Concurrent Programming with CXL

RVR and possibly others

June 14, 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 are concurrent programs more complicated than sequential ones? 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 output will be produced. Bugs are typically easily reproducible and easy to track down, for example by instrumenting the program. The output of running concurrent programs depends on how the various processes are interleaved. Some bugs may occur only occasionally and may never occur when the program is instrumented to find them (so-called *Heisenbugs*).
- In a sequential program, each statement and each function can be thought of as happening *atomically* because there is no other activity interfering with their execution. Even though a statement or function may be compiled into multiple machine instructions, they are executed back-to-back until completion. Not so with a concurrent program, where other instructions may update memory locations as a statement or function is being executed.

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 helps 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 if model checking does not prove a program correct, why is it useful? To answer that question 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 for sure 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. Note, however, that it would be easy to write a program that sorts all lists of up to five numbers correctly but fails to do so for a list of 6 numbers. (Hint: simply use an `if` statement.)

While model checking does not in general prove an algorithm correct, it can 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 proposed invariants when evaluating a model and provide valuable early feedback to somebody who is trying to construct a proof, even an informal one. Throughout this book we will include 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 test 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, stack, and machine instructions.

CXL is heavily influenced by Leslie Lamport's work on TLA+, TLC, and PlusCal []. CXL is designed to have a lower learning curve than those systems, but is not as powerful. When you finish this book and want to learn more, we strongly encourage checking those out. Another excellent resource is Fred Schneider's book "On Concurrent Programming."

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:

- Syntax: Every statement in CXL must be terminated by a semicolon (even `while` statements and `def` statements). In Python semicolons are optional and rarely used. There is no syntactic significance to indentation in CXL. Correct indentation in CXL is encouraged but not enforced. CXL only supports basic operator precedence or associativity. Use parentheses liberally to remove ambiguity.
- CXL does not (currently) support floating point. CXL does not support I/O;
- Python is object-oriented; CXL is not;

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 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. The first assigns to `x` some unspecified value in the range `0..N` and the second verifies that `triangle`(x) equals $x(x+1)/2$.

```
1  const N = 10;
2
3  def triangle(n):   # computes the n'th triangle number
4      result = 0;
5      for i in 1..n:
6          result = result + i;
7      ;
8  ;
9  x = choose(0..N);
10 assert triangle(x) == ((x * (x + 1)) / 2);
```

Figure 2.1: Computing triangle numbers.

"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
$
```

(For this to work, make sure `cxl` is in your command shell's search path.) 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.

In CXL, constants have a default, but those can be overridden on the command line using the `-c` option. For example, if you want to test the code for `N = 100`, run:

```
$ cxl -c N=100 triangle.cxl
#states = 103
no issues found
$
```

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 machine 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():
    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 `done[0]` to `True`;
6. incrementer 1 sets `done[1]` 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 manipulate 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. A dictionary maps values, known as *keys*, to values. The CXL dictionary syntax and properties are a little different from Python. 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 to Python programers, but in CXL square brackets can be used in the same way as parentheses, so you can express the same thing in the form that is familiar to Python programmers:

```
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). Thus using atoms, a dictionary can be made to look much like a Python object. 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 including a comma. 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 computers 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 the first argument to the method. 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 dictionary, `dict{}`. 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 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 a directory 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, and it is relative to the "root" of the directory tree.

Compiling the code in Figure 3.1 results in the CXL machine code listed in Figure 5.1. You can obtain this code by invoking `cxl` with the `-a` flag like so:

```
cxl -a Up.cxl
```

The CXL machine is predominantly a *stack machine*. All instructions are atomically executed. Most instructions pop values from the stack or push values onto the stack. At first there is one process (with name

```
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.

tag `__init__/()` that initializes the state. It starts executing at instruction 0 and keeps executing until the last execution in the program. In this case, the first instruction is a `JUMP` instruction that sets the program counter to 39. At program counter 1 is the code for the `incrementer` method. All methods start with a `Frame` instruction and end with a `Return` instruction.

The `Frame` instruction has three arguments: the name of the method, a list containing the names of the arguments of the method, and 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.

Once initialization completes, any processes that were spawned can run. You can think of CXL as trying every possible interleaving of processes executing instructions. Figure 5.2 shows the output produced by running CXL on the `Up.cxl` program. It starts by reporting that the assertion on line 9 in column 5 failed and that `count` has value 1 instead of 0: `Assertion failed ('assert', 'Up.cxl', 9, 5) 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[1]`;
3. Process 0 continues execution, storing value 1 into `count` and storing `True` into `done[0]`.

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, no other process is trying to do the same thing. We call this a *critical section* (aka critical region): 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 counter is a very simple example of a data structure, but as we have seen it too requires multiple instructions. 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. Figure 6.1 shows the CXL code. Here `@cs` is a *label*, 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 containing the name of the label (in this case, `.cs`). The operator returns a bag (multiset) 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. 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.

```
1  def process(self):
2      while True:
3          #enter critical section
4          @cs: assert atLabel.cs == dict{ nametag(): 1 };
5          #exit critical section
6      ;
7  ;
8  spawn process(0);
9  spawn process(1);
```

Figure 6.1: A barebones critical section with two processes.

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

Figure 6.2: CXL model of a 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.

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 show a high-level state diagram of the code in Figure 6.2. The circles represent states, while the arrows represent possible state transitions. The label in each circle summarizes the state; the label on an arrow describes the transition. In this case, the label contains two numbers: the total number of processes and the number of processes in the critical section. A process that is in the critical section cannot terminate.

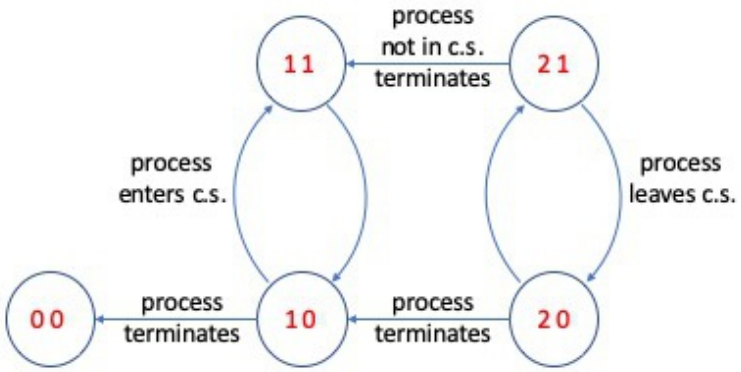

Figure 6.3: High-level state diagram specification of mutual exclusion with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes in the critical section.

Chapter 7

Naïve Attempts at Implementing Critical Sections

You may already have heard of the concept of a *lock* and have realized that it could be used to implement a critical section. The idea is that the lock is like a baton that at most one process can own (or hold) at a time. A process that wants to enter the critical section at a time must obtain the lock first and release it upon exiting the critical section. Using a lock is a good thought, but how does one implement one? Figure 7.1 presents a mutual exclusion attempt based on a naïve (and, as it turns out, broken) implementation of a lock. 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 through CXL, 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.

Run the program through CXL. Unfortunately, it turns out the solution has a problem: 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` $== i$. 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.

