Concurrent Programming with Harmony

Robbert van Renesse

August 30, 2020

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons AttributionNonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) at http://creativecommons.org/licenses/by-nc-sa/4.0.

Contents

1 On Concurrent Programming 4

2 Introduction to Harmony 7

3 The Problem of Concurrent Programming 9

4 The Harmony Virtual Machine 14

5 Critical Sections 21

6 Peterson's Algorithm 28

7 Harmony Methods and Pointers 35

8 Spinlock 38

9 Locks and Blocking 43

10 Concurrent Data Structures 49

11 Reader/Writer Locks using Busy Waiting 54

12 Reader/Writer Locks with Blocking 59

13 Semaphores 63

14 Bounded Buffer 68

15 Split Binary Semaphores 74

16 Starvation 78

17 Monitors 83

18 Deadlock 89

19 Actors and Message Passing 96

20 Interrupts **100**

21 Alternating Bit Protocol **107**

22 Non-Blocking Synchronization **110**

23 Barrier Synchronization **115**

Bibliography **118**

A List of Values **120**

B List of Operators **122**

C List of Statements **124**

D List of Modules **125**
D.1 The `alloc` module . . . 125
D.2 The `bag` module . . . 125
D.3 The `list` module . . . 125
D.4 The `set` module . . . 126
D.5 The `synch` module . . . 126

E List of Machine Instructions **127**

F Contexts and Processes **129**

G The Harmony Virtual Machine **131**

H Harmony Language Details **133**
H.1 Harmony is not object-oriented . . . 133
H.2 Constants, Global and Local Variables . . . 134
H.3 Operator Precedence . . . 134
H.4 Tuples, Lists, and Pattern Matching . . . 134
H.5 For Loops and Comprehensions . . . 135
H.6 Dynamic Allocation . . . 136
H.7 Comments . . . 139

Acknowledgments **140**

Index **141**

Glossary **143**

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. On the other hand, 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* (indivisibly) 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 Harmony that helps with *testing* concurrent code. The approach is based on *model checking*: instead of relying on luck, Harmony will run *all possible executions* of

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

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 the bug 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, the test program needs to be simple. Otherwise it may take longer than we 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, let us consider a sorting algorithm. Now suppose we create a test program, a model, that tries sorting *all* lists of up to five numbers chosen from the set { 1, 2, 3, 4, 5 }. Model checking proves that for those particular scenarios the sorting algorithm works: the output is a sorted permutation of the input. In some sense it is an excellent test: it will have considered all *corner cases*, including lists where all numbers are the same, lists that are already sorted or reversely sorted, etc. If there is a bug in the sorting algorithm, most likely it would be triggered and the model checker would produce a scenario that would make it easy to find the source of the bug. However, if the model checker does not find any bugs, we do not know for sure that the algorithm works for lists with more than five numbers or for lists that have values other than the numbers 1 through 5. Still, we would expect that the likelihood that there are bugs remaining in the sorting algorithm is small. Note, however, that it would be easy to write a program that sorts all lists of up to five numbers correctly but fails to do so for a list of 6 numbers. (Hint: simply use an **if** statement.)

While model checking does not in general prove an algorithm correct, it can help with proving an algorithm correct. The reason is that the correctness of many programs is built around *invariants*: predicates that must hold for every state in the execution of a program. A model checker can find violations of proposed invariants when evaluating a model and provide valuable early feedback to somebody who is trying to construct a proof, even an informal one. We will include examples of such invariants as they often provide excellent insight into why a particular algorithm works.

So what is Harmony? Harmony 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. Harmony programs are not intended to be "run" like programs in most other programming languages—instead Harmony programs are model checked to test that the program has certain desirable properties and does not suffer from bugs.

The syntax and semantics of Harmony 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 understands 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.

Harmony is heavily influenced by Leslie Lamport's work on TLA+, TLC, and PlusCal [Lam02, Lam09]. Harmony 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]. (This chapter is named after that book.)

The book proceeds as follows:

- Chapter 2 introduces the Harmony 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 Harmony 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 Harmony language needed for the rest of the book.
- Chapter 8 introduces "hardware" *locks* for implemented critical sections.
- Chapter 9 presents Harmony modules that supports locks and other synchronization primitives.
- Chapter 10 provides an introduction to building concurrent data structures.
- Chapter 11 and Chapter 12 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 11 also discusses *busy-waiting*, which is an easy but undesirable approach to synchronize processes.
- 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.
- Chapter 14 and Chapter 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.
- Chapter 17 presents *monitors* and *condition variables*, another approach to process synchronication.
- Chapter 18 describes *deadlock* where a set of processes are indefinitely waiting for one another to release a resource.
- Chapter 19 presents the *actor model* and *message passing* as an approach to synchronization.
- Chapter 20 discusses how to handle interrupts to a process, a problem closely related to synchronization among multiple processes.
- Chapter 21 presents a problem and a solution to the distributed systems problem of having two processes communicate reliably over an unreliable network.
- Chapter 22 introduces *non-blocking* or *wait-free* synchronization algorithms, which prevent processes waiting for one another more than a bounded number of steps.
- Chapter 23 describes *barrier synchronization*, useful in high-performance computing applications such as parallel simulations.

Chapter 2

Introduction to Harmony

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

- Every statement in Harmony must be terminated by a semicolon (even **def** statements). In Python semicolons are optional and rarely used. There is no syntactic significance to indentation in Harmony. Correct indentation in Harmony is encouraged but not enforced. Harmony only supports basic operator precedence or associativity. Use parentheses liberally to remove ambiguity.
- Harmony does not (currently) support floating point, iterators, or I/O; Harmony does support **for** loops and various "comprehensions."
- Python is object-oriented, supporting classes with methods and inheritance; Harmony has objects but does not support classes. Harmony supports pointers to objects and methods.

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

Figure 2.1 shows a simple example of a Harmony 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 Harmony). The method also has a variable called n containing the value of the argument. The $\{ x..y \}$ notation generates a set containing the numbers from x to y (inclusive). The last two lines in the program are the most interesting. The first assigns to x some unspecified value in the range `0..N` and the second verifies that `triangle`(x) equals $x(x+1)/2$.

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

```
const N = 10;

def triangle(n):   # computes the n'th triangle number
    result = 0;
    for i in {1..n}:      # for each integer from 1 to n inclusive
        result += i;    # add i to result
    ;
;
x = choose({0..N});       # select an x between 0 and N inclusive
assert triangle(x) == ((x * (x + 1)) / 2);
```

Figure 2.1: [code/triangle.hny] Computing triangle numbers.

```
$ harmony triangle.hny
#states = 13 diameter = 1
#components: 13
no issues found
Open file:///.../harmony.html for more information
$
```

(For this to work, make sure `harmony` 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 Harmony is the size of the "state graph" (13 states in this case). If not, Harmony also reports what went wrong, typically by displaying a summary of an execution in which something went wrong.

In Harmony, 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:

```
$ harmony -c N=100 triangle.hny
#states = 103 diameter = 1
#components: 103
no issues found
Open file:///.../harmony.html for more information
$
```

Exercises

2.1 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. They will be explained in more detail in Chapter 4.)

