiring it.

So it is considered ok to have processes using spinlock a implement a lock, but not to busy-wait for application-specific conditions.

Figure 12.3 presents a reader/writer lock implementation that does not busy-wait. It uses two locks: `rwlock` is used by both readers and writers, while `rlock` is only used by readers. `rwlock` is held either when readers are at label `@rcs` or when a writer is at label `@wcs`. `rlock` is used to protect the `nreaders` variable that counts the number of readers in the critical section.

The invariants that imply the reader/writer specification (which are again easy to verify) are as follows:

- n readers at `@rcs` $\Rightarrow$ `nreaders` $\geq n$,
- $\exists$ writer at `@wcs` $\Rightarrow$ `nreaders` $= 0$,

A writer simply acquires `rwlock` to enter the critical section and releases it to exit. The *first* reader to enter the critical section acquires `rwlock` and the *last* reader to exit the critical section releases `rwlock`. The implementation satisfies progress: if no process is in the critical section than any process enter.

It is instructive to see what happens when a writer is in the critical section and two readers try to enter. The first reader successfully acquires `rlock` and but *hangs* when trying to acquire `rwlock`, which is held by the writer. The second reader hangs trying acquire `rlock` because it is held by the first reader. When the writer leaves the critical section, the first reader acquires `rwlock`, sets `nreaders` to 1, and releases `rlock`. `rwlock` is still held. Then the second reader acquires `rlock` and, assuming the first reader is still in the critical section, increments `nreaders` to 2 and enter the critical section *without* acquiring `rwlock`. `rwlock` is essentially jointly held by both readers. It does not matter in which order they leave: the second will release `rwlock`.

```
import synch;

def process():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            call lock &(rlock);
            if nreaders == 0:
                call lock &(rwlock);
            ;
            nreaders = nreaders + 1;
            call unlock &(rlock);

            @rcs: assert atLabel.wcs == dict{}
                  ;
            call lock &(rlock);
            nreaders = nreaders - 1;
            if nreaders == 0:
                call unlock &(rwlock);
            ;
            call unlock &(rlock);
        else:                       # .write
            call lock &(rwlock);

            @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                         (atLabel.rcs == dict{})
                  ;
            call unlock &(rwlock);
        ;
    ;
;
rwlock = Lock();
rlock = Lock();
nreaders = 0;
for i in 1..4:
    spawn process();
;
```

Figure 12.3: Reader/Writer with Two Locks.

Chapter 13

Semaphores

So far we have looked at how to protect a single resource. Sometimes we have multiple but a limited amount of resources that require protection. That is, if there are only n resources, no more than n can be used at a time; if all are in use and a process come along that needs one of the resources, it has to wait until another process releases one of the resources. Note that we cannot solve this problem simply using a lock per resource—allocating a resource requires access to all of them.

A good example is the so-called *bounded buffer*. It is essentially a queue implemented using a circular buffer of a certain length and two pointers: one where items are inserted and one from where items are extracted. If the buffer is full, processes that want to add more items have to wait. In the usual formulation, processes that want to extract an item when the buffer is empty also have to wait. It would be easy to build such a thing using a busy waiting approach, but again, we would like to avoid busy waiting.

Introduced by the famous Dutch computer scientist Edsger Dijkstra, a *semaphore* is a primitive that fits the bill. A semaphore is essentially a counter that can be incremented and decremented but is not allowed to go below zero. The semaphore counter is typically initialized to the number of resources available. When allocating a resource, a process decrements the counter using the `procure` or simply `P` operation. The `procure` operation blocks the invoking process if the counter is zero. To release the resource, a process increments the resource using the `vacate` or V operation. Figure 13.1 shows the `synch` module implementation of semaphores.

```
def Semaphore(cnt):
    result = cnt;
;
def P(sema):
    let blocked = True:
        while blocked:
            atomic:
                if (^sema) > 0:
                    ^sema = (^sema) - 1;
                    blocked = False;
                ;
            ;
        ;
    ;
;
def V(sema):
    atomic:
        ^sema = (^sema) + 1;
    ;
;
```

Figure 13.1: The `Semaphore` interface in the `synch` module.

```
import synch;

const NSLOTS = 2;     # size of bounded buffer
const NPROCS = 3;     # number of producers and number of consumers

def produce(item):
    call P &(n_empty);
    call P &(mutex);
    buf[b_in] = item;
    b_in = (b_in % NSLOTS) + 1;
    call V &(mutex);
    call V &(n_full);
;
def consume():
    call P &(n_full);
    call P &(mutex);
    result = buf[b_out];
    b_out = (b_out % NSLOTS) + 1;
    call V &(mutex);
    call V &(n_empty);
;
buf = dict{ () for x in 1..NSLOTS };
b_in = 1; b_out = 1;
n_full = Semaphore(0);
n_empty = Semaphore(NSLOTS);
mutex = Semaphore(1);
for i in 1..NPROCS:
    spawn consume();
    spawn produce();
;
```

Figure 13.2: Bounded Buffer implementation using semaphores.

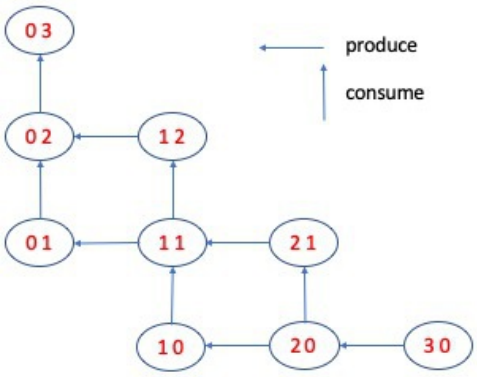

Figure 13.3: High-level state diagram specification of producers/consumers.

Chapter 14

Starvation

So far we have pursued two properties: *mutual exclusion* and *progress*. The former is an example of a *safety property*—it prevents something "bad" from happening, like a reader and writer process both entering the critical section. The *progress* property is an example of a *liveness property*—guaranteeing that something good eventually happens. Informally (and inexactly), progress states that if no processes are in the critical section, then some process that wants to enter can.

Progress is a weak form of liveness. It says that *some* process can enter, but it does not prevent a scenario such as the following. There are three processes repeatedly trying to enter a critical section using a spinlock. Two of the processes successfully enter, alternating, but the third process never gets a turn. This is an example of `starvation`. With a spinlock, this scenario could even happen with two processes. Initially both processes try to acquire the spinlock. One of the processes is successful and enters. After the process leaves, it immediately tries to re-enter. This state is identical to the initial state, and there is nothing that prevents the same process from acquiring the lock yet again.

It is worth noting that Peterson's Algorithm (Chapter 8) does not suffer from starvation, thanks to the `turn` variable that alternates between 0 and 1 when two processes are contending for the critical section.

While spinlocks suffer from starvation, it is a uniform random process and each process has an equal chance of entering the critical section. Thus the probability of starvation is exponentially vanishing. Unfortunately, such is not the case for the reader/writer solution of Figure 12.3. Consider this scenario: there are two readers and one writer. One reader is in the critical section while the writer is waiting. Now the second reader tries to enter and is able to. The first reader leaves. We are now in a similar situation as the initial state with one reader in the critical section and the writer waiting, but it is not the same reader. Unfortunately for the writer, this scenario can repeat itself indefinitely. So even if neither reader was in the critical section all of the time, and the second reader arrived well after the writer, the writer never had a chance. In this chapter, we will present a version of a reader/writer lock implementation that solves this type of starvation, but won't eliminate the type of starvation that comes with spinlocks.