# Concurrent Programming with CXL

Robbert van Renesse

June 19, 2020

# Contents

# 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 (aka *threads*[1]) read and update shared variables concurrently 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 execution of 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 processes may update memory locations while 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 simple or it will take longer than we may care to wait for or run out of memory. In particular, the model needs to have a relatively small number of reachable states.

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

[1] We do not make a distinction between processes and threads.

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. 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 [Lam02, Lam09]. 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" [Sch97].

The book proceeds as follows:

- Chapter 2 introduces the CXL programming language, as it provides the language for presenting synchronization problems and solutions.
- Chapter 3 illustrates the problem of concurrent programming through a simple example in which two processes are concurrently incrementing a counter.
- Chapter 4 presents the CXL virtual machine to understand the problem underlying concurrency better.
- Chapter 5 introduces the concept of a *critical section* and presents various flawed implementations of critical sections to demonstrate that implementing a critical section is not trivial.
- Chapter 6 introduces Peterson's Algorithm, an elegant solution to implementating a critical section.
- Chapter 7 gives some more details on the CXL language.
- Chapter 8 introduces hardware-supported locks for implemented critical sections. Chapter 9 presents a CXL module that supports locks and other synchronization primitives.
- Chapters 10 and 11 present the abstraction and implementation of reader/writer locks, which allow multiple processes in a critical section if they are all only executing read-only accesses. Chapter 10 also talks about busy-waiting, which is an undesirable approach to synchronize processes.

- Chapter 12 describes *deadlock* where a set of processes are indefinitely waiting for one another to release a resource.
- Chapter 13 introduces *semaphores*, a generalization of locks that is good not only for mutual exclusion but also for waiting for certain application-level conditions.
- Chapters 14 and 15 introduce the *bounded buffer problem* (aka *producer/consumer problem*) and various solutions.
- Chapter 16 talks about *starvation*: the problem that in some synchronization approaches processes may not be able to get access to a resource they need.
- Chapters 17 and 18 present *monitors* and *condition variables*, another approach to process synchronication.
- Chapter 19 revisits deadlocks and how to avoid them.
- Chapter 20 presents the *actor model* as an approach to synchronization.
- Chapter 21 introduces *non-blocking* or *wait-free* synchronization algorithms, which prevent processes waiting for one another more than a bounded number of steps.
- Chapter 22 describes *barrier synchronization*, useful in high-performance computing applications such as parallel simulations.
- Chapter 23 presents a problem and a solution to the distributed systems problem of having two processes communicate reliably over an unreliable network.

# 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, iterators, or I/O; CXL does support `for` loops and various comprehensions over sets.
- Python is object-oriented, supporting classes with methods and inheritance; CXL has objects but does not support classes. CXL supports pointers to objects.

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

Figure 2.1 shows a simple example of a CXL program. (The code for examples in this book can be found in the `code` folder under the name listed in the caption of the example.) The example is sequential and has a method `triangle` 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

```
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: [triangle.cxl] Computing triangle numbers.

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 $x(x+1)/2$.

"Running" this CXL program will try all possible executions, which includes all possible values for $x$. Try it out (here `$` represents a shell prompt):

```
$ 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 value, 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
$
```

## Exercises

1. See what happens if, instead of initializing `result` to 0, you initialize it to 1. (You do not need to understand the error report at this time.) Explain in English what the problem is.

2. Write a CXL program that computes squares by repeated adding. So the program should compute the square of $x$ by adding $x$ to an initial value of 0 $x$ times.

# 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 until `done` flags are set to `True`. 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()`.

- 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 4, 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;
2. incrementer 1 loads the value of `count`, which is still 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: [Up.cxl] Incrementing twice in parallel.

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. This is known as a *race condition*. 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.

## Exercises

The following exercises are intended to show you that on the one hand it is easy to translate CXL programs into Python, but on the other hand, it is easier to find concurrency bugs using CXL.

1. CXL programs can usually be easily translated into Python. For example, Figure 3.2 is a Python version of Figure 3.1.
   (a) Run Figure 3.2 using Python. Does the assertion fail?
   (b) Using a script, run Figure 3.2 1000 times. How many times does the assertion fail (if any).
2. Figure 3.3 is a version of Figure 3.2 that has each incrementer thread increment `count` `N` times. Run Figure 3.2 10 times (using Python). Report how many times the assertion fails and what the value of `count` was for each of the failed runs. Also experiment with lower values of `N`.

```
import threading

def incrementer(self):
    global count
    count = count + 1
    done[self] = True

def main():
    while not (done[0] and done[1]):
        pass
    assert count == 2, count
    print("Done")

def spawn(f, a):
    threading.Thread(target=f, args=a).start()

count = 0
done = [ False, False ]
spawn(incrementer, (0,))
spawn(incrementer, (1,))
spawn(main, ())
```

Figure 3.2: [Up.py] Python implementation of Figure 3.1.

3. Can you think of a fix to Figure 3.1? Try one or two different fixes and run them through CXL. Do not worry about having to come up with a correct fix at this time—the important thing is to develop an understanding of concurrency.

```
import threading

N = 10000000

def incrementer(self):
    global count
    for i in range(N):
        count = count + 1
    done[self] = True

def main():
    while not (done[0] and done[1]):
        pass
    assert count == 2*N, count
    print("Done")

def spawn(f, a):
    threading.Thread(target=f, args=a).start()

count = 0
done = [ False, False ]
spawn(incrementer, (0,))
spawn(incrementer, (1,))
spawn(main, ())
```

Figure 3.3: [UpMany.py] Incrementing `N` times.

# Chapter 4

# The CXL Virtual Machine

CXL programs are compiled to CXL bytecode (machine instructions for a virtual machine), which in turn is executed by the CXL virtual machine (CVM). To understand the problem of concurrent computing, it is important to have a basic understanding of the CVM.

## CXL Values

CXL programs, and indeed the CVM, 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,]`.

```
Up.cxl:1 def incrementer(self):
   0 Jump 32
   1 Frame incrementer(self) end=9
Up.cxl:2     count = count + 1;
   2 Push 1
   3 Load count
   4 2-ary +
   5 Store count
Up.cxl:3     done[self] = True;
   6 Push True
   7 PushVar self
   8 Store done 1
   9 Return
```

Figure 4.1: The first part of the CVM bytecode corresponding to Figure 3.1.

```
   ctx = __init__/() pc =   0 #states = 1 diameter = 0 queue = 0
CXL Assertion failed ('assert', 'Up.cxl', 9, 5) 1
#states = 79
==== Safety violation ====
__init__/() [0,32-47]       48 dict{ .count:0, .done:dict{ 0:False, 1:False } }
incrementer/0 [1-4]          5 dict{ .count:0, .done:dict{ 0:False, 1:False } }
incrementer/1 [1-8]          9 dict{ .count:1, .done:dict{ 0:False, 1:True  } }
incrementer/0 [5-8]          9 dict{ .count:1, .done:dict{ 0:True,  1:True  } }
main/() [11-15,18-22,24-29] 29 dict{ .count:1, .done:dict{ 0:True,  1:True  } }
```

Figure 4.2: The output of running Figure 3.1.

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.

Appendix 4 provides details on all the values that CXL currently supports.

# CXL Bytecode

A CXL program is translated into *bytecode*, which is list of machine instructions that the CXL virtual machine (CVM) executes. The CVM 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 CVM manipulates CXL values. A CVM has the following components:

- Code: This is an immutable and finite list of CVM instructions, generated from a CXL program. The types of instructions will be described later.
- Shared memory: A CVM 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* (aka continuation): it contains the value of its name tag, program counter, stack, and register. The state of a CVM 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 CVM bytecode listed in Figure 4.1. You can obtain this code by invoking `cxl` with the `-a` flag like so:

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

Each process in the CVM is predominantly a *stack machine*, but it also has a register. 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 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.

Appendix E provides a list of all CVM machine instructions, in case you want to read about the details. The `Frame` instruction lists the name of the method, names of its arguments, 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 `Load` instruction pushes the value of the `count` variable onto the stack.
4. `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.
5. The `Store` instruction pops pops a CXL value (the sum of the `count` variable and 1) and stores it in 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 4.2 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 CVM instructions that the process executed;
3. The current program counter of the process;