```
def process(self):
    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);
spawn process(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);
spawn process(1);
```

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

Run the program through CXL. It turns out that this solution also violates *progress*, albeit for a different reason: if process i terminates instead of entering the critical section when it is process i's turn, process $1 - i$ ends up waiting indefinitely for its turn.

Try your own solutions!

- See if you can come up with some different approaches that satisfy both mutual exclusion and progress. Try them with CXL and see if they work or not. If they don't, try to understand why. Do not despair if you can't figure it out—as we will find out, it is possible but not easy.

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 nowhere near the critical section (`flag[1 - self] == False`) or or has given way (`turn == self`). 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. A state is comprised of the values of the shared variables as well as the values of the process variables for each process, including its program counter and the contents of its stack. Everytime a process executes a CXL machine instruction, the state changes (if only because the program counter of the process changes). 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 there is more than one process that can make a step. The other is when a process executes a `choose` operation and there is more than one choice. Because there is non-determinism, there are multiple possible traces. We call a state *reachable* if it is either the initial state or it can be reached from the initial state through a trace. A state is final when there are no processes left to make state changes.

A *property* describes a subset of states. In other words, a property is equivalent to a subset of the states. *Initial*, *final*, and *reachable*, and *unreachable* are all examples of properties. Figure 8.2 depicts a Venn diagram of various properties and a trace. An *invariant* is a property that holds over all reachable states. We want to show that mutual exclusion is an invariant. In other words, we want to show that the set of reachable states is a subset of the set of states where there is at most one process in the critical section.

One way to prove that a property is an invariant is through induction on the number of steps. First you prove that the property holds over the initial state. Then you prove that for every reachable state, and for every step from that reachable state, the property also holds over the resulting state. For this you need a predicate that describes exactly which states are reachable. But we do not have such a predicate: we know how to describe the set of reachable states, but given an arbitrary state it is not easy to see whether it is reachable or not.

To solve this problem, we will use what is called an *inductive invariant*. An inductive invariant I is a predicate over states that satisfies the following:

- I holds in the initial state.

```
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

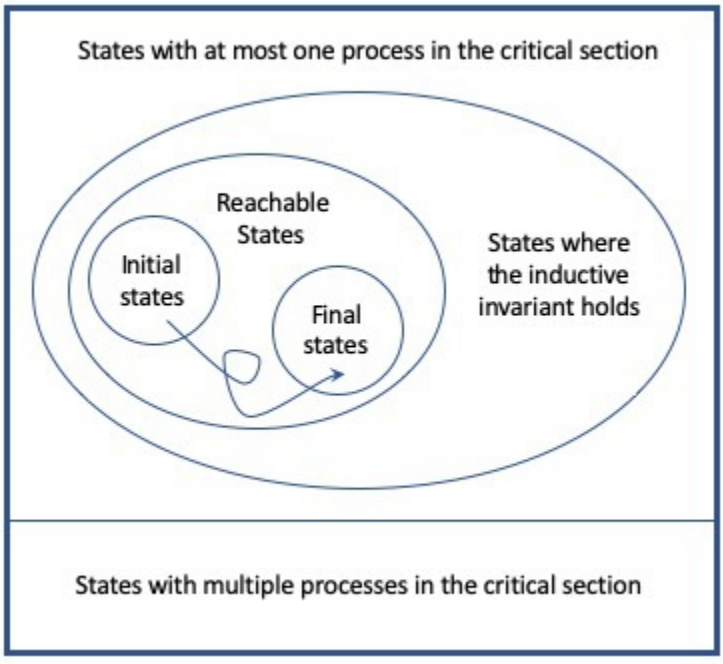

Figure 8.2: Venn diagram of all states and a trace.

```
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 8.3: Peterson's Algorithm with Inductive Invariant

- For any state in which I holds and any process in the state takes a step, then I also holds in the resulting state.

Unlike an invariant, an inductive invariant must also hold over unreachable states.

One candidate for such an invariant is mutual exclusion itself. After all, it certainly holds over the initial state. And as CXL has already determined, mutual exclusion is an invariant: it holds over every *reachable state*. Unfortunately, it is not an *inductive* invariant. 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.

So we are looking for an inductive invariant that *implies* mutual exclusion. In other words, the set of states where the inducive invariant holds must be a subset of the set of states where there is at most one process in the critical section. Let $C(i) =$ `flags`$[1-i] \wedge$ `turn` $= 1-i$, that is, the condition on the `while` loop for process i. Let predicate $I_p(i)$ be the following. if process i is label `@cs` (i.e., process i is in the critical section), then $C(i)$ does not hold or process $1-i$ is executing after setting `flags`$[i]$ but still before setting `turn` to $1-i$. More formally, $I_p(i) =$ `process`$(i)@cs) \Rightarrow (\neg C(i) \vee$ `process`$(1-i)@gate)$. For Peterson's Algorithm, an inductive invariant that works well is $I_p(0) \wedge I_p(1)$.

Figure 8.3 formalizes $I_p(i)$ in CXL. The label `@gate` refers to the instruction that sets `turn` to $1-i$. You can run Figure 8.3 to determine that $I_p(i)$ is indeed an invariant for $i = 0, 1$.

To see that the inductive invariant implies mutual exclusion, suppose not. Then when both process 0 and process 1 are in the critical section, the following must hold: $(\neg C(0) \vee$ `process`$(1)@gate) \wedge (\neg C(1) \vee$ `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 $I_p(i)$ implies mutual exclusion.

To see that $I_p(0) \wedge I_p(1)$ is, in fact, an inductive invariant, first note that it certainly holds in the initial state, because in the initial state no process is in the critical section. Without loss of generality, suppose $i = 0$ (a benefit from the fact that the algorithm is symmetric for both processes). We still 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 $I_p(0)$ still holds. If $I_p(0)$ 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 three cases must hold before process 1 takes a step:

1. `flags[1] = False`;
2. `turn = 0`;
3. 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. qed.

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 is 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 8.3 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 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):
    while choose({ False, True }):
        call P_enter(&(mutex), self);
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        call P_exit(&(mutex), self);
    ;
;
mutex = P_mutex();
spawn process(0);
spawn process(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, multi-core 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 in two ways. First, any CXL statement can be made atomic by placing a label in front of it. Second, a group of CXL statements can be made atomic 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 (the "test") and then sets the shared variable to `True` ("set").

Figure 10.1 goes on how to implement mutual exclusion for a set of N processes. It is called *spinlock*, because each process is "spinning" until it can acquire the lock. The program 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, $I_2(i)$, is that if process i is in the critical section, then `private[`i`] == False`. Between the two (i.e., $I_1 \wedge \forall i : I_2(i)$), it is clear that only one process can be in the critical section at the same time.

I_1 is an inductive invariant. 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.