2.2 Write a Harmony 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 all *done* flags are set to `True`. (**await** *c* is semantically the same as **while not** *c*: **pass**.) After that, it verifies that the value of *count* equals 2. Like Python, Harmony supports `assert` statements with two arguments: the first is a Boolean condition, and the second is a value that is reported if the condition fails, in this case 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()`. Note that although the processes are *spawned* one at a time, they will execute concurrently. It is, for example, quite possible that `incrementer(1)` finishes before `incrementer(0)` even gets going. And because Harmony tries every possible execution, it will consider such executions as well.

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

What is going on is that the Harmony program is compiled to machine instructions, and it is the machine instructions that are executed by the underlying Harmony 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;

```
def incrementer(self):
    count = count + 1;
    done[self] = True;
;
def main():
    await all(done);
    assert count == 2, count;
;
count = 0;
done = [ False, False ];
spawn incrementer(0);
spawn incrementer(1);
spawn main();
```

Figure 3.1: [code/Up.hny] Incrementing twice in parallel.

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. Harmony 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;
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 Harmony, 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 while it is just as easy to write concurrent programs in Python, it is much easier to find concurrency bugs using Harmony.

3.1 Harmony programs can usually be easily translated into Python. For example, Figure 3.2 is a Python version of Figure 3.1.

1. Run Figure 3.2 using Python. Does the assertion fail?
2. Using a script, run Figure 3.2 1000 times. For example, if you are using the bash shell (in Linux, say), you can do the following:

   ```
   for i in {1..1000}
   do
       python Up.py
   done
   ```

   How many times does the assertion fail (if any)?

3.2 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`. How large does `N` need to be for assertions to fail?

3.3 Can you think of a fix to Figure 3.1? Try one or two different fixes and run them through Harmony. 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.

```python
import threading

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

def main():
    while not all(done):
        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: [python/Up.py] Python implementation of Figure 3.1.

```python
import threading

N = 10000000

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

def main():
    while not all(done):
        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: [python/UpMany.py] Incrementing `N` times.

Chapter 4

The Harmony Virtual Machine

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

Harmony Values

Harmony programs, and indeed the HVM, manipulate Harmony values. Harmony values are recursively defined: they include booleans (`False` and `True`), integers (but not floating point numbers), strings (enclosed by double quotes), sets of Harmony values, and dictionaries that map Harmony values to other Harmony values. Another type of Harmony 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.

Harmony makes extensive use of dictionaries. A dictionary maps values, known as *keys*, to values. The Harmony dictionary syntax and properties are a little different from Python. Unlike Python, any Harmony value can be a key, including another dictionary. Harmony 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 Harmony value that k maps to in d:

```
d k
```

This is unfamiliar to Python programers, but in Harmony 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 $d\ a\ b$ ... is $(((d\ a)\ b)\ ...)$.

Tuples are special forms of dictionaries where the keys are the indexes into the tuple. For example, the tuple `(5, False)` is the same Harmony 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 Harmony, so `[a, b, c]` (which looks like a Python list) is the same Harmony value as `(a, b, c)` (which looks like a Python tuple), which in turn is the same Harmony 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,]`. The expression `[False]` just means `False`.

Harmony is not an object-oriented language, so objects don't have built-in methods. However, Harmony 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 Harmony currently supports.

Harmony Bytecode

A Harmony program is translated into HVM bytecode. The HVM 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 HVM manipulates Harmony values. A HVM has the following components:

- Code: This is an immutable and finite list of HVM instructions, generated from a Harmony program. The types of instructions will be described later.
- Shared memory: A HVM has just one memory location containing an Harmony 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 Harmony values, and a single mutable general purpose *register* that contains a Harmony 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. A name tag is represented as a dictionary with keys `.name` and `.tag`, but we shall often abbreviate it using the notation `name/tag`. For example, 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 HVM 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 Harmony memory is the empty dictionary, `dict{}`. The context bag has an initial context in it with name tag `__init__/()`, pc 0, register `()`, and an empty stack. Each machine instruction updates the state in some way.

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 Harmony values are unbounded trees. Both the memory and the register of a process always contain a dictionary that maps atoms to Harmony values. We call this a *directory*. A directory represents the state of a collection of variables named

```
code/Up.hny:1 def incrementer(self):
   0 Jump 25
   1 Frame incrementer self
code/Up.hny:2     count = count + 1;
   2 Load count
   3 Push 1
   4 2-ary +
   5 Store count
code/Up.hny:3     done[self] = True;
   6 Push ?done
   7 LoadVar self
   8 Address
   9 Push True
   10 Store
   11 Return
code/Up.hny:5 def main():
   12 Jump 25
   13 Frame main ()
code/Up.hny:6     await all(done);
   14 Load done
   15 1-ary all
   16 JumpCond False 14
code/Up.hny:7     assert count == 2, count;
   17 AtomicInc
   18 Load count
   19 Push 2
   20 2-ary ==
   21 Load count
   22 Assert2
   23 AtomicDec
   24 Return
code/Up.hny:9 count = 0;
   25 Push 0
   26 Store count
code/Up.hny:10 done = [ False, False ];
   27 Push [False, False]
   28 Store done
code/Up.hny:11 spawn incrementer(0);
   29 Push 0
   30 Dup
   31 Push PC(1)
   32 Spawn
```

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

```
#states = 71 diameter = 5
==== Safety violation ====
__init__/()    [0,25-41] 41 { count: 0, done: [False, False] }
incrementer/0 [1-4]       5 { count: 0, done: [False, False] }
incrementer/1 [1-11]     11 { count: 1, done: [False, True ] }
incrementer/0 [5-11]     11 { count: 1, done: [True,  True ] }
main/()       [13-22]    22 { count: 1, done: [True,  True ] }
>>> Harmony Assertion failed : 1
Open file://.../harmony.html for more information
```

Figure 4.2: The text output of running Harmony on Figure 3.1.

by the atoms. Because directories are Harmony 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 Harmony value, and it is relative to the "root" of the directory tree. For example, in Figure 3.1 the memory is a dictionary with two entries: `.count` and `.done`. And the value of entry `.done` is a dictionary with keys 0 and 1.