```
1  def incrementer(self):
2      entered[self] = True;
3      if entered[1 - self]:
4          while not done[1 - self]:
5              pass;
6          ;
7      ;
8      count = count + 1;
9      done[self] = True;
10 ;
11 def main():
12     while not (done[0] and done[1]):
13         pass;
14     ;
15     assert count == 2, count;
16 ;
17 count = 0;
18 entered = [ False, False ];
19 done = [ False, False ];
20 spawn incrementer(0);
21 spawn incrementer(1);
22 spawn main();
```

Figure 4.3: [UpEnter.py] Broken attempt at fixing the code of Figure 3.1.

4. The contents of the shared memory.

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

- Process 0 executed instructions 1 through 4, loading the value of `count` but stopping just before storing 1 into `count`;
- Process 1 executed instructions 1 through 8, storing 1 into into `count` and storing `True` into `done[1]`;
- Process 0 continues execution, storing value 1 into `count` and storing `True` into `done[0]`.

This makes precise the concurrency problem that we encountered.

## Exercises

1. Figure 4.3 shows an attempt at trying to fix the code of Figure 3.1. Run it through CXL and see what happens. Based on the error output, describe in English what is wrong with the code by describing, in broad steps, how running the program can get into a bad state.
2. What if we moved line 2 of Figure 4.3 to after the `if` statement (between lines 7 and 8)? Do you think that would work? Run it through CXL and describe either why it works or why it does not work.

# Chapter 5

# 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. This is called a *critical section* (aka critical region) [Dij65]: 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. An implementation of a data structure that can be safely accessed by multiple processes (or threads) is called *thread-safe*.

A critical section is often modeled as processes in an infinite loop entering and exiting the critical section. Figure 5.1 shows the CVM bytecode. Here `@cs` is a *label*, identifying a location in the CVM bytecode. The first thing we need to ensure is that there can never be two processes at the critical section. This property is called `mutual exclusion`. 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

```
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 5.1: [csbarebones.cxl] 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 5.2: [cx.cxl] CXL model of a critical section.

current process. If you run the code through CXL, the assertion should fail because there is no code yet for entering and exiting the critical section.

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

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

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

Mutual exclusion is an example of a *safety property*, a property that ensures that *nothing bad will happen*, in this case two processes being in the critical section. What we need now a *liveness property*: we want to ensure that *eventually something good will happen*. There are various possible liveness properties we could use, but here we will propose the following informally: if there exists a non-empty set $S$ of processes that 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 5.1, the spawned processes never terminate. In order to resolve this, we will model processes in critical sections using the framework in Figure 5.2: 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.

Figure 5.3 show a high-level state diagram of the code in Figure 5.2. The circles represent states, while the arrows represent possible state transitions (high-level steps). 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.

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 5.4 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`.

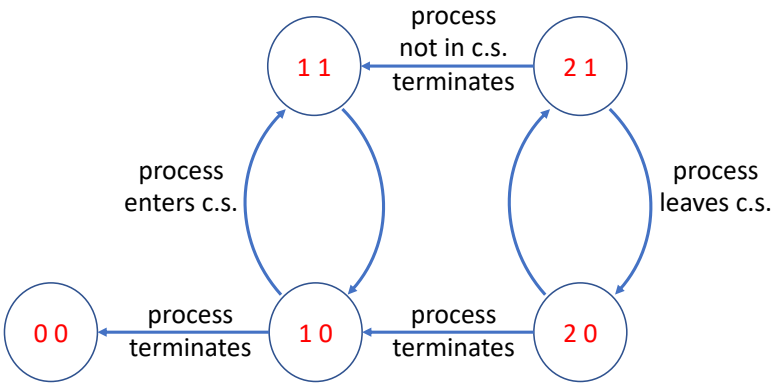


Figure 5.3: High-level state diagram specification of mutual exclusion with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes in the critical section.

```
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 5.4: [naiveLock.cxl] 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 5.5: [naiveFlags.cxl] 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 5.6: [naiveTurn.cxl] Naïve use of turn variable to solve mutual exclusion.

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 5.5 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.

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.

## Exercises

1. Run Figure 5.2 using CXL. As there is no protection of the critical section, mutual is violated, the assertion should fail, and a trace should be reported. Now plug `while True: pass;;` before entering the critical section in Figure 5.2 and run CXL again. Now mutual exclusion is guaranteed but progress is violated. CXL should print a trace to a state from which a terminating state cannot be reached. Describe in English the difference in the failure reports.

2. If we change the loop in Figure 5.2 into an infinite loop, does the *progress* property guarantee that all processes that are trying to enter eventually enter? Why or why not?

3. 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 6

# Peterson's Algorithm

In 1981, Peterson came up with a beautiful solution to the mutual exclusion problem, now known as "Peterson's Algorithm" [Pet81]. The algorithm is an amalgam of the (incorrect) algorithms in Figures 5.5 and 5.6, and is presented in Figure 6.1. 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 4 we talked about the concept of *state*: at any point of time the CVM 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 CVM 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 CVM 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.

It is often useful to classify states into sets. *Initial*, *final*, and *reachable*, and *unreachable* are all examples of classes of states. Figure 6.2 depicts a Venn diagram of various classes of states and a trace. One way to classify states is to define a predicate over states. All states in which `x = 1`, or all states where there are two or more processes executing, are examples of such predicates. For our purposes, it is useful to define a predicate that says that at most one process is in the critical section. We shall call such states *exclusive*.

An *invariant* of a program is a predicate that holds over all states that are reachable by that program. We want to show that exclusivity is an invariant. In other words, we want to show that the set of reachable states of executing the program 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 predicate is an invariant is through induction on the number of steps. First you prove that the predicate holds over the initial state. Then you prove that for every reachable state, and for every step from that reachable state, the predicate 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.

```
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 = choose({0, 1});
spawn process(0);
spawn process(1);
```

Figure 6.1: [Peterson.cxl] Peterson's Algorithm

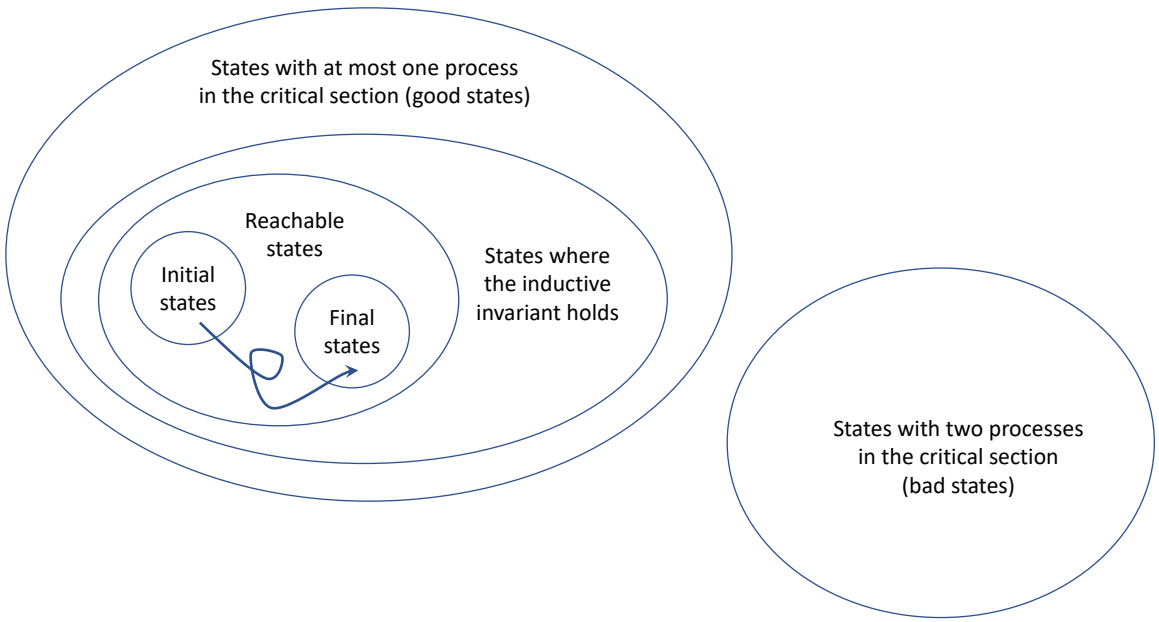


Figure 6.2: Venn diagram classifying all states and a trace.