```
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 omitted to save space

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

Figure 10.2: Checking invariants.

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

CXL can check these invariants as well. Figure 10.1 already has the code to check $I_2(i)$. 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 process variable `sum` is used to compute the number of shared 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 actually implemented as shared variables, 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 of spinlock. 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. 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.

We call a process *blocked* in a state if the process is in an *infinite loop* in which the process does not update the shared memory. A process trying to acquire a test-and-set spinlock held by another process is a good example of a process being blocked: the process only reads the shared memory but does not change it. The only way to break out of the infinite loop is if another process changes the shared memory. The `while` loop in Peterson's algorithm, in case its condition does not hold, is another example of a process being blocked.

In most operating systems, processes are virtual and can be suspended until some condition changes. For example, a process can be suspended until the lock is available to it. In CXL, a process can suspend itself and save its context in a list. Recall that the context of a process consists of its name tag, its program counter, and the contents of its register and stack. A context is a regular CXL value. The syntax of the expression is as follows:

```
stop L
```

Here `L` is a shared list. Another process can revive the process using the `go` stateent:

```
go C R
```

```
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():
    result = False;
;
def lock(lk):
    while tas(lk):
        pass;
    ;
;
def unlock(lk):
    ^lk = 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);
spawn process(1);
spawn main();
```

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

Here `C` is a context and `R` is a CXL value. It causes a process with context `C` to be added to the state that has just executed the `stop` expression. The `stop` expression returns the value `R`.

There is a second version of the `synch` module that uses suspension instead of spinlocks. Figure 11.4 shows the same `Lock` interface implemented using suspension. A `Lock` maintains both a boolean indicating whether the lock is taken, and a list of contexts of processes that want to acquire the lock. `lock` sets the lock if available, or suspends itself if not. Note that `stop` is called within an `atomic` statement—this is the only exception to an atomic statement running to completion. While the process is running no other processes can run, but when it suspends itself it allows other processes to run.

`unlock` checks to see if any processes are waiting to get the lock. If so, it uses the `head` and `tail` methods from the `list` module to resume the first process that got suspend and to remove its context from the list. Selecting the first process is a design choice. Another implementation could have picked the last one, and yet another implementation could have used `choose` to pick an arbitrary one. Selecting the first is a common choice in lock implementations as it prevents *starvation*: every process gets a chance to obtain the lock (assuming every process eventually releases it). Chapter 18 will talk more about starvation.

CXL allows you to select which version of the `synch` module you would like to use with the `-m` flag. For example,

```
cxl -m synch=synchS x.cxl
```

runs the file `x.cxl` using the Suspension version of the `synch` module. You will find that using the `synchS` module often leads to searching a significantly larger search space than using the `synch` module. Part of the reason is that the `synchS` module keeps track of the order in which processes wait for a lock.

```
import list;

def Lock():
    result = dict{ .locked: False, .suspended: [] };
;
def lock(lk):
    atomic:
        if (^lk).locked:
            call stop (^lk).suspended;
            assert (^lk).locked;
        else:
            (^lk).locked = True;
        ;
    ;
;
def unlock(lk):
    atomic:
        if (^lk).suspended == []:
            (^lk).locked = False;
        else:
            go (head((^lk).suspended)) ();
            (^lk).suspended = tail((^lk).suspended);
        ;
    ;
;
```

Figure 11.4: The `Lock` interface in the `synchS` module uses suspension.

Chapter 12

Reader/Writer Locks using Busy Waiting

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. We will call the lock the *mutex* to distinguish it clearly from the reader/writer lock that we are implementing. The mutex is used to protect shared access to the 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 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* (aka spin-waiting): processes spin in a loop until some desirable application-level condition is met. The astute reader might wonder if obtaining the mutex itself is an example of busy-waiting. After all, the CXL `synch` implementation of `lock()` spins in a loop until the mutex is available (see Figure 11.2).

```
import synch;

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

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

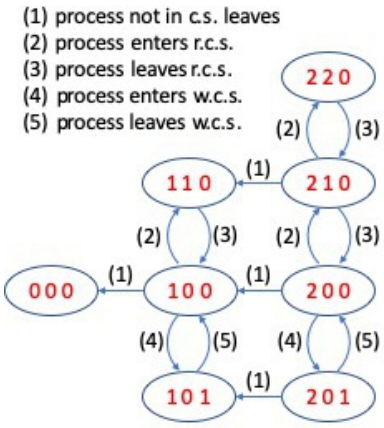

Figure 12.2: High-level state diagram specification of reader/writer locks with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes reading in the critical section; the third is the number of processes writing in the critical section.

However, there is a big difference. As we pointed out, the processes that are waiting for a spinlock are *blocked*: they are in an infinite loop in which they do not change the state. The processes waiting for a reader/writer lock do change the state while in the infinite loop: they acquire the mutex and release it.

In most operating systems and programming language runtimes, when a process acquires a lock, the process is placed on a scheduling queue and stops using CPU cycles until the lock becomes available. CXL supports this with the `synchS` module. Busy waiting disables all this: even when using the `synchS` module, readers and writers only get suspended temporarily in case there is contention for the mutex. A process trying to obtain a read or write lock in Figure 12.1 has to obtain the mutex repeatedly until the read or write lock is available.

Thus, it is considered ok to have processes be blocked while waiting for application-specific conditions, but not for them to busy-wait for application-specific conditions.

CXL does not complain about Figure 12.1, but, using the right test, CXL can be used to check for busy-waiting. Figure 12.3 presents such a test. Note that the initialization code obtains a read lock, preventing all readers and writers from entering the critical section. In that case, those processes should ideally be blocked and CXL can test for that by running it with the `-b` flag. This flag tells CXL to check that all states are non-terminating and lead to a state in which all processes are blocked.