Compiling the code in Figure 3.1 results in the HVM bytecode listed in Figure 4.1. You can obtain this code by invoking `harmony` with the `-a` flag like so:

```
harmony -a Up.hny
```

Each process in the HVM 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 25 (skipping over the code for the methods). 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 HVM machine instructions, in case you want to read about the details. The `Frame` instruction lists the name of the method and the names of its arguments. The code generated from $count := count + 1$ in line 2 of `Up.hny` is as follows (see Figure 4.1):

2. The `Load` instruction pushes the value of the *count* variable onto the stack.
3. the `Push` instruction pushes the constant 1 onto the stack of the process.
4. `2-ary` is a `+` operation with 2 arguments. It pops two values from the stack (the value of *count* and 1), adds them, and pushes the result back onto the stack.
5. The `Store` instruction pops a Harmony value (the sum of the *count* variable and 1) and stores it in the *count* variable.

file:///.../harmony.html

Issue: process failure

Process	Steps	Shared Variables: *count*	Shared Variables: *done*
__init__/()	0 25-41	0	[False, False]
incrementer/0	1-4	0	[False, False]
incrementer/1	1-11	1	[False, True]
incrementer/0	5-11	1	[True, True]
main/()	13-22	1	[True, True]

Up.hny:1 def incrementer(self):	
0	Jump 25
1	Frame incrementer self
Up.hny:2 count = count + 1;	
2	Load count
3	Push 1
4	2-ary +
5	Store count

Process	Status	Stack Trace	Variables
main/()	failed	22 main(): assert count == 2, count; Harmony Assertion failed : 1	result = ()

Figure 4.3: The HTML output of running Harmony on Figure 3.1. There are three sections. The top shows the steps to the problematic state. The bottom left shows the source code and bytecode of the program. The bottom right shows the state of each process after the selected step. Each row in the stack trace of a process shows the current program counter, the method that is being evaluated, and the line of code that is being evaluated.

Once initialization completes, any processes that were spawned can run. You can think of Harmony as trying every possible interleaving of processes executing instructions. Figure 4.2 shows the output produced by running Harmony on the `Up.hny` program.

Harmony can report a variety of failure types:

- `Safety violation`: this means something went wrong with at least one of the executions of the program that it tried. This can include a failing assertion, divide by zero, using an uninitialized or non-existent variable, dividing a set by an integer, and so on. It also includes certain infinite loops. Harmony will print at trace of the shortest bad execution that it found.

- `Non-terminating States`: Harmony found one or more states from which there does not exist an execution such that all processes terminate. Harmony will not only print the non-terminating state with the shortest trace, but also the list of processes at that state, along with their name tags and program counters.

- `Stopped States`: Similar to non-terminating states, these are states in which some processes are left that cannot terminate. Chapter 9 will explain what it means for a process to be stopped.

In this case, Harmony reports a safety violation, in particular it reports that the assertion and that *count* has value 1 instead of 2. Next it reports an execution that failed this assertion. The program got to the faulty execution in 5 "steps." The output has four columns:

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

The first step is initialization. It sets shared variable *count* to 0 and shared variable *done* to `[False, False]`. If we look at the next three steps, we see that:

- Process `implementer/0` executed instructions 1 through 4, loading the value of *count* but stopping just before storing 1 into *count*;
- Process `implementer/1` executed instructions 1 through 11, storing 1 into into *count* and storing `True` into *done*[1];
- Process `implementer/0` continues execution, executing instructions 5 through 11 storing value 1 into *count* and storing `True` into *done*[0].

Finally, process `main()` runs and finds the problem. This makes precise the concurrency issue that we encountered.

Harmony also generates an HTML file that allows exploring more details of the execution interactively. Open the suggested HTML file and you should see something like Figure 4.3. If you hover the mouse over a machine instruction, it provides a brief explanation.

Exercises

4.1 Figure 4.4 shows an attempt at trying to fix the code of Figure 3.1. Run it through Harmony 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.

4.2 What if we moved line 2 of Figure 4.4 to after the **if** statement (between lines 7 and 8)? Do you think that would work? Run it through Harmony and describe either why it works or why it does not work.

```
def incrementer(self):
    entered[self] = True;
    if entered[1 − self]:        # if the other process has already started
        await done[1 − self];     # wait until it is done
    ;
    count = count + 1;
    done[self] = True;
;
def main():
    await all(done);
    assert count == 2, count;
;
count = 0;
entered = [ False, False ];
done = [ False, False ];
spawn incrementer(0);
spawn incrementer(1);
spawn main();
```

Figure 4.4: [code/UpEnter.hny] Broken attempt at fixing the code of Figure 3.1.

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 accessing 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. Typically, you want some invariant property of the data structure to hold at the beginning and at the end of the critical section, but in the middle the invariant may be temporarily broken—this is no issue as critical sections guarantee that no other process will be able to see it. An implementation of a data

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

Figure 5.1: [code/csbarebones.hny] A bare bones 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: [code/cs.hny] Harmony model of a critical section.

structure that can be safely accessed by multiple processes (or threads) and free of race conditions 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 Harmony code. Here `@cs` is a *label*, identifying a location in the HVM bytecode. The first thing we need to ensure is that there can never be two processes in 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.

Harmony in fact supports this. It has an operator **atLabel** L, where L is the atom containing the name of the label (in this case, `.cs`). The operator returns a bag (multiset) of name tags of processes executing at that label. The bag is represented by a dictionary that maps each element in the bag to the number of times the element appears in the bag. Method **atLabel** only exists for specification purposes—do not use it in normal code. The assertion also makes use of the **nametag**() operator that returns the name tag of the current process. If you run the code through Harmony, the assertion should fail because there is no code yet for safely entering and exiting the critical section.

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

```
await False;
```

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 is 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 (1) there exists a non-empty set S of processes that are trying to enter the critical section and (2) processes in the critical section always leave eventually, then eventually one process in S will enter the critical section. We call this *progress*.

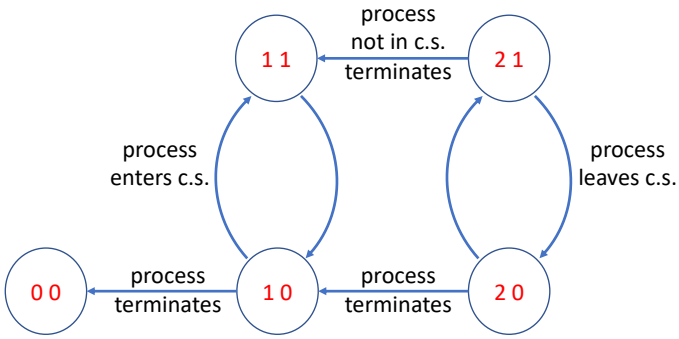

Figure 5.3: High-level state diagram specification of mutual exclusion with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes in the critical section.

In order to detect violations of progress, and other liveness problems in algorithms in general, Harmony 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. We will instead 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. (Recall that Harmony will try every possible execution, and so it will evaluate both choices.)

Figure 5.3 shows a high-level state diagram of mutual exclusion and progress. 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 until after leaving the critical section. We will now consider various approaches toward implementing this specification.

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 *lockTaken* being `False`. To enter the critical section, a process waits until *lockTaken* 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 *lockTaken* back to `False`.

Unfortunately, if we run the program through Harmony, we find that the assertion still fails. Figure 5.5 shows the Harmony output. Process 0 finds that the lock is available, but just before it stores `True` in *lockTaken* in instruction 9, process 1 gets to run. (Recall that you can hover your mouse over a machine instruction in order to see what it does.) Because *lockTaken* is still `False`, it too believes it can acquire the lock, and stores `True` in `lockTaken` and moves on to

```
1   def process(self):
2       while choose({ False, True }):
3           # Enter critical section
4           await not lockTaken;
5           lockTaken = True;
6
7           # Critical section
8           @cs: assert atLabel.cs == dict{ nametag(): 1 };
9
10          # Leave critical section
11          lockTaken = False;
12      ;
13  ;
14  lockTaken = False;
15  spawn process(0);
16  spawn process(1);
```

Figure 5.4: [code/naiveLock.hny] Naïve implementation of a shared lock.