```
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 = choose({0, 1});
nametags = [ dict{ .name: .process, .tag: tag } for tag in 0..1 ];
spawn process(0);
spawn process(1);
```

Figure 6.3: [PetersonInductive.cxl] Peterson's Algorithm with Inductive Invariant

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.
- 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 exclusivity itself. After all, it certainly holds over the initial state. And as CXL has already determined, exclusivity 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, entering a state that is not exclusive. So exclusivity 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* exclusivity. 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 6.3 formalizes $I_p(i)$ in CXL. The label `@gate` refers to the instruction that sets `turn` to $1-i$. You can run Figure 6.3 to determine that $I_p(i)$ is indeed an invariant for $i = 0, 1$.

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

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 6.3 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.

# Exercises

1. Figure 6.4 presents another correct solution to the mutual exclusion problem. It is similar to the one in Figure 5.5, but has a process *back out and try again* if it finds that the other process is either trying to enter the critical section or already has. Compare this algorithm with Peterson's. What are the advantages and disadvantages?

2. Can you find an inductive invariant for the algorithm in Figure 6.4 to prove it correct? Here's a pseudo-code version of the algorithm to help you. Each line is an atomic action:

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

        # 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 6.4: [csonebit.cxl] Mutual exclusion using a flag per process.

```
initially: flagX = flagY = False

process X:                          process Y:
    X0: flagX = True                   Y0: flagY = True
    X1: if not flagY goto X4           Y1: if not flagX goto Y4
    X2: flagX = False                  Y2: flagY = False
    X3: goto X0                        Y3: goto Y0
    X4: ...critical section...         Y4: ...critical section...
    X5: flagX = False                  Y5: flagY = False
```

3. A colleague of the author asked if the first two assignments in Peterson's algorithm (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. See if you can figure out for yourself if the two assignments can be reversed. Then run the program in Figure 6.1 after reversing the two assignments and describe in English what happens.

4. Bonus question: Can you generalize Peterson's algorithm to more than two processes?

# Chapter 7

# 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` 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. In particular, `^&x == x`. This is similar to how C pointers work.

Figure 7.1 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: choose({0, 1}), .flags: [ False, False ] };
;

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

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

Figure 7.1: [PetersonMethod.cxl] Peterson's Algorithm accessed through methods.

# Chapter 8

# Spinlock

Figure 5.4 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 CVM 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:

```
1  def atomic_inc(ptr):
2      atomic:
3          ^ptr = ^ptr + 1;
4      ;
5  ;
6  count = 0;
7  atomic_inc(&count);
```

Many CPUs have an atomic "test-and-set" (TAS) operation. Method `tas` in Figure 8.1 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 8.1 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.

```
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]:
            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);
;
```

Figure 8.1: [spinlock.cxl] 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 8.2: [spinlockInv.cxl] Checking invariants.

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

Invariant $I_1$ is also easy to verify for exiting the critical section. Invariant $I_2(i)$ 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 8.1 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 8.2 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.

## Exercises

1. Implement an atomic swap operation. It should take two pointer arguments and swap the values.
2. Implement a spinlock using the atomic swap operation.
3. Write out the invariants that need to hold and check them using CXL.

# Chapter 9

# Locks and the Synch Module

In Figure 8.1 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 9.1 shows a more straightforward implementation of spinlock. The solution is similar to the naïve solution of Figure 5.4, 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 synchronization primitives like test-and-set. Other common examples include compare-and swap and fetch-and-add. Atomic statements are not intended to *replace* locks or other synchonization primitives. When solving synchronization problems you should not directly use atomic statement but use the synchronization primitives that are available to you. But if you want to design a new synchronization primitive, then use `atomic` by all means.

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 9.2 shows how they are implemented, and Figure 9.3 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` statement:

```
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 9.1: [csTAS.cxl] Fixed version of Figure 5.4 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 9.2: The `Lock` interface in the `synch` module.

```
import synch;

def process(self):
    lock(&countlock);
    count = count + 1;
    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 9.3: [UpLock.cxl] Program of Figure 3.1 fixed with a lock.

```
go C R
```

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 9.4 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 16 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:
            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 9.4: The `Lock` interface in the `synchS` module uses suspension.

## Exercises

1. Write a synchronization module using your own implementation of `Lock` (for example, using Peterson's algorithm or using `swap`), and run the code in Figure 9.3 using your own module.

# Chapter 10

# 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* [CHP71]. We will explore various ways of implementing reader/writer locks in this chapter and future ones.

Figure 10.1 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 10.2 shows a high-level specification for two processes.

- Draw additional states and steps in Figure 10.2 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` $\leq 1$).

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 9.2).

```
import synch;

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

Figure 10.1: [RWbusy.cxl] Busy-Waiting Reader/Writer Lock implementation.

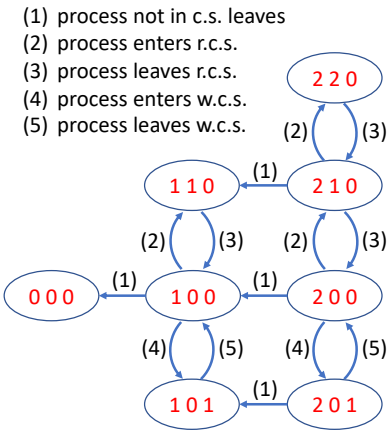


Figure 10.2: High-level state diagram specification of reader/writer locks with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes reading in the critical section; the third is the number of processes writing in the critical section.

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 10.1 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 10.1, but, using the right test, CXL can be used to check for busy-waiting. Figure 10.3 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():
    lock(&rlock);
    if nreaders == 0:
        lock(&rwlock);
    ;
    nreaders = nreaders + 1;
    unlock(&rlock);
;
def release_rlock():
    lock(&rlock);
    nreaders = nreaders - 1;
    if nreaders == 0:
        unlock(&rwlock);
    ;
    unlock(&rlock);
;
def acquire_wlock():
    lock(&rwlock);
;
def release_wlock():
    unlock(&rwlock);
;
def reader(self):
    acquire_rlock();
    @rcs: assert atLabel.wcs == dict{};
    release_rlock();
;
def writer():
    acquire_wlock();
    @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                 (atLabel.rcs == dict{})
          ;
    release_wlock();
;
rwlock = Lock();
rlock = Lock();
nreaders = 0;
for i in 1..NREADERS: spawn reader(i);
;
for i in 1..NWRITERS: spawn writer();
;
acquire_rlock();
```

Figure 10.3: [RWbusychk.cxl] Checking for Busy Waiting.

# Chapter 11

# Reader/Writer Locks with Blocking

Figure 11.1 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 11.2 using an ordinary lock, never allowing more than one process in the critical section.

Figure 11.2 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.

## Exercises

1. Why do you suppose the code in Figure 11.2 uses a flag per process rather than a simple counter?

```
import synch;

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

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

Figure 11.1: [RWlock.cxl] Reader/Writer with Two Locks.

```
import synch;

const NREADERS = 2;
const NWRITERS = 2;

def acquire_rlock():
    lock(&rwlock);
;
def release_rlock():
    unlock(&rwlock);
;
def acquire_wlock():
    lock(&rwlock);
;
def release_wlock():
    unlock(&rwlock);
;
def reader(self):
    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;
                ;
            ;
        ;
    ;
    release_rlock();
;
def writer():
    acquire_wlock();
    @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                 (atLabel.rcs == dict{})
          ;
    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 11.2: [RWmulti.cxl] Checking that multiple readers can acquire the read lock.

2. Can you write a version of this test that uses a counter instead of a flag per process?

3. Write a "library implementation" of reader/writer locks so multiple reader/writer locks can be instantiated.

# Chapter 12

# 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 12.1 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 (Chapter 19), but one important approach to preventing deadlock is to prevent a cycle from forming, with each process waiting for the next process on the cycle. We can do this by establishing 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:

```
1       if left < right:
2           lock(&forks[left]);
3           lock(&(orks[right]);
4       else:
5           unlock(&forks[right]);
6           unlock(&forks[left]);
7       ;
```

- Try it out!

```
import synch;

const N = 5;

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

Figure 12.1: [Diners.cxl] Dining Philosophers.

# Chapter 13

# 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 use one fork each to eat spaghetti—they need the other to speak[1]). 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 [Dij62]. 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 13.1 shows the `synch` module implementation of semaphores. Figure 13.2 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.

You may find the free "The Little Book of Semaphores" by Allen Downey a great resource for working with semaphores [Dow09].

[1] The joke was provided to me by my dear friend and colleague Lorenzo Alvisi.