```
import synch;

const NREADERS = 2;
const NWRITERS = 2;

def acquire_rlock():
    call lock &(rlock);
    if nreaders == 0:
        call lock &(rwlock);
    ;
    nreaders = nreaders + 1;
    call unlock &(rlock);
;
def release_rlock():
    call lock &(rlock);
    nreaders = nreaders - 1;
    if nreaders == 0:
        call unlock &(rwlock);
    ;
    call unlock &(rlock);
;
def acquire_wlock():
    call lock &(rwlock);
;
def release_wlock():
    call unlock &(rwlock);
;
def reader(self):
    call acquire_rlock();
    @rcs: assert atLabel.wcs == dict{};
    call release_rlock();
;
def writer():
    call acquire_wlock();
    @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                 (atLabel.rcs == dict{})
          ;
    call release_wlock();
;
rwlock = Lock();
rlock = Lock();
nreaders = 0;
for i in 1..NREADERS: spawn reader(i);
;
for i in 1..NWRITERS: spawn writer();
;
call acquire_rlock();
```

Figure 12.3: Checking for Busy Waiting.

Chapter 13

Reader/Writer Lock, Part 2

Figure 13.1 presents a reader/writer lock implementation that does not busy-wait. It uses two ordinary 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 blocks (!) when trying to acquire `rwlock`, which is held by the writer. The second reader blocks 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`.

While CXL does not detect any issues, we must remember what it is checking for: it checks to make sure that there are never a reader and a writer in the critical section, and there are never two readers in the critical section. Moreover, it checks progress: all executions eventually terminate. What it does *not* check is if the code allows more than one reader in the critical section. Indeed, we could have implemented the reader/writer lock as in Figure 13.2 using an ordinary lock, never allowing more than one process in the critical section.

Figure 13.2 contains a new test that ensures that indeed more than one reader can enter the critical section at a time. It does so by having processes sometimes wait in the critical section until all other readers are there. If other readers cannot enter, then that process enters in an infinite loop that CXL will readily detect. Try it out!

An important lesson here is that one should not celebrate too early if CXL does not find any issues. While a test may be successful, a test may not explore all desirable executions. While CXL can help explore all cornercases in a test, it cannot find problems that the test is not looking for.

- Why do you suppose the code in Figure 13.2 uses a flag per process rather than a simple counter?

```
import synch;

def acquire_rlock():
    call lock &(rlock);
    if nreaders == 0:
        call lock &(rwlock);
    ;
    nreaders = nreaders + 1;
    call unlock &(rlock);
;
def release_rlock():
    call lock &(rlock);
    nreaders = nreaders - 1;
    if nreaders == 0:
        call unlock &(rwlock);
    ;
    call unlock &(rlock);
;
def acquire_wlock():
    call lock &(rwlock);
;
def release_wlock():
    call unlock &(rwlock);
;
rwlock = Lock();
rlock = Lock();
nreaders = 0;

def process():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            call acquire_rlock();
            @rcs: assert atLabel.wcs == dict{};
            call release_rlock();
        else:                       # .write
            call acquire_wlock();
            @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                         (atLabel.rcs == dict{})
                  ;
            call release_wlock();
        ;
    ;
;
for i in 1..3:
    spawn process();
;
```

Figure 13.1: Reader/Writer with Two Locks.

```
import synch;

const NREADERS = 2;
const NWRITERS = 2;

def acquire_rlock():
    call lock &(rwlock);
;
def release_rlock():
    call unlock &(rwlock);
;
def acquire_wlock():
    call lock &(rwlock);
;
def release_wlock():
    call unlock &(rwlock);
;
def reader(self):
    call acquire_rlock();
    @rcs: assert atLabel.wcs == dict{};
    flags[self] = True;
    if choose({ False, True }):
        let blocked = True:
            while blocked:
                if { flags[i] for i in 1..NREADERS } == { True }:
                    blocked = False;
                ;
            ;
        ;
    ;
    call release_rlock();
;
def writer():
    call acquire_wlock();
    @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                 (atLabel.rcs == dict{})
          ;
    call release_wlock();
;
rwlock = Lock();
nreaders = 0;
flags = dict{ False for i in 1..NREADERS };
for i in 1..NREADERS:
    spawn reader(i);
;
for i in 1..NWRITERS:
    spawn writer();
;
```

Figure 13.2: Checking that multiple readers can acquire the read lock.

- Can you write a version of this test that uses a counter instead of a flag per process?
- Write a "library implementation" of reader/writer locks so multiple reader/writer locks can be instantiated.

Chapter 14

Deadlock

When multiple processes are synchronizing access to shared resources, they may end up in a *deadlock* situation where one or more of the processes end up being blocked indefinitely because each is waiting for another to give up a resource. The famous Dutch computer scientist Edsger W. Dijkstra illustrated this using a scenario he called "Dining Philosophers."

Imagine five philosopers sitting around a table, each with a plate of food in front of them, and a fork between each two plates. Each philosopher requires two forks to eat. To start eating, a philosopher first picks up the fork on the left, then the fork on the right. Each philospher likes to take breaks from eating to think for a while. To do so, the philosopher puts down both forks. The philosophers repeat this procedure. Dijkstra had them repeating this for ever, but for the purposes of this book, philosphers can leave the table when they're not eating.

Figure 14.1 implements the dining philosophers in CXL, using a process for each philosopher and a lock for each fork. If you run it, CXL complains that the execution may not terminate, with all five processes being blocked trying to acquire the lock.

- Do you see what the problem is?
- Does it depend on `N`, the number of philosophers?
- Can you come up with a solution?
- Does it matter in what order the philosophers lay down their forks?

We will discuss ways of preventing deadlock later, but one important approach to avoiding deadlock is to establish an ordering among the resources (in this case the forks) and, when needing more than one resource, always acquiring them in order. In the case of the philosopers, they could prevent deadlock by always picking up the lower numbered fork before the higher numbered fork, like so:

```
    if left < right:
        call lock &(forks[left]);
        call lock &(forks[right]);
    else:
        call unlock &(forks[right]);
        call unlock &(forks[left]);
    ;
```

- Try it out!

```
import synch;

const N = 5;

def diner(which):
    while choose({ False, True }):
        let left = which, right = (which % N) + 1:
            call lock &(forks[left]);
            call lock &(forks[right]);
            # dine
            call unlock &(forks[left]);
            call unlock &(forks[right]);
            # think
        ;
    ;
;
forks = dict{ Lock() for i in 1..N };
for i in 1..N:
    spawn diner(i);
;
```

Figure 14.1: Dining Philosophers.

Chapter 15

Semaphores

So far we have looked at how to protect a single resource. Some types of resources may have multiple units. If there are only n units, no more than n units can be used at a time; if all are in use and a process comes along that needs one of the units, it has to wait until another process releases one of the units. Note that we cannot solve this problem simply using a lock per unit—allocating a unit requires access to the entire collection.

Consider a variant of the Dining Philosophers (which were clearly Dutch—who else need two forks to eat a meal?) that we call Italian Philosophers (who only need one fork each to eat spaghetti). Again, there are five philosophers. This time, however, there is a glass sitting in the middle of the table with only three forks in it. A philosopher needs one of those forks to eat, but clearly, no more than three philosophers can eat at a time.

- See if you can come up with a deadlock-free solution using locks. Represent each fork as a lock.

It's not easy. It would be easy to come up with a solution that uses a single lock and busy waiting, but we would like to avoid busy waiting. Introduced by Dijkstra, a *semaphore* is a synchronization 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 units of a resource initially available. When allocating a resource, a process decrements the counter using the `P` operation. You can think of `P` standing for *Procure*, as in procuring the resource associated with the semaphore. The `P` operation blocks the invoking process if the counter is zero. To release the resource, a process increments the resource using the `V` operation. You can think of `V` standing for *vacate*, as in vacating the resource.

Figure 15.1 shows the `synch` module implementation of semaphores. Figure 15.2 shows a solution to the Italian Philosphers problem using a semaphore.

- Can you come up with a test (an assertion) that shows that at most three philosophers are eating at the same time?
- Can you come up with a test (different code) that checks to see that the solution allows more than one philosopher to eat at the same time?

One can think of a semaphore as a generalization of a lock. In fact, a lock can be implemented by a semaphore initialized to 1. Then `P` procures the lock, and `V` vacates the lock.