Issue: process failure

Process	Steps	Shared Variables: *count*	Shared Variables: *done*
__init__/()	0 34-50	0	[False, False]
incrementer/0	1-4	0	[False, False]
incrementer/1	1-11	1	[False, True]
incrementer/0	5-11	1	[True, True]
main/()	13-21 23-24 26-31	1	[True, True]

Up.hny:1 def incrementer(self):	
0	Jump 34
1	Frame incrementer self
Up.hny:2 count = count + 1;	
2	Load count
3	Push 1
4	2-ary +
5	Store count
Up.hny:3 done[self] = True;	

Process	Status	Stack Trace	Variables
main/()	failed	31 main(): assert count == 2, count; Harmony Assertion failed : 1	result = ()

Figure 5.5: The HTML output of running Harmony on Figure 5.4.

```
1   def process(self):
2       while choose({ False, True }):
3           # Enter critical section
4           flags[self] = True;
5           await not flags[1 − self];
6
7           # Critical section
8           @cs: assert atLabel.cs == dict{ nametag(): 1 };
9
10          # Leave critical section
11          flags[self] = False;
12      ;
13  ;
14  flags = [ False, False ];
15  spawn process(0);
16  spawn process(1);
```

Figure 5.6: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Issue: non-terminating state

Process	Steps	Shared Variable *flags*
__init__/()	0 37-47	[False, False]
process/0	1-2 3(choose True) 4-12	[True, False]
process/1	1-2 3(choose True) 4-12	[True, True]

13	Load flags
14	Apply
15	JumpCond False 17
naiveFlags.hny:6 pass;	
16	Jump 10
naiveFlags.hny:10 @cs: assert atLabel.cs == dict{ nametag(): 1 };	
17	AtomicInc
18	AtomicInc

Process	Status	Stack Trace	Variables
process/0	blocked	13 process(0): while flags[1 - self]:	result = ()
			self = 0
process/1	blocked	13 process(1): while flags[1 - self]:	result = ()
			self = 1

Figure 5.7: The HTML output of running Harmony on Figure 5.6.

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

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

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

Figure 5.8: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

the critical section. Finally, process 0 moves on, also stores `True` and also moves into the critical section. Process 0 is the one that detects the problem. The *lockTaken* 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 5.6 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 following invariant: 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 in the critical section at time t_1. Let t_2 be the time at which process 0 sets $flags[0]$ to `False`. Process 0 is in the critical section sometime between t_1 and t_2. It is easy to see that process 1 cannot enter the critical section between t_1 and t_2, because $flags[1]$ == `False` at time t_1. To reach the critical section between t_1 and t_2, it would first have to set $flags[1]$ to `True` and then wait until $flags[0]$ == `False`. But that does not happen until time t_2.

However, if you run the program through Harmony (Figure 5.7), it turns out the solution does 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 that we propose 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 Harmony. 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

5.1 Run Figure 5.2 using Harmony. As there is no protection of the critical section, mutual exclusion is violated, the assertion should fail, and a trace should be reported. Now insert

```
await False;
```

just before entering the critical section in Figure 5.2 and run Harmony again. Mutual exclusion is guaranteed but progress is violated. Harmony should print a trace to a state from which a terminating state cannot be reached. Describe in English the difference in the failure reports before and after inserting the code.

5.2 See if you can come up with some different approaches that satisfy both mutual exclusion and progress. Try them with Harmony 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, Gary L. 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 Figure 5.6 and Figure 5.8, 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 Harmony are atomic. The process continues to be polite, waiting until either the other process is nowhere near the critical section ($flag[1 - self] == \texttt{False}$) or has given way ($turn == self$). Running the algorithm with Harmony 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 Harmony virtual machine (HVM) works. In Chapter 4 we talked about the concept of *state*: at any point in time the HVM 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 of each process, including its program counter and the contents of its stack. Each time a process executes a HVM machine instruction, the state changes (if only because the program counter of the process changes). We call that a *step*. Steps in Harmony are atomic.

The HVM 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 Harmony. 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. *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,

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        turn = 1 − self;
        await (not flags[1 − self]) or (turn == self);

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

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

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

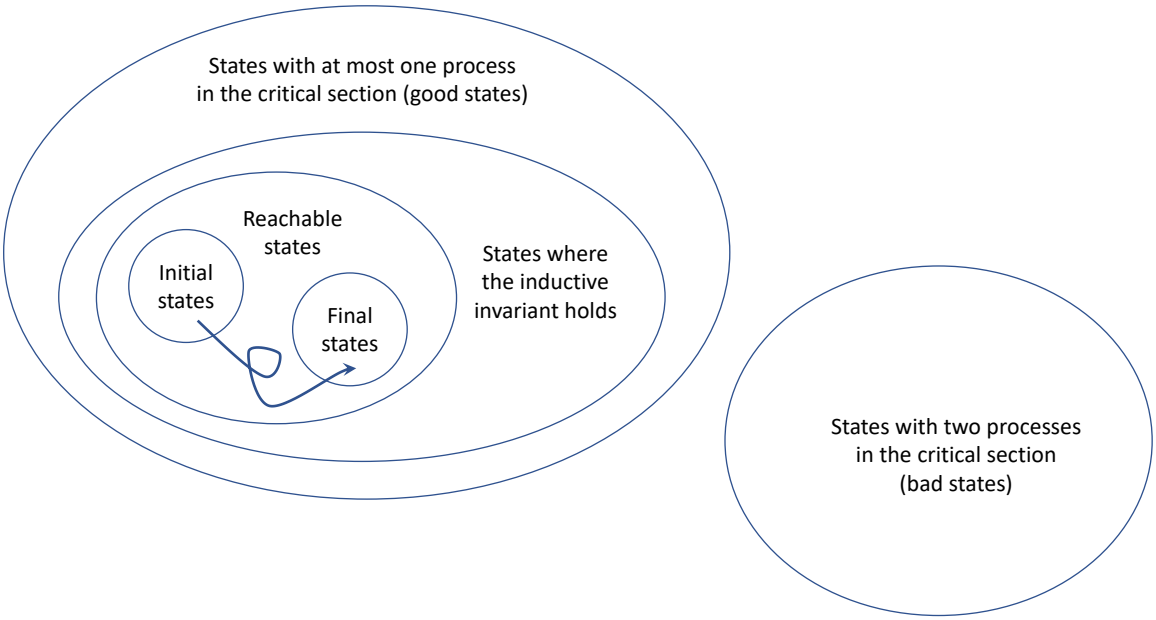

Figure 6.2: Venn diagram classifying all states and a trace.

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 to work you would need a predicate that describes exactly which states are reachable. But we do not have such a predicate: we know how to describe the set of reachable states, but given an arbitrary state it is not easy to see whether it is reachable or not.