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

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

```
import synch;

const NDINERS = 5;
const NFORKS = 3;

def diner(which):
    while choose({ False, True }):
        P(&forks);
        # dine
        V(&forks);
        # think
    ;
;
forks = Semaphore(NFORKS);
for i in 1..NDINERS:
    spawn diner(i);
;
```

Figure 13.2: [DinersSema.cxl] Italian Philosophers.

# Chapter 14

# Bounded Buffer

A good example of a resource with multiple units is the so-called *bounded buffer* (aka *ring 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 14.1 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 14.2 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

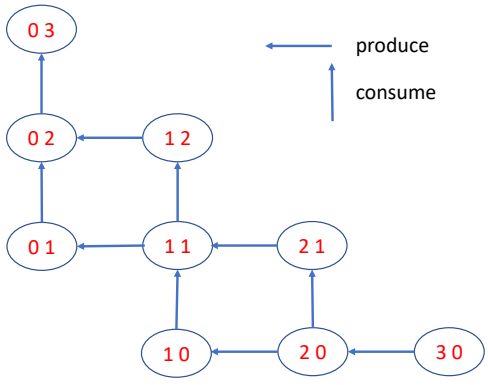


Figure 14.1: High-level state diagram specification of producers/consumers.

```
import synch;

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

def produce(item):
    P(&n_empty);
    P(&l_in);
    buf[b_in] = item;
    b_in = (b_in % NSLOTS) + 1;
    V(&l_in);
    V(&n_full);
;
def consume():
    P(&n_full);
    P(&l_out);
    result = buf[b_out];
    b_out = (b_out % NSLOTS) + 1;
    V(&l_out);
    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 14.2: [BBsema.cxl] 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 14.3: Script for testing producer/consumer synchronization (assuming the code is in file `PCsema.cxl`).

atomically decrements `n_empty` once this is the case. In other words, the producer procures an empty slot. 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 14.1, by experimenting with different non-negative values for the three constants. In particular, any run with a positive `NSLOTS` and `NCONSS` $\leq$ `NPRODS` $\leq$ `NCONSS` + `NSLOTS` should terminate, while other values should lead to processes blocking. One could write a script like the one in Figure 14.3.

Figure 14.4 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):
    produce((i, 0));
    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 14.4: [BBsemadata.cxl] Test whether the bounded buffer delivers the right data in the right order.

# Chapter 15

# Split Binary Semaphores

Reader/writer locks and bounded buffers are both examples of what are sometimes called *conditional critical sections* (CCSs) [Hoa73]. All critical sections are, in a way, conditional. Even the basic critical section 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:
        lock();
        if application-condition-holds:
            blocked = False;
        else:
            unlock():
@cr: application-code;
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 [Dij79]. 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 15.1 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):
        V(&r_sema);
    elif ((r_entered + w_entered) == 0) and (w_waiting > 0):
        V(&w_sema);
    else:
        V(&mutex);
    ;
;
def acquire_rlock():
    P(&mutex);
    if w_entered > 0:
        r_waiting = r_waiting + 1;
        V(&mutex);
        P(&r_sema);
        r_waiting = r_waiting - 1;
    ;
    r_entered = r_entered + 1;
    V_one();
;
def release_rlock():
    P(&mutex);
    r_entered = r_entered - 1;
    V_one();
;
def acquire_wlock():
    P(&mutex);
    if (r_entered + w_entered) > 0:
        w_waiting = w_waiting + 1;
        V(&mutex);
        P(&w_sema);
        w_waiting = w_waiting - 1;
    ;
    w_entered = w_entered + 1;
    V_one();
;
def release_wlock():
    P(&mutex);
    w_entered = w_entered - 1;
    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 15.1: [RWsbs.cxl] 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 16

# Starvation

A *property* is a set of traces. If a program has a certain property, that means that the traces that that program allows are a subset of the traces in the property. 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 6) 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. 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 11.1 fair. However, the split binary semaphore solution of Figure 15.1 provides much control over whom to pass baton to. Figure 16.1 contains a version of Figure 15.1 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;

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

Figure 16.1: [RWfair.cxl] Reader/Writer Lock implementation addressing fairness.

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 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 17

# 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* [Hoa74]. (A similar construct was independently invented by Per Brinch Hansen [BH73].) 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      V(&mutex);
3      P(c);
4  ;
5  def signal(c):
6      V(c);
7      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 [LR80]. 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 17.1 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.

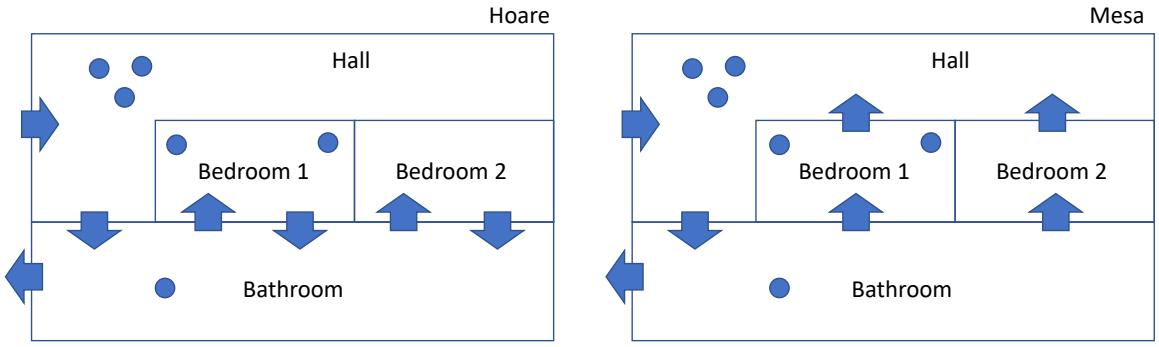


Figure 17.1: High-level depictions of Hoare and Mesa monitors. The hall is where processes wait to enter the bathroom (the critical section). The bedrooms illustrate two condition variables. The circles are processes.

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" or "Mesa condition variable" semantics have become more popular, adopted by all major programming languages.

That said, few programming languages provide full support for monitors. In Java, each object has a hidden lock *and* a hidden condition variable associated with it. Methods declared with the `synchronized` keyword automatically obtain the lock. Java objects also support `wait`, `notify`, and `notifyAll`. Java also supports explicit allocations of locks and condition variables. In Python, locks and condition variables must be explicitly declared. The `with` statement makes it easy to acquire and release a lock for a section of code. In C and C++, support for locks and condition variables is through libraries.

# Chapter 18

# 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 18.1 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 18.2 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 17: 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`. Many implementation of condition variables depend on this, and optimizations of the code allow so-called "spurious wakeups," where `wait` may sometimes return even if it has not been notified.

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 15.1), 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.

Andrew Birrell's paper on Programming with Threads gives an excellent introduction to working with Mesa-style condition variables [Bir89].

```
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()):
        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():
            bagRemove(&(^c).waiters, bagChoose(waiters));
        ;
    ;
;
def notifyAll(c):
    (^c).waiters = bagEmpty();
;
```

Figure 18.1: Implementation of condition variables in the `synch` module.

```
import synch;

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

Figure 18.2: [RWcv.cxl] Reader/Writer Lock implementation using Mesa-style condition variables.

# Chapter 19

# Deadlock Prevention

As introduced in Chapter 12, deadlock is the situation when there are a collection of processes waiting for one another to release one or more resources. There are four conditions that must hold for deadlock to occur [CES71]:

1. *Mutual Exclusion*: each resource can only be used by one process at a time:
2. *Hold and Wait*: each process holds resources it already allocated while it waits for other resources that it needs;
3. *No Preemption*: Resources cannot be forcibly taken back from processes that allocated them;
4. *Circular Wait*: There exists a directed circular chain of processes, each waiting to allocate a resource held by the next.

Preventing deadlock thus means preventing that one of these conditions occurs. However, mutual exclusion is not easily prevented (although, for some resources it is possible, as demonstrated in Chapter 21). Havender [Hav68] proposed the following techniques that avoid the remaining three conditions:

- *No Hold and Wait*: a process must request all resources it is going to need at the same time;
- *Preemption*: if a process is denied a request for a resource, it must release all resources that it has already acquired and start over;
- *No Circular Wait*: define an ordering on all resources and allocate resources in a particular order.

To implement a *No Hold and Wait* solution, a philosopher would need a way to lock both the left and right forks at the same time. Locks do not have such an ability, and neither do semaphores. so we re-implement the Dining Philosophers using condition variables that allow one to wait for arbitrary application-specific conditions.