```
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 15.1: The `Semaphore` interface in the `synch` module.

```
1  import synch;
2
3  const NDINERS = 5;
4  const NFORKS = 3;
5
6  def diner(which):
7      while choose({ False, True }):
8          call P &(forks);
9          # dine
10         call V &(forks);
11         # think
12     ;
13 ;
14 forks = Semaphore(NFORKS);
15 for i in 1..NDINERS:
16     spawn diner(i);
17 ;
```

Figure 15.2: Italian Philosophers.

Chapter 16

Bounded Buffer

A good example of a resource with multiple units 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. This problem is known as the "Producer/Consumer Problem."

Figure 16.1 gives a high-level description of what we want. The two numbers in a state specify the number of items that the producers still can produce and the number of items that the consumers have consumed. In this case there are three items to produce initially.

Figure 16.2 presents the implementation of a bounded buffer using semaphores. It features a circular bounded buffer `buf` with slots numbered 1 through `NSLOTS`. There are two indexes into the buffer: `b_in` specifies where the next produced item is inserted, while `b_out` specifies from where the next can be consumed. As there may be multiple producers and multiple consumers, updates to the indexes, which are shared variables, must be protected. To this end we use two semaphores as locks: `l_in` for `b_in` and `l_out` for `b_out`.

Without additional synchronization, the indexes may overflow and point to invalid entries in the circular buffer. To this end there is a semaphore `n_full` that keeps track of the number of filled entries in the buffer and a semaphore `n_empty` that keeps track of the number of empty entries in the buffer.

To add an item to the bounded buffer (`produce(item)`), the producing process first has to wait until there is room in the bounded buffer. To this end, it invokes `P(n_empty)`, which waits until `n_empty > 0` and atomically decrements `n_empty` once this is the case. In other words, the producer procures an empty slot.

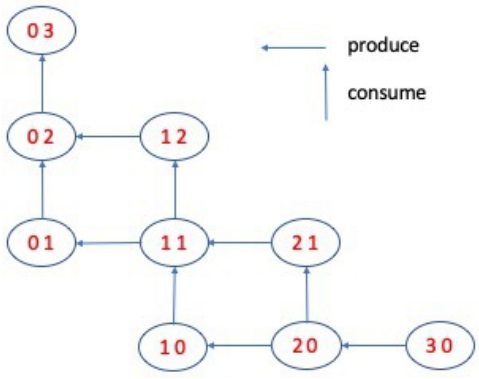

Figure 16.1: High-level state diagram specification of producers/consumers.

```
import synch;

const NSLOTS = 2;     # size of bounded buffer
const NPRODS = 3;     # number of producers
const NCONSS = 3;     # number of consumers

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

Figure 16.2: Bounded Buffer implementation using semaphores.

```
cxl -b -c NSLOTS=0 -c NPRODS=1 -c NCONSS=1 PCsema.cxl
cxl -b -c NSLOTS=1 -c NPRODS=0 -c NCONSS=1 PCsema.cxl
cxl    -c NSLOTS=1 -c NPRODS=1 -c NCONSS=0 PCsema.cxl
cxl    -c NSLOTS=1 -c NPRODS=1 -c NCONSS=1 PCsema.cxl
cxl -b -c NSLOTS=1 -c NPRODS=1 -c NCONSS=2 PCsema.cxl
cxl -b -c NSLOTS=1 -c NPRODS=2 -c NCONSS=0 PCsema.cxl
cxl    -c NSLOTS=1 -c NPRODS=2 -c NCONSS=1 PCsema.cxl
cxl    -c NSLOTS=1 -c NPRODS=2 -c NCONSS=2 PCsema.cxl
cxl -b -c NSLOTS=1 -c NPRODS=2 -c NCONSS=3 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=1 -c NCONSS=0 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=1 -c NCONSS=1 PCsema.cxl
cxl -b -c NSLOTS=2 -c NPRODS=1 -c NCONSS=2 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=2 -c NCONSS=0 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=2 -c NCONSS=1 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=2 -c NCONSS=2 PCsema.cxl
cxl -b -c NSLOTS=2 -c NPRODS=2 -c NCONSS=3 PCsema.cxl
cxl -b -c NSLOTS=2 -c NPRODS=3 -c NCONSS=0 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=3 -c NCONSS=1 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=3 -c NCONSS=2 PCsema.cxl
cxl    -c NSLOTS=2 -c NPRODS=3 -c NCONSS=3 PCsema.cxl
```

Figure 16.3: Script for testing producer/consumer synchronization (assuming the code is in file `PCsema.cxl`).

Next, it adds the item to the next position in the buffer. Since there may be more than one empty slot, multiple producers may attempt to do the same thing concurrently. Hence the use of the `l_in` semaphore used as a lock. Finally, once the item is added, the process increments the `n_full` semaphore to signal to consumers that an item is available.

The consumer code is symmetric.

Note the quite different usage of the `l_` semaphores and `n_` semaphores. People often call semaphores that are used as locks *binary semaphores*, as their counters only take on the values 0 and 1. They protect critical sections. The second type of semaphores are called *counting semaphores*. They can be used to send signals between processes. However, binary and counting semaphores are implemented the same way.

We can run the code through CXL, but what does it test? There are no assert statements in the code. We can split the functionality of this code into two pieces. One is the synchronization part: producers should never run more than `NSLOTS` ahead of the consumers, but they should be able to produce up to `NSLOTS` items even if there are no consumers. The other part is the data: we want to make sure that the items are sent through the buffer in order and without loss. We will first focus on the synchronization.

The first part can be tested with the code given in Figure 16.1, by experimenting with different non-negative values for the three constants. In particular, any run with a positive `NSLOTS` and $\texttt{NCONSS} \leq \texttt{NPRODS} \leq \texttt{NCONSS} + \texttt{NSLOTS}$ should terminate, while other values should lead to processes blocking. One could write a script like the one in Figure 16.3.

Figure 16.4 tests whether the correct data is sent in the correct order. There are the same number of producers and consumers. Each producer i produces two tuples: $(i, 0)$ and $(i, 1)$ in that order. Each consumer consumes two tuples and checks that the tuples are different and that, if the two tuples come from the same producer, then they have to be in the order sent. Moreover, consumers add the tuples they received to the set `received` (atomically—a lock could have been used here as well).) The `main()` process waits for all consumers to have received exactly the set of tuples that the producers produce.