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

- $\mathcal{I}$ holds in the initial state.
- For any state in which $\mathcal{I}$ holds and for any process in the state, if the process takes a step, then $\mathcal{I}$ also holds in the resulting state.

Unlike an ordinary 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 Harmony 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 be at the start of the **await** statement. Also imagine that in this particular state $turn = 1$. Now let process 1 make a sequence of steps. Because $turn = 1$, process 1 will stop waiting 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 $\mathcal{C}(i) = \neg flags[1-i] \vee turn = i$, that is, the condition on the **await** statement for process i. Let predicate $\mathcal{I}_p(i)$ be the following: if process i is at label `@cs` (i.e., process i is in the critical section), then $\mathcal{C}(i)$ holds or process $1-i$ is executing after setting $flags[i]$ but still before setting $turn$ to $1-i$. More formally, $\mathcal{I}_p(i) = \texttt{process}(i)@cs \Rightarrow \mathcal{C}(i) \vee \texttt{process}(1-i)@gate$. For Peterson's Algorithm, an inductive invariant that works well is $\mathcal{I}_p(0) \wedge \mathcal{I}_p(1)$.

Figure 6.3 formalizes $\mathcal{I}_p(i)$ in Harmony. The label `@gate` refers to the instruction that sets $turn$ to $1-i$. You can run Figure 6.3 to determine that $\mathcal{I}_p(i)$ 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: $(\mathcal{C}(0) \vee \texttt{process}(1)@gate) \wedge (\mathcal{C}(1) \vee \texttt{process}(0)@gate)$. 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 $\mathcal{C}(0) \wedge \mathcal{C}(1)$. So we conclude that $turn = 0 \wedge turn = 1$, a logical impossibility. Thus $\mathcal{I}_p(i)$ implies exclusivity.

To see that $\mathcal{I}_p(0) \wedge \mathcal{I}_p(1)$ is, in fact, an inductive invariant, first note that it 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

```
def process(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True;
        @gate: turn = 1 − self;
        await (not flags[1 − self]) or (turn == self);

        # 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: [code/PetersonInductive.hny] Peterson's Algorithm with Inductive Invariant

have to show that if we are in a state in which $\mathcal{I}_p(0)$ holds, then any step will result in a state in which $\mathcal{I}_p(0)$ still holds. If $\mathcal{I}_p(0)$ 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 $\mathcal{I}_p(0)$ would hold trivially over the next state. Therefore we only need to consider a step by process 1.

From $\mathcal{I}_p(0)$ 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. We have to show that if process 1 takes a step, then one of those cases must hold after the step. 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 Harmony 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 invariants, particularly inductive ones, in your Harmony code.

A cool anecdote is the following. When the author of Harmony 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 $\mathcal{C}(i)$ (i.e., $\mathcal{I}_p(i)$ 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 a proof by induction 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 Harmony. 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. (The author fixed the Wikipedia page with the help of Fred B. Schneider.)

Exercises

6.1 Figure 6.4 presents another correct solution to the mutual exclusion problem. It is similar to the one in Figure 5.6, 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?

6.2 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:

```
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
```

6.3 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.

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

6.5 Bonus question: Implement Dekker's Algorithm, Eisenstein and McGuire's Algorithm, Szymański's Algorithm, or the Lamport's Bakery Algorithm. Note that the last one uses unbounded state, so you should modify the processes so they only try to enter the critical section a bounded number of times.

```
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: [code/csonebit.hny] Mutual exclusion using a flag per process.

Chapter 7

Harmony 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 Harmony values to Harmony values, and their syntax is indistinguishable. If `f` is either a method or a dictionary, and x is an arbitrary Harmony value, then `f` x, `f`(x), and `f`[x] are all the same expression in Harmony.

Harmony does not have a **return** statement. (You can assign a return value to a method by setting the *result* variable.) Neither does it support **break** or **continue** statements in loops. One reason for their absence is that, particularly in concurrent programming, such control flow directions are highly error-prone. It's easy to forget to, say, release a lock when returning a value in the middle of a method, a major source of bugs in practice.

Harmony 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 Harmony, it is also sometimes convenient to have a method update a shared variable specified as an argument. For this, as mentioned in Chapter 4, each shared variable has an *address*. If x is a shared variable, then the expression ?x is the address of x. If a variable contains an address, we call that variable a *pointer*. 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 (`*&`$x == x$).

Often, pointers point to dictionaries, and so if p is such a pointer, then !(p)`.field` would evaluate to the specified field in the dictionary. Note that the parentheses in this expression are needed, as `!`p`.field` would wrongly evaluate `!`(p`.field`). !(p)`.field` is such a common expression that, like C, Harmony supports the shorthand p−>`field`, which greatly improves readability.

Figure 7.1 again shows Peterson's algorithm, but this time with methods defined to enter and exit the critical section. The name *mutex* is often used to denote a variable or value that is used for mutual exclusion. `P_mutex` is a method that returns a "mutex," which, in this case, is a dictionary that contains Peterson's Algorithm's state: a turn variable and two flags. Both methods `P_enter`

```
def P_enter(pm, pid):
    pm→flags[pid] = True;
    pm→turn = 1 − pid;
    await (not pm→flags[1 − pid]) or (pm→turn == pid);
;
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 Harmony 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: [code/PetersonMethod.hny] Peterson's Algorithm accessed through methods.

and `P_exit` take two arguments: a pointer to a mutex and the process identifier (0 or 1). $pm \rightarrow$ `turn` is the value of the `.turn` key in the dictionary that pm points to.

You can put the first three methods in its own Harmony source file and include it using the Harmony `import` statement. This would make the code re-usable by other applications.

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 HVM does not have any specific built-in interlock instructions, it does have support for executing multiple instructions atomically. This feature is available in the Harmony language in two ways. First, any Harmony statement can be made atomic by placing a label in front of it. Second, a group of Harmony statements can be made atomic through its **atomic** statement. We can use **atomic** statement blocks to implement a wide variety of interlock operations. For example, we could fix the program in Figure 3.1 by constructing an atomic increment operation for a counter, like so:

```
def atomic_inc(ptr):
    atomic:
        !ptr += 1;
    ;
;
count = 0;
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 to implement mutual exclusion for a set of `N` processes. The approach is called *spinlock*, because each process is "spinning" (executing a tight loop) until it can acquire the lock. The program uses `N` + 1 variables. Variable *shared* is initialized to `False` while *private*[*i*] for

```
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, ] * N;
for i in {0..N–1}:
    spawn process(i);
;
```

Figure 8.1: [code/spinlock.hny] Mutual Exclusion using a "spinlock" based on test-and-set.

```
import spinlock;

def checkInvariant():
    let sum = 0:
        if not shared:
            sum = 1;
        ;
        for i in {0..N–1} such that not private[i]:
            sum += 1;
        ;
        result = sum <= 1;
    ;
;
def invariantChecker():
    assert checkInvariant();
;
spawn invariantChecker();
```

Figure 8.2: [code/spinlockInv.hny] Checking invariants.

each process i is initialized to `True`. An important invariant, $\mathcal{I}_1$, of the program is that at any time at most one of these variables is `False`. Another invariant, $\mathcal{I}_2(i)$, is that if process i is in the critical section, then $private[i]$ == `False`. Between the two (i.e., $\mathcal{I}_1 \wedge \forall i : \mathcal{I}_2(i)$), it is clear that only one process can be in the critical section at the same time.