Figure 19.1 demonstrates how this might be done. We use a single mutex for the diners, and, for each fork, a boolean and a condition variable. The boolean indicates if the fork has been taken. Each diner waits if either the left or right fork is already taken. But which condition variable to wait on? The code demonstrates an important technique to use when waiting for multiple conditions. The condition in the `while` statement is the condition that the diner is waiting for to be `False`. Within the `while` statement, there is an `if` statement for each disjunct of that condition. The code waits for either or both forks if necessary. After that, it goes back to the top of the `while` loop.

A common mistake is to write the following code instead:

```
import synch;

const N = 5;

def diner(which):
    let left = which, right = (which % N) + 1:
        while choose({ False, True }):
            lock(&mutex);
            while forks[left] or forks[right]:
                if forks[left]:
                    wait(&conds[left]);
                ;
                if forks[right]:
                    wait(&conds[right]);
                ;
            ;
            assert not (forks[left] or forks[right]);
            forks[left] = True;
            forks[right] = True;
            unlock(&mutex);

            # dine

            lock(&mutex);
            forks[left] = False;
            forks[right] = False;
            notify(&conds[left]);
            notify(&conds[right]);
            unlock(&mutex);

            # think
        ;
    ;
;
mutex = Lock();
forks = dict{ False for i in 1..N };
conds = dict{ Condition(&mutex) for i in 1..N };
for i in 1..N:
    spawn diner(i);
;
```

Figure 19.1: [DinersCV.cxl] Dining Philosophers that grab both forks at the same time.

```
while forks[left]:
    wait(&conds[left]);
;
while forks[right]:
    wait(&conds[right]);
;
```

- Can you see why this does not work? What can go wrong?
- Run it through CXL in case you are not sure!

The `Preemption` approach suggested by Havender is to allow processes to back out. While this could be done, this invariably leads to a busy waiting solution where a process keeps obtaining locks and releasing them again until it finally is able to get all of them. The *No Circular Waiting* approach to order the resources was already presented in Chapter 12

There is, however, another approach. which is sometimes called *deadlock avoidance*. . In the case of the Dining Philosophers, we want to avoid the situation where each diner picks up a fork. So if we can prevent more than four diners from starting to eat at the same time, then we can avoid the conditions for deadlock from ever happening. Figure 19.2 demonstrates this concept. It uses a semaphore to restrict the number of diners at any time to four.

This concept can be generalized using something called the Banker's Algorithm [Dij64], but it is outside the scope of this book. The problem with these kinds of schemes is that one needs to know ahead of time what the maximum number of resources is that each process wants to allocate, making them generally quite impractical.

```
import synch;

const N = 5;

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

Figure 19.2: [DinersAvoid.cxl] Dining Philosophers that carefully avoid getting into a deadlock scenario.

# Chapter 20

# Actors and Synchronized Queues

Some programming languages favor a completely different way of implementing synchronization using so-called *actors* [HBS73]. Actors are processes that have only private memory and communicate through message queues. See Figure 20.1 for an illustration. Given that there is no shared memory in the actor model (other than the message queues, which have built-in synchronization), there is no need for critical sections. Instead, some sequential process owns a particular piece of data and other processes access it by sending request messages to the process and optionally waiting for response messages. Each process handles one message at a time, serializing all access to the data it owns.

The actor synchronization model is popular in a variety of programming languages, including Erlang and Scala. Actor support is also available through popular libraries such as Akka, which is available for various programming languages. In Python, Java, and C/C++, actors can be easily emulated using threads and *synchronized queues* (aka *blocking queues*) for messaging. Each thread would have one such queue for receiving messages. Dequeuing from an empty synchronized queue blocks the thread until another thread enqueues a message on the queue.

The `synch` library supports a synchronized message queue, similar to the `Queue` object in Python. Its interface is as follows:

- `Queue()` returns a new message queue;
- `enqueue(q, item)` add `item` to the queue pointed to by `q`;
- `dequeue(q)` waits for and returns an item on the queue pointed to by `q`.

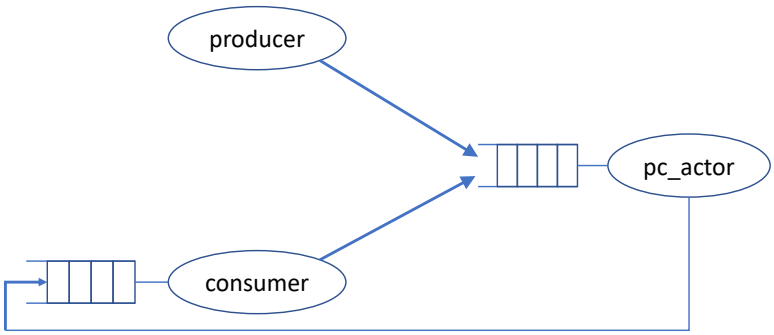


Figure 20.1: Depiction of three actors. The producer does not receive messages.

```
import synch;

def pc_actor(q, nrequests):
    let requests = {}, balance = 0:
        while nrequests > 0:
            let req = dequeue(q):
                if req.type == .produce:
                    if balance >= 0:
                        requests = requests + { req };
                    else:
                        let r = choose(requests):
                            assert r.type == .consume;
                            enqueue(r.queue, req.item);
                            requests = requests - { r };
                        ;
                    ;
                    balance = balance + 1;
                else:
                    assert req.type == .consume;
                    if balance <= 0:
                        requests = requests + { req };
                    else:
                        let r = choose(requests):
                            assert r.type == .produce;
                            enqueue(req.queue, r.item);
                            requests = requests - { r };
                        ;
                    ;
                    balance = balance - 1;
                ;
            ;
            nrequests = nrequests - 1;
        ;
    ;
;
def produce(q, item):
    enqueue(&pc_queue, dict{ .type: .produce, .item: item });
;
def consume(q1, q2):
    enqueue(q1, dict{ .type: .consume, .queue: q2 });
    result = dequeue(q2);
;
```

Figure 20.2: [actor.cxl] A producer/consumer actor.

```
const NITEMS = 2;

pc_queue = Queue();
spawn pc_actor(&pc_queue, 2 * NITEMS);
queues = [ Queue() for i in 0..(NITEMS - 1) ];
for i in 0..(NITEMS-1):
    spawn produce(&pcqueue, i);
    spawn consume(&pcqueue, &queues[i]);
;
```

Figure 20.3: [actortest.cxl] Test code for producer/consumer actor.

Note that a `Queue` behaves much like a zero-initialized semaphore. `enqueue` is much like `V`, except that it is accompanied by data. `dequeue` is much like `P`, except that it also returns data. Thus, synchronized queues can be considered a generalization of counting semaphores.

Figure 20.2 shows a CXL solution to the producer/consumer problem using an actor (as illustrated in Figure 20.1. `pc_actor` is essentially a matchmaker: it matches `.produce` requests with `.consume` requests. Sometimes it may have buffered `.consume` requests and is waiting for `.produce` requests, and sometimes it is vice versa. The variable `balance` specifies what the situation is: if it is positive then it has buffered `.produce` requests; if it is negative then it has buffered `.consume` requests. Note that both `balance` and `requests` are variables that are local to the `pc_actor` process.

Each request message is a dictionary with two fields. One of the fields is `.type`, which either is `.produce` or `.consume`. In case of `.produce`, the second field is `.item` and holds the data that is produced. In case of `.consume`, the second field is `.queue` and holds a pointer to the queue where the response message must be enqueued. `.produce` messages do not get a response.

The `pc_actor` process has two arguments. `q` is a pointer to the queue on which it receives messages. `nrequests` is the total number of requests that it will handle. We need the latter because in CXL all processes are required to terminate. In a more realistic implementation, the `pc_actor` would be in an infinite loop awaiting requests.

The `produce` method takes two arguments: a pointer to the queue of the `pc_actor` and the item. The `consume` method takes two queue pointer arguments. The first is a pointer to the queue of the `pc_actor`. The second is a pointer to the queue of the invoker: it is the queue to which the response must be posted.

Figure 20.3 shows how this code may be used. It spawns `NITEMS` producer and consumer processes. Note that in that case the `pc_actor` is expected to receive `2 * NITEMS` requests.

# Chapter 21

# Non-Blocking Synchronization

So far we have concentrated on critical sections to synchronize multiple processes. Certainly, preventing multiple processes from accessing certain code at the same time simplifies how to think about synchronization. However, it can lead to starvation. Even in the absence of starvation, if some process is slow for some reason while being in the critical section, the other processes have to wait for it to finish the critical section. In this chapter, we will have a brief look at how one can develop concurrent code in which processes never have to wait for other processes. For this, we will revisit the producer/consumer problem, in which a set of processes are producing items and a set of processes are consuming items.

The code in Figure 21.1 is based on code developed by Herlihy and Wing [HW87]. The code is a "proof of existence" for non-blocking synchronization; it is not necessarily practical. There are two variables. `items` is an unbounded array with each entry initialized to `()`. `back` is an index into the array and points to the next slot where a new value is inserted. The code uses two interlock instructions:

- `inc(p)`: atomically increments `^p` and returns the old value;
- `swap(p)`: sets `^p` to `()` and returns the old value.

Method `produce(item)` uses `inc &(back)` to allocate the next available slot in the `items` array. It stores the item as a singleton tuple. Method `consume()` repeatedly scans the array, up to the `back` index, trying to find an item to return. To check an entry, it uses `swap()` to atomically remove an item from a slot if there is one. This way, if two or more processes attempt to extract an item from the same slot, at most one will succeed.

There is no critical section. If one process is executing instructions very slowly, this does not negatively impact the other processes, as it would with solutions based on critical sections. On the contrary, it helps them because it creates less contention.

The solution is not practical because the `items` array must be unbounded if an unbounded number of items may be produced. Each slot in the array is only used once, which is inefficient. Also, the `inc` and `swap` interlock instructions are not universally available on existing hardware. However, in the literature there are many examples of practical non-blocking (aka *wait-free*) synchronization algorithms [].