```
import synch;

const NSLOTS = 2;       # size of bounded buffer
const NPRODS = 2;       # number of producers
const NCONSS = NPRODS;  # number of consumers

# bounded buffer implementation omitted to save space

def producer(i):
    call produce((i, 0));
    call produce((i, 1));
;
def consumer():
    let first = consume():
        let second = consume():
            assert (first[0] != second[0]) or (first[1] < second[1]);
            atomic:
                received = received + { first, second };
            ;
        ;
    ;
;
def main():
    while cardinality(received) < (2 * NCONSS):
        pass;
    ;
    assert received == ({ (i, 0) for i in 1..NPRODS } +
                            { (i, 1) for i in 1..NPRODS })
    ;
;
received = {};
for i in 1..NPRODS:
    spawn producer(i);
;
for i in 1..NCONSS:
    spawn consumer();
;
spawn main();
```

Figure 16.4: Test whether the bounded buffer delivers the right data in the right order.

Chapter 17

Split Binary Semaphores

Reader/writer locks and bounded buffers are both examples of what are sometimes called *conditional critical sections* (CCSs). All critical sections are in fact conditional. Even the basic one has the condition that a process can only enter if no other process is in the critical section. Reader/writer locks refined that, loosening the restriction for improved performance. In the case of bounded buffers, there were two CCSs. First, a CCS to add an entry to the bounded buffer that can only be entered by one process and only if there is room in the bounded buffer (`produce`). Second, a CCS that can only be entered by one process and only if there is an item in the bounded buffer (`consume`).

CCSs are easy to implement using busy waiting:

```
let blocked = True:
    while blocked:
        call lock();
        if application-condition-holds:
            blocked = False;
        else:
            call unlock():
@cr: application-code;
call unlock():
```

We have seen solutions to the reader/writer problem and the producer/consumer problem that do not busy-wait, but they are quite different. It would be nice to have a general technique for all CCS problems. The "Split Binary Semaphore" (SBS) is such a technique, originally proposed by Tony Hoare and popularized by Edsger Dijkstra. SBS is an example of what one could call a "baton-passing" technique. A process needs the baton to access shared variables. When a process has the baton and does not need it any more, it first checks to see if there are any processes that are waiting for the baton and can make progress. If so, it passes the baton directly to one of those processes. If not, it simply gives up the baton.

We will illustrate the technique using the reader/writer problem. Figure 17.1 shows the `rlock_acquire`, `rlock_release`, `wlock_acquire`, and `wlock_release` methods implemented using SBS.

Step 1 is to enumerate all waiting conditions. In the case of the reader/writer problem, there are two: a process that wants to read may have to wait for a writer to leave the critical regiion, while a process that wants to write may have to wait until all readers have left the critical section. If there are N waiting conditions, the SBS technique will use $N+1$ binary semaphores: one for each waiting condition and an extra one that processes use when they first try to enter the critical section and do not yet know if they need to wait. We shall call this the `mutex`.

An important invariant of the SBS technique is that the sum of the $N+1$ semaphores must always be 0 or 1, and it should be 0 when a process is accessing the shared variables. As it were, these semaphores taken together are emulating a single binary semaphore. Initially, `mutex` is 1 and the other N semaphores are all 0.

```
import synch;

def V_one():
    if (w_entered == 0) and (r_waiting > 0):
        call V &(r_sema);
    elif ((r_entered + w_entered) == 0) and (w_waiting > 0):
        call V &(w_sema);
    else:
        call V &(mutex);
    ;
;
def acquire_rlock():
    call P &(mutex);
    if w_entered > 0:
        r_waiting = r_waiting + 1;
        call V &(mutex);
        call P &(r_sema);
        r_waiting = r_waiting - 1;
    ;
    r_entered = r_entered + 1;
    call V_one();
;
def release_rlock():
    call P &(mutex);
    r_entered = r_entered - 1;
    call V_one();
;
def acquire_wlock():
    call P &(mutex);
    if (r_entered + w_entered) > 0:
        w_waiting = w_waiting + 1;
        call V &(mutex);
        call P &(w_sema);
        w_waiting = w_waiting - 1;
    ;
    w_entered = w_entered + 1;
    call V_one();
;
def release_wlock():
    call P &(mutex);
    w_entered = w_entered - 1;
    call V_one();
;
mutex = Semaphore(1); r_sema = Semaphore(0); w_sema = Semaphore(0);
r_entered = 0; r_waiting = 0;
w_entered = 0; w_waiting = 0;
```

Figure 17.1: Reader/Writer Lock implementation using Split Binary Semaphores.

To maintain the invariant, each process alternates `P` and `V` operations, starting with `P &(mutex)` when it tries to enter the CCS, and ending with a `V` operation on one of the semaphores (yet to be determined).

Another important part of the SBS technique is to keep careful track of the state. In the case of the reader/writer lock, the state consists of the following shared variables:

- `r_entered`: the number of readers in the critical section;
- `w_entered`: the number of writers in the critical section;
- `r_waiting`: the number of readers waiting to enter;
- `w_waiting`: the number of writers waiting to enter.

Each of the `rlock_acquire`, `rlock_release`, `wlock_acquire`, and `wlock_release` methods must maintain this state. To wit, check out `rlock_acquire`. It first checks to see if there are any writers. If so, it increments `r_waiting`. If not, it increments `r_entered`.

Either way, it vacates one of the semaphores next. `V_one()` is the baton passing function that selects which of the binary semaphores to vacate. Method `V_one()` first checks to see if there are any readers or writers waiting. If there are readers waiting and there are no writers in the critical section, it vacates the semaphore associated with readers waiting. If there are writers waiing and there are no readers nor writers in the critical section, it vacates the semaphore associated with writers waiting. If both conditions hold, it could vacates either one—it would not matter. However, it can only vacate one of the semaphores. If none of the conditions hold, `V_one()` vacates `mutex`.

Let us return now to `rlock_acquire`, in the case that the process found there is a writer in the critical section, it vacates `mutex` and procures `r_sema`, blocking on the semaphore associated with waiting readers. When later some other process passes the baton to it by incrementing `r_sema`), it updates the state by decrementing `r_waiting` and incrementing `r_entered`. It then invokes `V_one()` one more time.

As an example, consider the case where there is a writer in the critical section and there are two readers waiting. Let us see what happens when the writer calls `wlock_release`:

1. After procuring `mutex`, the sum of the three semaphores is 0. The writer then decrements `w_entered` and calls `V_one()`.
2. `V_one()` finds that there are no writers in the critical section and there are two readers waiting. It therefore vacates `r_sema`. The sum of the semaphores is now 1, because `r_sema` is 1. This guarantees that only a waiting reader, procuring `r_sema`, can enter the critical section.
3. When it does, the reader decrements `r_waiting` from 2 to 1, and increments `r_entered` from 0 to 1. The reader finally calls `V_one()`.
4. Again, `V_one()` finds that there are no writers and that there are readers waiting, so again it vacates `r_sema`, passing the baton to the one remaining waiting reader.
5. The remaining reader decrements `r_waiting` from 1 to 0 and increments `r_entered` from 1 to 2.
6. Finally, the remaining reader calls `V_one()`. `V_one()` does not find any processes waiting, and so it vacates `mutex`.

- Implement the producer/consumer problem with SBS.

Chapter 18

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 solutions that we presented thus far. 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. In particular, the probability that either a reader or writer gets starved will be exponentially diminishing. We shall call such a solution *fair* (although it does not quite match the usual formal nor vernacular concepts of fairness).

It is difficult if not impossible to make the reader/writer lock solution of Figure 13.1 fair. However, the split binary semaphore solution of Figure 17.1 provides much control over whom to pass baton to. Figure 18.1 contains a version of Figure 17.1 that is a fair solution to the reader/writer problem. There are two changes compared to the original version:

1. When a reader tries to enter the critical section, it yields not only if there are writers in the critical section, but also if there are writers waiting to enter the critical section;

2. There are two versions of `V_one()`. In particular, when the last reader leaves the critical section, it preferentially passes the baton to a waiting writer rather than to a waiting reader. Similarly, when a