$\mathcal{I}_1$ is an inductive invariant. To see that invariant $\mathcal{I}_1$ is maintained, note that `!`p == `True` upon entry of `tas` (because of the condition on the **while** loop that the `tas` method is invoked in). So there are two cases:

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

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

Invariant $\mathcal{I}_1$ is also easy to verify for exiting the critical section. Invariant $\mathcal{I}_2(i)$ is obvious as (i) process i only proceeds to the critical section if $private[i]$ == `False`, and (ii) no other process modifies $private[i]$.

Harmony can check these invariants as well. Figure 8.1 already has the code to check $\mathcal{I}_2(i)$. But how would one go about checking an invariant like $\mathcal{I}_1$? Invariants must hold for every state. For $\mathcal{I}_2$ 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 $\mathcal{I}_1$ in *every state* (after the variables have been initialized).

We can do this by adding another process that 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 Harmony: 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 method is invoked by a process that runs alongside the other processes. In Harmony, **assert** statements

are executed atomically, so the evaluation of the assertion is not interleaved with the execution of other processes. Because Harmony tries every possible execution, the process is guaranteed to find violations of the invariant if it does not hold.

Exercises

8.1 Implement an atomic swap operation. It should take two pointer arguments and swap the values.

8.2 Implement a spinlock using the atomic swap operation.

8.3 For the solution to Exercise 8.2, write out the invariants that need to hold and check them using Harmony.

8.4 People who use an ATM often first check their balance and then withdraw a certain amount of money not exceeding their balance. A negative balance is not allowed. Figure 8.3 shows two operations on bank accounts: one to check the balance and one to withdraw money. Note that all operations on accounts are carefully protected by a lock (i.e., there are no data races). The `customer` method models going to a particular ATM and withdrawing money not exceeding the balance. Method `checker` looks for negative balances. Running the code through Harmony reveals that there is a bug. It is a common type of concurrency bug known as *Time Of Check Time Of Execution* (TOCTOE). In this case, by the time the withdraw operation is performed, the balance can have changed.

1. Fix the code in Figure 8.3. Note, you should leave the customer code the same. You are only allowed to change the `atm_` methods, and you cannot use the **atomic** statement.
2. Is it necessary to obtain a lock in `atm_check_balance()`?

```
import synch;

const N_ACCTS = 2;
const N_CUSTOMERS = 2;
const N_ATMS = 2;
const MAX_BALANCE = 1;

def atm_check_balance(acct):    # return the balance on acct
    lock(?accounts[acct].lock);
    result = accounts[acct].balance;
    unlock(?accounts[acct].lock);
;
def atm_withdraw(acct, amount): # withdraw amount from acct
    lock(?accounts[acct].lock);
    accounts[acct].balance -= amount;
    result = True;              # return success
    unlock(?accounts[acct].lock);
;

def customer(atm, acct, amount):
    let balance = atm_check_balance(acct):
        if amount <= balance:
            atm_withdraw(acct, amount);
        ;
    ;
;
def checker():
    assert min(accounts[acct].balance for acct in {0..N_ACCTS–1}) >= 0;
;
accounts = [ dict{ .lock: Lock(), .balance: choose({0..MAX_BALANCE}) }
                for acct in {0..N_ACCTS–1} ]
;
spawn checker();
for i in {1..N_ATMS}:
    spawn customer(i, choose({0..N_ACCTS–1}), choose({0..MAX_BALANCE}));
;
```

Figure 8.3: [code/atm.hny] Withdrawing money from an ATM.

Chapter 9

Locks and Blocking

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 (e.g., none of Python, C, or Java support it), while the latter is very common.

In Harmony, `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* statements 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. You can also use **atomic** statements in your test code. In fact, `assert` statements are executed atomically.

Locks are probably the most prevalent and basic form of synchronization in concurrent programs. Typically, whenever you have a shared data structure, you want to protect the data structure with a lock and acquire the lock before access and release it immediately afterward. In other words, you want the access to the data structure to be a critical section. When there is a bug in a program because some code omitted obtaining a lock before accessing the data structure, that is known as a *data race*.

Because locks are so common, Harmony 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.

```
def tas(s):
    atomic:
        result = !s;
        !s = True;
    ;
;
def process():
    while choose({ False, True }):
        await not tas(?lock);
        @cs: assert atLabel.cs == dict{ nametag(): 1 };
        lock = False;
    ;
;
lock = False;
for i in {1..10}:
    spawn process();
;
```

Figure 9.1: [code/csTAS.hny] 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):
    await not tas(lk);
;
def unlock(lk):
    !lk = False;
;
```

Figure 9.2: [modules/synch.hny] The `Lock` interface and implementation in the `synch` module.

```
import synch;

def process(self):
    lock(?countlock);
    count = count + 1;
    unlock(?countlock);
    done[self] = True;
;
def main(self):
    await all(done);
    assert count == 2, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

Figure 9.3: [code/UpLock.hny] Program of Figure 3.1 fixed with a lock.

We call a process *blocked* if a process cannot change the state or terminate or can only do so if another process changes the state first. A process trying to acquire a test-and-set spinlock held by another process is a good example of a process being blocked. The only way forward is if the other process releases the lock. A process that is in an infinite loop is also considered blocked, although Harmony can often detect infinite loops specifically.

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 Harmony, a process can suspend itself and save its context (state) 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 Harmony value. The syntax of the expression is as follows:

stop L

Here L is a shared variable containing a list. Another process can revive the process using the **go** statement:

go C R

Here C is a context and R is a Harmony 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.

```
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: [modules/syncS.hny] The `Lock` interface in the `synchS` module uses suspension.

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. The interface is implemented as follows:

- 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 and 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 (see Appendix D) to resume the first process that got suspended 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 and how to prevent it.

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

```
harmony -m synch=synchS x.hny
```

runs the file `x.hny` using the suspension version of the `synch` module. You will find that using the `synchS` module often leads to searching a significantly larger state 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, while the `synch` module does not.

Exercises

9.1 Run Figure 9.3 using (i) `synch` and (ii) `synchS`. Report how many states were explored by Harmony for each module.

9.2 Figure 9.5 shows a Harmony program with two variables x (initially 0) and y (initially 100) that can be accessed through methods `setX` and `getXY`. An application invariant is that `getXY` should return a pair that sums to 100. Add the necessary synchronization code.

9.3 Implement `trylock`(lk) as an additional interface for both the `synch` and `synchS` modules. This interface is like `lock`(lk) but never blocks. It returns `True` if the lock was available (and now acquired) or `False` if the lock was already taken. Hint: you do not have to change the existing code.