```
const MAX_ITEMS = 3;

def inc(pcnt):
    atomic:
        result = ^pcnt;
        ^pcnt = (^pcnt) + 1;
    ;
;
def swap(pv):
    atomic:
        result = ^pv;
        ^pv = ();
    ;
;
def produce(item):
    items[inc(&back)] = (item,);
;
def consume():
    result = ();
    while result == ():
        let range = back, i = 0:
            while (i < range) and (result == ()):
                result = swap(&items[i]);
                i = i + 1;
            ;
        ;
    ;
    result = result[0];     # convert (item,) into item
;
back = 0;
items = [ () for i in 0..MAX_ITEMS ];

for i in 1..MAX_ITEMS:
    spawn produce(i);
;
for i in 1..choose(0..MAX_ITEMS):
    spawn consume();
;
```

Figure 21.1: [hw.cxl] Non-blocking queue.

# Chapter 22

# Barrier Synchronization

Barrier synchronization is a problem that comes up in high-performance parallel computing, used, among others, for scalable simulation. Barrier synchronization is almost the opposite of critical sections: the intention is to get a group of processes to run some code at the same time. More precisely, the processes run in rounds. At the end of each round there is a so-called *barrier* where processes wait until all processes have completed the round. For example, in an iterative matrix algorithm, the matrix may be cut up into fragments. During a round, one process runs for each fragment concurrently. The next round is not allowed to start until all processes have completed operations on their fragment.

Figure 22.1 shows a high-level depiction with three processes. Initially all processes are in round 0. Then each process can start and finish the round. However, none of the processes can progress to the next round until all processes have finished the current round.

Counting semaphores work well for implementing barrier synchronization. Figure 22.2 shows an example. There is a 0-initialized semaphore for each of the `N` processes. Before process $i$ enters a round, it first sends a signal to every other process and then waits until it receives a signal from every other process. Process $i$ sends a signal to process $j$ by incrementing `sema[`$j$`]` (using `V`), while process $i$ waits for a signal be decrementing `sema[`$i$`]` (using `P`).

The `round` array is kept to check the correctness of this approach. Each process increments its entry every time it enters a round. If the algorithm is correct, it can never be that two processes are more than one round apart.

- See if you can implement barrier synchronization for $N$ processes with just three semaphores. Busy waiting is not allowed.

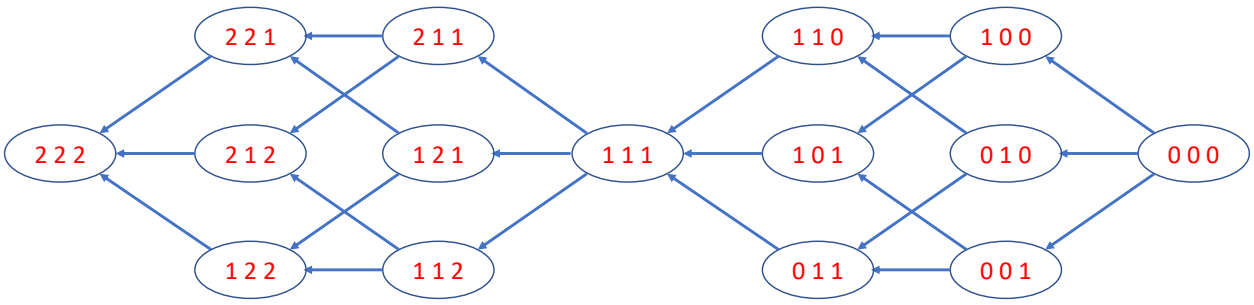


Figure 22.1: High-level state diagram specification of barrier synchronization with three processes and two rounds. The three numbers specify how many rounds each process has completed.

```
import synch;
import list;

const N = 3;        # number of processes
const R = 4;        # number of rounds

def process(self):
    for r in 1..R:
        for i in 0..(N-1):
            if i != self:
                V(&sema[i]);
            ;
        ;
        for i in 0..(N-1):
            if i != self:
                P(&sema[self]);
            ;
        ;
        round[self] = (round[self] + 1);
        assert (listMax(round) - listMin(round)) <= 1;
    ;
;
round = [ 0 for i in 0..(N-1) ];
sema = [ Semaphore(0) for i in 0..(N-1) ];
for i in 0..(N-1):
    spawn process(i);
;
```

Figure 22.2: [barrier.cxl] Barrier synchronization with semaphores.

- How about just two?

# Chapter 23

# Networking: Alternating Bit Protocol

```
def net_send(pchan, m, reliable):
    ^pchan = m if (reliable or choose({ False, True })) else ();
;
def net_recv(pchan):
    result = ^pchan;
;
def app_send(payload):
    s_seq = 1 - s_seq;
    let m = dict{ .seq: s_seq, .payload: payload }, blocked = True:
        while blocked:
            net_send(&s_chan, m, False);
            let response = net_recv(&r_chan):
                blocked = (response == ()) or (response.ack != s_seq);
            ;
        ;
    ;
;
def app_recv(reliable):
    r_seq = 1 - r_seq;
    let blocked = True:
        while blocked:
            let m = net_recv(&s_chan):
                if m != ():
                    net_send(&r_chan, dict{ .ack: m.seq }, reliable);
                    if m.seq == r_seq:
                        result = m.payload;
                        blocked = False;
                    ;
                ;
            ;
        ;
    ;
;
s_chan = ();
r_chan = ();
s_seq = 0;
r_seq = 0;
```

Figure 23.1: [abp.cxl] Alternating Bit Protocol.