```
def V_one_r():  # prefers reader next over writer next
    if (w_entered == 0) and (w_waiting == 0) and (r_waiting > 0):
        call V &(r_sema);
    elif ((r_entered + w_entered) == 0) and (w_waiting > 0):
        call V &(w_sema);
    else:
        call V &(mutex);
    ;
;
def V_one_w():  # prefers writer next over reader next
    if ((r_entered + w_entered) == 0) and (w_waiting > 0):
        call V &(w_sema);
    elif (w_entered == 0) and (w_waiting == 0) and (r_waiting > 0):
        call V &(r_sema);
    else:
        call V &(mutex);
    ;
;
def acquire_rlock():
    call P &(mutex);
    if (w_entered > 0) or (w_waiting > 0):
        r_waiting = r_waiting + 1;
        call V &(mutex); call P &(r_sema);
        r_waiting = r_waiting - 1;
    ;
    r_entered = r_entered + 1;
    call V_one_r();     # only other readers can enter
;
def release_rlock():
    call P &(mutex);
    r_entered = r_entered - 1;
    call V_one_w();     # other writers have right of way
;
def acquire_wlock():
    call P &(mutex);
    if (r_entered + w_entered) > 0:
        w_waiting = w_waiting + 1;
        call V &(mutex); call P &(w_sema);
        w_waiting = w_waiting - 1;
    ;
    w_entered = w_entered + 1;
    call V &(mutex);    # no other process can enter
;
def release_wlock():
    call P &(mutex);
    w_entered = w_entered - 1;
    call V_one_r();     # other readers have right of way
;
```

Figure 18.1: Reader/Writer Lock implementation addressing fairness.

writer leaves the critical section, it preferentially passes the baton to a waiting reader rather than to a waiting writer.

The net effect of this is that if there is contention between readers and writers, than readers and writers end up alternating entering the critical section. While readers can still starve other readers and writers can still starve other writers, they can not starve one another. And contention for a semaphore is resolved probabilistically, each reader and each writer will eventually have a chance to enter the critical section almost surely.

Chapter 19

Monitors

Tony Hoare, who came up with the concept of split binary semaphores, devised an abstraction of the concept in a programming language paradigm called *monitors*. A monitor is a special version of an object-oriented *class*, comprising a set of variables and methods that operate on those variables. There is a split binary semaphore associated with each such class. The `mutex` is hidden: it is automatically acquired when invoking a method and released upon exit. The other semaphores are called *condition variables*. There are two operations on condition variables: `wait` and `signal`. If `c` is a pointer to a condition variable, then the semantics of the operations, in CXL, are as follows:

```
1  def wait(c):
2      call V &(mutex);
3      call P c;
4  ;
5  def signal(c):
6      call V c;
7      call P &(mutex);
8  ;
```

`wait` releases the mutex and blocks while trying to procure the condition variable (which is a semaphore that is 0 when `wait` is invoked). `signal` vacates the condition variable, passing the baton to a process trying to procure it, and then procures the mutex. The use of `wait` and `signal` ensures that `P` and `V` operations alternate as prescribed by the SBS technique. Our implementations of reader/writer locks can be easily changed to use `wait` and `signal` instead of using `P V`.

In the late 70s, Xerox PARC developed a new programming language called Mesa [**?**]. Mesa introduced various important concepts to programming languages, including software exceptions and incremental compilation. Mesa also incorporated a version of monitors. However, there are some subtle but important differences with Hoare monitors that make Mesa monitors quite unlike split binary semaphores.

As in Hoare monitors, there is a hidden mutex associated with each monitor, and the mutex is automatically acquired upon entry to a method and released upon exit. Mesa monitors also have condition variable that a process can wait on. Like in Hoare monitors, the `wait` operation release the mutex. The most important difference is in what `signal` does. To make the distinction more clear, we shall call the corresponding Mesa operation `notify` rather than `signal`. When a process p invokes `notify`, it does not immediately pass the baton to some process q requiring the baton. Instead, p holds onto the baton until it leaves the monitor. At that point q gets the baton, assuming there even was such a process q.

Figure 19.1 illustrates the difference with a drawing. Here a bathroom represents the critical section, allowing only one process at a time say. The bedrooms represent condition variables. There is some condition associated with each bedroom. In the hall are processes waiting to enter the critical section.

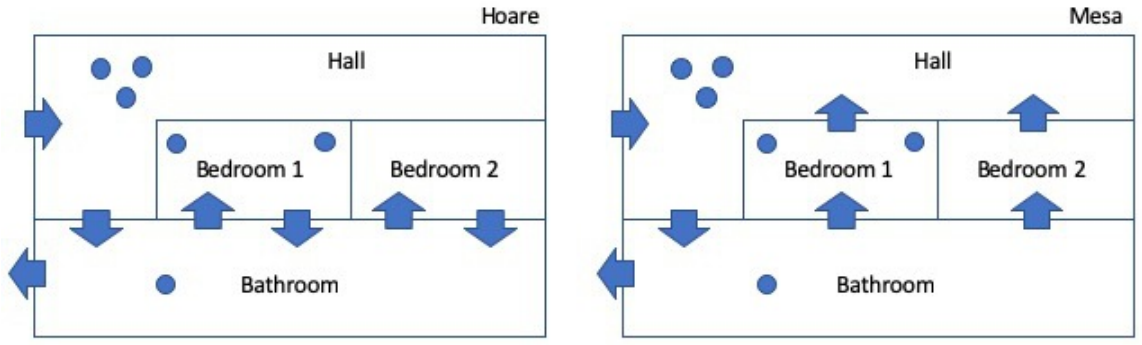

Figure 19.1: High-level depictions of Hoare and Mesa monitors. The hall is where processes wait to enter the bathroom (the critical section). The bedrooms illustrate two condition variables. The circles are processes.

On the left is a Hoare monitor. When a process leaves the monitor (let's call that process p), it must be in the bathroom. It can choose to either let in a process in one of the bedrooms or a process in the hall, assuming there are any. To make the choice, p checks for each bedroom if its condition holds and if there are processes in the bedroom. If there are such bedrooms, p selects one and lets that process into the bathroom. Let's call that process q. Importantly, when q enters the bathroom, the condition that q was waiting for still holds.

On the right is a Mesa monitor. The only way into the bathroom is through the hall. When process p leaves the bathroom, if also checks each of the bedrooms to see if there is a process q that can run. But instead of letting q go straight into the bathroom, q goes into the hall, joining any other processes that are also waiting to enter the bathroom. Finally, when p has left the bathroom, one of the processes in the hall is let into the bathroom. But that may not be q. Assuming every process eventually leaves the bathroom and the system is fair, eventually q will enter the bathroom. However, it is no longer guaranteed that its condition holds. Therefore, the first thing q must do is check, again, to see if the condition holds. If not, it should go back into the bedroom corresponding to the condition.

Mesa monitors provide another important option: when process p leaves the bathroom, it can choose to let any number of processes in the bedrooms into the hall. It can do so by calling `notify` repeatedly, each time letting one process in one of the bedrooms into the hall. Using `notify` on an empty bedroom is considered a no-op in Mesa. (Hoare monitors do not allow using `signal` on an empty bathroom.) Mesa also has a function called `notifyAll` (aka `broadcast`) that, when applied to a bedroom, let's all processes in that bedroom into the hall.

While Hoare monitors are conceptually simpler, the so-called "Mesa monitor semantics" have become more popular, adopted by all major programming languages including C, Java, and Python.

Chapter 20

Mesa Condition Variables in CXL

The `synch` module supports Mesa condition variables. However, like C, Java, and Python, it does not have built-in language support for them but instead associates condition variables with explicit locks. Figure 20.1 shows the `synch` interface to condition variables. `Condition(lk)` creates a new condition variable associated with the given pointer to a lock `lk`. The condition variable itself is represented by a dictionary containing the pointer to the lock and a bag of name tags of processes waiting on the condition variable. (The `synchS` library instead has a list of contexts.)

`wait` adds the nametag of the process to the bag. This increments the number of processes in the bag with the same context. `wait` then waits until that count is restored to the value that it had upon entry to `wait`. `notify` removes an arbitrary context from the bag, allowing one of the processes with that context to return from `wait`. `notifyAll` empties out the entire bag, allowing all processes in the bag to resume.

To illustrate how Mesa condition variables are used in practice, we demonstrate, once again, using an implementation of reader/writer locks. Figure 20.2 shows the code. `rwlock` is the global lock or mutex for the reader/writer lock. There are two condition variables: readers wait on `rcond` and writers wait on `wcond`. The implementation also keeps track of the number of readers and writers in the critical section.

When inspecting the code, one important feature to notice is that `wait` is always invoked within a `while` loop that checks for the condition that the process is waiting for. We explained why in Chapter 19: when a process resumes after being notified, it is not guaranteed that the condition it was waiting for holds, even if it did at the time of invoking `notify` or `notifyAll`. It is *imperative* that you always have a `while` loop around any invocation of `wait`.

In `release_rlock`, notice that `notify &(wcond)` is invoked when there are no readers left, *without* checking if there are writers waiting to enter. With Mesa monitors this is ok, because calling `notify` is a no-op if no process is waiting.

`release_wlock` executes `notifyAll &(wcond)` as well as `notify &(wcond)`. Again, because we do not keep track of the number of waiting readers or writers, we have to conservatively assume that all waiting readers can enter, or, alternatively, up to one waiting writer can enter. So `release_wlock` wakes up all potential candidates. There are two things to note here. First, unlike split binary semaphores or Hoare monitors, where multiple waiting readers would have to be signaled one at a time in a baton-passing fashion (see Figure **??**), with Mesa monitors all readers are awaked in one fell swoop using `notifyAll`. Second, both readers and writers are awakened—this is ok because both execute `wait` within a `while` loop, re-checking the condition that they are waiting for. So if both type of processes are waiting, either all the readers get to enter next or one of the writers gets to enter next.

```
import bag;

def Condition(lk):
    result = dict{ .lock: lk, .waiters: bagEmpty() };
;
def wait(c):
    let lk = (^c).lock, blocked = True, cnt = bagCount((^c).waiters, nametag()):
        call bagAdd(&((^c).waiters), nametag());
        ^lk = False;
        while blocked:
            atomic:
                if (not (^lk)) and (bagCount((^c).waiters, nametag()) <= cnt):
                    ^lk = True;
                    blocked = False;
                ;
            ;
        ;
    ;
;
def notify(c):
    let waiters = (^c).waiters:
        if waiters != bagEmpty():
            call bagRemove(&((^c).waiters), bagChoose(waiters));
        ;
    ;
;
def notifyAll(c):
    (^c).waiters = bagEmpty();
;
```

Figure 20.1: Implementation of condition variables in the `synch` module.

```
import synch;

def acquire_rlock():
    call lock &(rwlock);
    while nwriters > 0:
        call wait &(rcond);
    ;
    nreaders = nreaders + 1;
    call unlock &(rwlock);
;
def release_rlock():
    call lock &(rwlock);
    nreaders = nreaders - 1;
    if nreaders == 0:
        call notify &(wcond);
    ;
    call unlock &(rwlock);
;
def acquire_wlock():
    call lock &(rwlock);
    while (nreaders + nwriters) > 0:
        call wait &(wcond);
    ;
    nwriters = 1;
    call unlock &(rwlock);
;
def release_wlock():
    call lock &(rwlock);
    nwriters = 0;
    call notifyAll &(rcond);
    call notify &(wcond);
    call unlock &(rwlock);
;
rwlock = Lock();
rcond = Condition &(rwlock);
wcond = Condition &(rwlock);
nreaders = 0;
nwriters = 0;
```

Figure 20.2: Unfair Reader/Writer Lock implemtation using Mesa-style condition variables.