```
def setX(a):
    x = a;
    y = 100 - a;
;
def getXY():
    result = [x, y];
;
def checker():
    let xy = getXY():
        assert (xy[0] + xy[1]) == 100, xy;
    ;
;
x, y = 0, 100;
spawn checker();
spawn setX(50);
```

Figure 9.5: [code/xy.hny] Incomplete code with desired invariant $x + y = 100$.

Chapter 10

Concurrent Data Structures

The most common use for locks is in building concurrent data structures. By way of example, we will demonstrate how to build a concurrent queue. The queue will have the following API:

- $q =$ `Qnew`(): allocate a new queue;
- `Qenqueue`(q, v): add v to the tail of the queue;
- $r =$ `Qdequeue`(q): returns $r = ()$ if q is empty or $r = (v,)$ (a singleton tuple) if v was at the head of the queue.

See Figure 10.1 for a simple test program that uses the queue.

We will first implement the queue as a linked list. The implementation in Figure 10.2 uses the `alloc` module for dynamic allocation of nodes in the list using `malloc`() and `free`(). `malloc`(v) returns a new memory location initialized to v, which should be released with `free`() when it is no longer in use. The queue maintains a `head` pointer to the first element in the list and a `tail` pointer to the last element in the list. The `head` pointer is **None** if and only if the queue is empty.

`Qnew()` returns a new queue object consisting of a **None** head and tail pointer and a lock. `Qenqueue`(q, v) and `Qdequeue`(q) both take a pointer q to the queue object because both may modify the queue. Before they access the value of the head or tail of the queue they first obtain the lock. When they are done, they release the lock.

The implementation in Figure 10.3 [MS98] is similar but uses separate locks for the head and tail, allowing an enqueue and a dequeue operation to proceed concurrently. To avoid contention between the head and the tail that requires both locks, the queue uses a dummy node at the head of the linked list. Except for the initial state of the queue, the dummy node is the last node that was dequeued. Note that neither the `head` nor `tail` pointer are ever **None**.

Exercises

10.1 Add a method `Qcontains`(q, v) to both Figure 10.2 and Figure 10.3 that checks to see if v is in queue q.

10.2 Add a method `Qremove`(q, v) to both Figure 10.2 and Figure 10.3 that removes v, if any, from queue q.

```
import queue;

def sender(q, v):
    Qenqueue(q, v);
;
def receiver(q):
    let done = False:
        while not done:
            let v = Qdequeue(q):
                done = v == ();
                assert done or (v[0] in { 1, 2 });
            ;
        ;
    ;
;

queue = Qnew();
spawn sender(?queue, 1);
spawn sender(?queue, 2);
spawn receiver(?queue);
spawn receiver(?queue);
```

Figure 10.1: [code/queuetest.hny] Test program for a concurrent queue.

```
import alloc;

def Qnew():
    result = dict{ .head: None, .tail: None, .lock: Lock() };
;
def Qenqueue(q, v):
    let node = malloc(dict{ .value: v, .next: None }):
        lock(?q→lock);
        if q→head == None:
            q→head = q→tail = node;
        else:
            q→tail→next = node;
            q→tail = node;
        ;
        unlock(?q→lock);
    ;
;
def Qdequeue(q):
    lock(?q→lock);
    let node = q→head:
        if node == None:
            result = ();
        else:
            result = (node→value,);
            q→head = node→next;
            free(node);
        ;
    ;
    unlock(?q→lock);
;
```

Figure 10.2: [code/queue.hny]A basic concurrent queue data structure.

```
import alloc;

def Qnew():
    let dummy = malloc(dict{ .value: (), .next: None }):
        result = dict{ .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() };
    ;
;
def Qenqueue(q, v):
    let node = malloc(dict{ .value: v, .next: None }):
        lock(?q→tllock);
        q→tail→next = node;
        q→tail = node;
        unlock(?q→tllock);
    ;
;
def Qdequeue(q):
    lock(?q→hdlock);
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = ();
        else:
            free(dummy);
            result = (node→value,);
            q→head = node;
        ;
    ;
    unlock(?q→hdlock);
;
```

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.

10.3 The test program in Figure 10.1 is not very thorough. Design and implement a better one.

10.4 Create a thread-safe sorted binary tree. Implement methods `BTnew`() to create a new binary tree, `BTinsert`(t, v) that inserts v into tree t, and `BTcontains`(t, v) that checks if v is in tree t. Use a single lock per binary tree.

10.5 Bonus question: create a binary tree that uses, instead of a single lock per tree, a lock for each node in the tree.

Chapter 11

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 (i) one writer or (ii) 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 11.1 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. Figure 11.2 shows how the code may be used. 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 11.3 shows a high-level specification for two processes.

A process that wants to read first waits until there are no writers: $nwriters == 0$. Then it increments the number of readers. Similarly, a process that wants to write waits until there are no readers *or* writers. Then 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` $\Rightarrow nwriters \geq n$,
- $(nreaders \geq 0 \wedge nwriters = 0) \vee (nreaders = 0 \wedge 0 \leq nwriters \leq 1)$.

```
import synch;

def acquire_rlock():
    let blocked = True:
        while blocked:
            lock(?rwlock);
            if nwriters == 0:
                nreaders += 1;
                blocked = False;
            ;
            unlock(?rwlock);
        ;
    ;
;
def release_rlock():
    lock(?rwlock);
    nreaders -= 1;
    unlock(?rwlock);
;
def acquire_wlock():
    let blocked = True:
        while blocked:
            lock(?rwlock);
            if (nreaders + nwriters) == 0:
                nwriters = 1;
                blocked = False;
            ;
            unlock(?rwlock);
        ;
    ;
;
def release_wlock():
    lock(?rwlock);
    nwriters = 0;
    unlock(?rwlock);
;
rwlock = Lock();
nreaders = 0;
nwriters = 0;
```

Figure 11.1: [code/RW.hny] Busy-Waiting Reader/Writer Lock implementation.

```
import RW;

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..4}:
    spawn process();
;
```

Figure 11.2: [code/RWtest.hny] Test code for Figure 11.1.

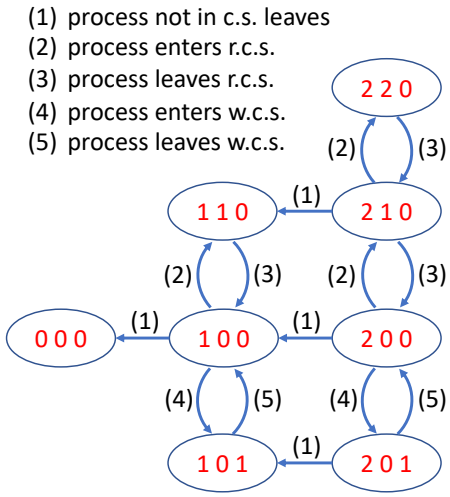

Figure 11.3: High-level state diagram specification of reader/writer locks with up to two processes. The first number in a state gives the number of processes; the second number is the number of processes reading in the critical section; the third is the number of processes writing in the critical section.