```
const NMSGS = 5;

def sender():
    for i in 1..NMSGS:
        app_send(i);
    ;
;
def receiver():
    for i in 1..NMSGS:
        let payload = app_recv(i == NMSGS):
            assert payload == i;
        ;
    ;
;
spawn sender();
spawn receiver();
```

Figure 23.2: [abptest.cxl] Test code for alternating bit protocol.

# Bibliography


[BH73] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., USA, 1973.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. SRC reports 35, Digital Systems Research Center, Palo Alto, CA, USA, January 1989.

[CES71] Edward G. Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[CHP71] Pierre-Jacques. Courtois, Frans Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, October 1971.

[Dij62] Edsger W. Dijkstra. EWD-35: Over de sequentialiteit van procesbeschrijvingen. circulated privately, approx. 1962.

[Dij64] Edsger W. Dijkstra. EWD-108: Een algorithme ter voorkoming van de dodelijke omarming. circulated privately, approx. 1964.

[Dij65] Edsger W. Dijkstra. EWD-123: Cooperating Sequential Processes. circulated privately, 1965.

[Dij79] Edsger W. Dijkstra. EWD-703: A tutorial on the split binary semaphore. circulated privately, March 1979.

[Dow09] Allen B. Downey. *The Little Book Of Semaphores*. Green Tea Press, 2009.

[Hav68] James W. Havender. Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7(2):74–84, June 1968.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[Hoa73] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*. Academic Press, New York, NY, 1973.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 13–26, New York, NY, USA, 1987. Association for Computing Machinery.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[Lam09] Leslie Lamport. The PlusCal Algorithm Language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009*, pages 36–60, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, February 1980.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115 – 116, 1981.

[Sch97] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, Berlin, Heidelberg, 1997.

# Appendix A

# List of Values

Chapter 4 provides an introduction to CXL values. Below is a complete list of CXL value types with examples:

| | |
|---|---|
| Boolean | `False`, `True` |
| integers | `..., -2, -1, 0, 1, 2, ...` |
| Atom | `.example`, `.test1` |
| Program Counter | (method names are program counter constants) |
| Dictionary | `dict{ .account: 12345, .valid: False }` |
| Set | `{ 1, 2, 3 }, { False, .id, 3 }` |
| Address | `&lock, &flags[2]` |
| Context | (generated by `stop` expression) |

Tuples, lists, strings, and bags are all special cases of dictionaries. Both tuples and lists map indexes (starting at 0) to CXL values. Their format is either `(e, e, ..., e,)` or `[e, e, ..., e,]`. If the tuple or list has two or more elements, then the final comma is optional. A string must be enclosed by double quotes and is represented as a tuple of its characters. Characters are one-character atoms. A bag or multiset is a dictionary that maps a value to how many times it occurs in the bag.

All CXL values are ordered with respect to one another. First they are ordered by type according to the table above. So, for example, `True < 0 < .xyz < { 0 })`. Within types, the following rules apply:

- `False < True`;
- integers are ordered in the natural way;
- atoms are lexicographically ordered;
- program counters are ordered by their integer values;
- dictionaries are first converted into a list of ordered (key, value) pairs. Then two dictionaries are lexicographically ordered by this representation;
- a set is first converted into an ordered list, then lexicographically ordered;
- an address is a list of atoms. Addresses are lexicographically ordered accordingly;
- contexts (Appendix F) are ordered by hash.

# Appendix B

# List of Operators

CXL currently supports the following operators:

| | |
|---|---|
| `e == e` | two CXL values of the same type |
| `e != e` | two CXL values of the same type |
| `e < e` | any two CXL values |
| `e <= e` | any two CXL values |
| `e > e` | any two CXL values |
| `e >= e` | any two CXL values |
| `e and e and ...` | two or more Booleans |
| `e or e or ...` | two or more Booleans |
| `not e` | a Boolean |
| `-e` | an integer |
| `e + e + ...` | two or more integers, sets, or lists |
| `e - e` | two integers or sets |
| `e * e * ...` | two or more integers or sets |
| `e / e` | two integers (integer division) |
| `e % e` | two integers (division remainder) |
| `e .. e` | two integers |
| `e in e` | first is any CXL value, second is set |
| `e not in e` | first is any CXL value, second is set |
| `e if e else e` | middle `e` must be a Boolean |
| `^e` | an address |
| `choose e` | a set |
| `min e` | a set |
| `max e` | a set |
| `cardinality e` | a set |
| `keys e` | a dictionary |
| `len e` | a dictionary (includes tuple or list) |
| `bagsize e` | a bag |
| `atLabel e` | atom (corresponding to a label) |
| `nametag e` | `e` must be `()` |
| `processes e` | `e` must be `()` |
| `&lv` | `lv` is an lvalue |
| `stop lv` | `lv` is an lvalue |

`+` computes the union when applied to sets and the concatenation when applied to lists or tuples. `-` computes set difference when applied to two sets. `*` computes the intersection when applied to sets. `x..y` computes the set consisting of the integers `x` through `y` inclusive. It returns the empty set if `x > y`.

An *lvalue* (short for left hand value of an assignment statement) is something that can be assigned. This can be a shared variable, a process variable, or a dereferenced pointer variable. It can also be indexed. Examples include `var`, `var[e]`, and `^p`.

CXL also supports the following comprehensions:

| | |
|---|---|
| Set comprehension | `{ f(e) for e in s }` |
| List comprehension | `[ f(e) for e in s ]` |
| Dict comprehension | `dict{ f(e) for e in s }` |

In each case, `s` must be a set. For list comprehensions, the list will be sorted by `e`. For dictionary comprehensions, each `e` will be mapped to `f(e)`.

# Appendix C

# List of Statements

CXL currently supports the following statements:

| | |
|---|---|
| `lv = e` | `lv` is an lvalue and `e` is an expression |
| `pass` | do nothing |
| `del lv` | delete |
| `assert b [, x]` | `b` is a Boolean. If `False`, optionally print expression `e` |
| `const v = e` | `v` is an identifier, `e` is a constant expression |
| `def m(v, ...): S` | `m` and `v` are identifiers, `S` a list of statements |
| `let v = e, ...: S` | `e` is an expression, `S` a list of statements |
| `if b: S` | `b` is a Boolean, `S` a list of statements |
| `while b: S` | `b` is a Boolean, `S` a list of statements |
| `for v in e: S` | `e` is a set evaluated in order, `S` a list of statements |
| `atomic: S` | `S` a list of statements |
| `spawn p e [, t]` | `p` is a method, `e` is an expression, `t` is an optional tag (an expression) |
| `go c e` | `c` is a context, `e` is an expression |
| `import m` | `m` identifies a module |

CXL does not support Python-style iterators. To iterate through a dictionary, list, or tuple, use: `for v in keys(e): S`.

# Appendix D

# List of Modules

The `bag` module has various useful methods that operate on bags or multisets:

| | |
|---|---|
| `bagEmpty()` | returns an empty bag |
| `bagFromset(s)` | create a bag from set `s` |
| `bagCount(b, e)` | count how many times `e` occurs in bag `b` |
| `bagChoose(b)` | like `choose(s)`, but applied to a bag |
| `bagAdd(pb, e)` | add one copy of `e` to bag `^pb` |
| `bagRemove(pb, e)` | remove one copy of `e` from bag `^pb` |

The `list` module has various useful methods that operate on lists or tuples:

| | |
|---|---|
| `subseq(t, b, f)` | return a *slice* of list `t` starting at index `b` and ending just before index `f` |
| `append(t, e)` | append `e` to list `t` |
| `head(t)` | return the first element of list `t` |
| `tail(t)` | return all but the first element of list `t` |
| `qsort(t)` | sort list `t` using quicksort |
| `list2bag(t)` | convert list `t` into a bag |
| `list2set(t)` | convert list `t` into a set |
| `listMin(t)` | return the minimum of all elements in `t` |
| `listMax(t)` | return the maximum of all elements in `t` |
| `listSum(t)` | return the sum of all elements in `t` |

# Appendix E

# List of Machine Instructions

| | |
|---|---|
| Address | compute address |
| Apply | pop $m$ and $i$ and apply $i$ to $m$, pushing a value |
| Assert | pop $b$ and check that it is `True` |
| AtomicInc | increment the atomic counter of this context |
| AtomicDec | decrement the atomic counter of this context |
| Continue | no-op (but causes a context switch) |
| Choose | choose an element from the set on top of the stack |
| Del | delete a shared variable |
| DelVar $v$ | delete process variable $v$ |
| Dict | pop $n$ and $n$ key/value pairs and push a dictionary |
| Dup | duplicate the top element of the stack |
| Frame $m(a)$ end=$r$ | start method $m$ with arguments $a$, initializing variables. Last instruction is at $r$ |
| Go | pop context and value, push value on context's stack, and add to context bag |
| Jump $p$ | set program counter to $p$ |
| JumpCond $e$ $p$ | pop expression and, if equal to $e$, set program counter to $p$ |
| Load $v$ | push shared variable $v$ (`^` is indirect) onto the stack |
| LoadVar $v$ | load process variable $v$ and push it onto the stack |
| $n$-ary $op$ | apply $n$-ary operator $op$ to the top $n$ elements on the stack |
| Pop | pop a value of the stack and discard it |
| Push $c$ | push constant $c$ onto the stack |
| PushAddress $v$ | push the address of shared variable $v$ |
| Return | pop return address, push `result`, and restore program counter |
| Set | pop $n$ and $n$ elements and push a set of the elements |
| Spawn | pop tag, argument, and method and spawn a new context |
| Split | pop set and push minimum and remainder |
| Stop | save context into shared variable and remove from context bag |
| Store $v$ [$n$] | pop a value from the stack and store it in shared variable $v$ (`^` is indirect) |
| StoreVar $v$ | pop a value from the stack and store it in process variable $v$ |

Clarifications:

- The effect of the `Apply` instructions depends much on $m$. If $m$ is a dictionary, then `Apply` finds $i$ in the dictionary and pushes the value. If $m$ is a program counter, then `Apply` invokes method $m$ by pushing the current program counter and setting the program counter to $m$. $m$ is supposed to leave the result on the stack.

- The `Frame` instruction pushes the value of the process register (*i.e.*, the values of the process variables) onto the stack. It initializes the `result` variable to the empty dictionary. The `Return` instruction restores the process register by popping its value of the stack.
- All method calls have exactly one argument, although it sometimes appears otherwise:
  - `m()` invokes method `m` with the empty dictionary `()` as argument;
  - `m(a)` invokes method `m` with argument `a`;
  - `m(a, b, c)` invokes method `m` with tuple `(a, b, c)` as argument.

  The `Frame` instruction unpacks the argument to the method and places them into process variables by the given names.
- Every `Stop` instruction must immediately be followed by a `Continue` instruction.

# Appendix F

# Contexts and Processes

A context captures the state of a process. Each time the process executes an instruction, it goes from one context to another. All instructions update the program counter, and so no instruction leaves the context the same. There may be multiple processes with the same state at the same time.

A context consists of the following:

| | |
|---|---|
| name tag | a dictionary with atoms `.name` and `.tag` |
| program counter | a Program Counter value pointing into the code |
| end | a Program Counter value indicating the end of the process code |
| atomic | if non-zero, the process is in atomic mode |
| stack | a list of CXL values |
| register | a CXL dictionary mapping atoms (representing process variables) to CXL values |
| stopped | a Boolean indicating if the context is stopped |

Details:

- The name in a name tag is the name of the method that the process is executing;
- The tag in a name tag is its argument by default. Optionally `spawn` allows the tag to be specified explicitly;
- A process terminates when it reaches the `Return` instruction of the method or when it hits a fault. Faults include divide by zero, reading a non-existent key in a dictionary, accessing a non-existent variable, as well as when an assertion fails;
- The execution of a process in atomic mode does not get interleaved with that of other processes.
- The register of a process always contains a dictionary, mapping atoms to arbitrary values. The atoms correspond to the variable names in a CXL program.

A context is considered *blocked* if there does not exist a sequence instructions that the process can execute that either writes to a shared variable (even if that leaves the shared variable unchanged) or terminates the process.

# Appendix G

# The CXL Virtual Machine

The CXL Virtual Machine (CVM) has the following state:

| | |
|---|---|
| code | a list of CVM machine instructions |
| labels | a dictionary of atoms to program counters |
| variables | a dictionnary mapping atoms to values |
| ctxbag | a bag of non-stopped contexts |
| stopbag | a bag of stopped contexts |
| choosing | if not `None`, indicates a context that is choosing |
| failure | Boolean indicating some context has failed |
| initializing | Boolean indicating that the state is not fully initialized. |

There is initially a single context with nametag `__init__/()` and program counter 0. It starts executing in atomic mode until it finishes executing the last instruction in the code. All states until then are `initializing` states. Other processes, created through `spawn` statements, do not start executing until then.

A *micro step* is the execution of a single CVM machine instruction by a context. Each micro step generates a new state. When there are multiple contexts, the CVM can interleave them. However, trying to interleave every microstep would be needlessly expensive, as many micro steps involve changes to a context that are invisible to other contexts.

A *macro step* can involve multiple micro steps. The following instructions start a new macro step: `Load`, `Store`, `AtomicInc`, and `Continue`. The CVM interleaves macro steps, not micro steps. Like micro steps, each macro step involves a single context. Unlike a micro step, a macro step can leave the state unchanged.

Executing a CXL program results in a graph where the nodes are CXL states and the edges are macro steps. When a state is `choosing`, the edges from that state are by a single context, one for each choice. If not, the edges from the state are one per context. A terminating state is a state with an empty context bag or a state where all contexts are blocked.

# Index

# Glossary

**actor model** is a concurrency model where there are no shared variables, only processes with private variables and exchanging messages. 68

**atomicity** describes that a certain machine instruction or sequence of machine instructions by a process is executed indivisibly and cannot be interleaved with machine instructions of another process. 3

**barrier synchronization** is when a set of processes execute in rounds, waiting for one another to complete each round. 73

**blocked process** is a process cannot make progress without the help of another process. For example, a process that is waiting for a spinlock to become available. 32

**busy waiting** (aka spin-waiting) is when a process waits in a loop for some application-defined condition while changing the state, for example by continually acquiring and releasing a lock. 37

**concurrent execution** (aka parallel execution) is when there are multiple processes executing and their machine instructions are interleaved in an unpredictable manner. 3

**condition variable** a variable that keeps track of which processes are waiting for a specific application-level condition. The variable can be waited on as well as signaled or notified. 59

**conditional critical section** is a critical section with, besides mutual exclusion, additional conditions on when a process is allowed to enter the critical section. 53

**context** (aka continuation) describes the state of a running process, including its program counter, the values of its variables (stored in its register), and the contents of its stack. 14

**critical section** (aka critical region) is a set of instructions that only one process is allowed to execute at a time. The instructions are, however, not executed atomically, as other processes can continue to execute and access shared variables. 16

**deadlock** is when there are two or more processes waiting indefinitely for one another to release a resource. 45

**determinism** is when the outcome of an execution is uniquely determined by the initial state. 3

**fairness** is when each process eventually can access each resource it needs to access with high probability. 56

**interlock instruction** a machine instruction that involves multiple memory load and/or store operations, executed atomically. 28

**invariant** is a binary predicate over states that must hold for every reachable state of a process. 4

**lock** an object that can be owned by at most one process at a time. Useful for implementing mutual exclusion. 17

**machine instruction** is an atomic operation on the CXL virtual machine, executed by a process. 8

**model checking** is a formal verification method that explores all possible executions of a program, which must have a finite number of states. 3

**monitor** is a programming language paradigm that supports mutual exclusion as well as waiting for resources to become available. 59

**mutual exclusion** is a property that two processes never enter the same critical section. 16

**non-blocking synchronization** (aka wait-free synchronization) is when access to a shared resource can be guaranteed in a bounded number of steps even if other processes are not making progress. 71

**process** is a method in execution. We do not make the distinction between processes and threads. A process has a program counter, a register, and a stack. 8

**process variable** is a variable that is private to a single process and stored in its register. 21

**producer/consumer problem** is a synchronization problem whereby one or more producing processes submit items and one or more consuming process want to receive them. No item can get lost or forged or be delivered to more than one consumer, and producers and consumers should block if resources are exhausted. 49

**property** describes a set of execution traces or histories that are allowed by a program. Safety properties are properties in which "no bad things happen," such as violating mutual exclusion in a critical section. Liveness properties are properties where "something good eventually happens," like processes being able to enter the critical section if they want to. 56

**race condition** describes when multiple processes access shared state concurrently, leading to undesirable outcomes. 9

**reader/writer lock** is a lock on a resource that can be held by multiple processes if they all only read the resource. 37

**semaphore** is a counter that can be atomically incremented and decremented, but blocks the process until the counter is larger than zero first. 47

**sequential execution** is when there is just one process executing, as opposed to concurrent execution. 3

**shared variable** is a variable that is stored in the memory of the CXL virtual machine and shared between multiple processes, as opposed to a process variable. 3

**spinlock** is an implementation of a lock whereby a process loops until the lock is available, at which point the process atomically obtains the lock. 28

**stack machine** is a model of computing where the state of a process is kept on a stack. CXL uses a combination of a stack machine and a register-based machine. 14

**starvation** is when a process cannot make progress because it is continuously losing a competition with other processes to get access to a resource. 56

**state** an assignment of values to variables. In a CXL virtual machine, this includes the contents of its shared memory and the set of contexts. 21

**step** is the execution of a machine instruction by a process, updating its state. CXL distinguishes micro steps (machine instructions) and macro step (a sequence of instructions with at most one effect visible by other processes). 21

**thread safety** is when the implementation of a data structure allows concurrent access with well-defined semantics. 16

**trace** is a sequence of steps, starting from an initial state. An infinite trace is also called a *behavior*. 21