```
import RW;

const NREADERS = 3;
const NWRITERS = 3;

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();
;
for i in {1..NREADERS}:
    spawn reader(i);
;
for i in {1..NWRITERS}:
    spawn writer();
;
acquire_rlock();
```

Figure 11.4: [code/RWbusychk.hny] Checking for Busy Waiting.

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, wasting CPU cycles. Figure 3.1 also has an example of busy-waiting: the `main` process waits for the other two processes to finish by checking their `done` flags.

The astute reader might wonder if obtaining the mutex itself is an example of busy-waiting. After all, the Harmony `synch` implementation of `lock()` spins in a loop until the mutex is available (see Figure 9.2). However, there is a big difference. As we pointed out, the processes that are waiting for a spinlock are *blocked*. In most operating systems and programming language runtimes, when a process acquires a mutex, the process is placed on a scheduling queue and stops using CPU cycles until the mutex becomes available. Harmony 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 11.1 tries to obtain the mutex repeatedly.

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.

Harmony does not complain about Figure 11.1, but, using the right test, Harmony can be used to check for busy-waiting. Figure 11.4 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 Harmony can test for that by running it with the `-b` flag. This flag tells Harmony to check that all states are non-terminating and lead to a state in which all processes are blocked.

Exercises

11.1 Draw additional states and steps in Figure 11.3 for three processes.

11.2 Optimize your solutions to Exercise 10.1 to use reader/writer locks.

11.3 Using busy waiting, implement a "bound lock" that allows up to `M` processes to acquire it at the same time. A bound lock with `M == 1` is an ordinary lock. You should define a constant `M` and two methods: `acquire_bound_lock()` and `release_bound_lock()`. (Bound locks are useful for situations where too many processes working at the same time might exhaust some resource such as a cache.)

11.4 Write a test program for your bound lock that checks that no more than `M` processes can acquire the bound lock.

Chapter 12

Reader/Writer Locks with Blocking

Figure 12.1 presents a reader/writer lock implementation that does not busy-wait. You can test the code by running `harmony -m RW=RWlock RWtest.hny`. 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 then any process can 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* 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 enters 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 Harmony 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 writers 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 using an ordinary lock, never allowing more than one process in the critical section, and Harmony would not have complained.

```
import synch;

def acquire_rlock():
    lock(?rlock);
    if nreaders == 0:
        lock(?rwlock);
    ;
    nreaders += 1;
    unlock(?rlock);
;
def release_rlock():
    lock(?rlock);
    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;
```

Figure 12.1: [code/RWlock.hny] Reader/Writer with Two Locks.

```
import RWlock;

const NREADERS = 3;
const NWRITERS = 2;

def reader(self):
    acquire_rlock();
    @rcs: assert atLabel.wcs == dict{};
    flags[self] = True;
    if choose({ False, True }):
        let blocked = True:
            while blocked:
                if all(flags[i] for i in {1..NREADERS}):
                    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 12.2: [code/RWmulti.hny] Checking that multiple readers can acquire the read lock.

Figure 12.2 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 Harmony will readily detect. Try it out!

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

Exercises

12.1 Why do you suppose the code in Figure 12.2 uses a flag per process rather than a simple counter? Can you write a version of this test that uses a counter instead of a flag per process?

12.2 Write a test program for "bound locks" (Exercise 11.3) that checks that up to `M` processes can acquire the bound lock at the same time.

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 the following synchronization problem that we call *Italian Philosophers* (a simplification of the famous Dining Philosophers problem described in Chapter 18). There are five philosophers sitting around a table, each with a plate of spaghetti in front of them. 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. Unfortunately, a lock can only count to one.

Introduced by Dijkstra, a *semaphore* is a synchronization primitive that can solve this kind of problem [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. 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.

Figure 13.1 shows the `synch` module implementation of semaphores. It represents a semaphore as an integer. Figure 13.2 shows a solution to the Italian Philosphers problem using a semaphore.

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

Exercices

13.1 How might you check with Harmony that at most three Italian philosophers can eat at the same time?

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

Figure 13.1: [modules/synch.hny] 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: [code/DinersSema.hny] Italian Philosophers.

13.2 Implement a test program that checks to see that the solution allows more than one Italian philosopher to eat at the same time?

13.3 Figure 13.3 shows an iterative implementation of the Qsort algorithm, and Figure 13.4 an accompanying test program. The array to be sorted is stored in shared variable a. Another shared variable, *todo*, contains the ranges of the array that need to be sorted (initially the entire array). Re-using as much of this code as you can, implement a parallel version of this. You should not have to change the methods `swap`, `partition`, or `sortrange` for this. Create `NWORKERS` "worker processes" that should replace the `qsort` code. Each worker loops until *todo* is empty and sorts the ranges that it finds until then. The `main` process needs to wait until all workers are done. Use a semaphore for this: the `main` process should use `P` to wait for each worker to terminate. After that it should check that the array is sorted and that it is a permutation of the original array.

```
def swap(p, q):                # swap !p and !q
    !p, !q = !q, !p;
;
def partition(lo, hi):
    result = lo;
    for i in {lo..hi − 1}:
        if a[i] <= a[hi]:
            swap(?a[result], ?a[i]);
            result += 1;
        ;
    ;
    swap(?a[result], ?a[hi]);
;
def sortrange(range):
    let lo, hi = range let pivot = partition(lo, hi):
        if (pivot − 1) > lo:
            todo |= { (lo, pivot − 1) };
        ;
        if (pivot + 1) < hi:
            todo |= { (pivot + 1, hi) };
        ;
    ;
;
def qsort():
    while todo != {}:
        let range = choose(todo):
            todo −= { range };
            sortrange(range);
        ;
    ;
;
```

Figure 13.3: [code/qsort.hny] Iterative qsort() implementation.

```
import list;
import qsort;

const NITEMS = 4;

def sorted():                    # return whether array 'a' is sorted
    result = all(a[i − 1] <= a[i] for i in {1..len(a)−1});
;
def main(copy):
    qsort();                         # sort array 'a'
    assert sorted();                    # is the array sorted?
    assert list2bag(a) == list2bag(copy); # is it a permutation?
;

const values = {1..NITEMS};
a = [ choose(values) for i in {1..choose({1..NITEMS})} ];
todo = { (0, len(a) − 1) };
spawn main(a);
```

Figure 13.4: [code/qsorttest.hny] Test program for Figure 13.3.

Chapter 14

Bounded Buffer

A good example of a resource with multiple units is the so-called *bounded buffer* (aka *ring buffer*). Suppose you have a *producer* (some process) wanting to communicate a stream of items with a *consumer* (another process). A bounded buffer is essentially a queue implemented using a circular buffer of a certain length and two pointers: one where the producer inserts items, and one where the consumer extracts items. If the buffer is full, the producer must wait; if the buffer is empty, the consumer must wait. This problem is known as the "Producer/Consumer Problem" and was proposed by Dijkstra [Dij72].

The producer/consumer pattern is common. Processes may be arranged in *pipelines*, where each upstream process is a producer and each downstream process is a consumer. Or processes may be arranged in a manager/worker pattern, with a manager producing jobs and workers consuming and executing them in parallel. Or, in the client/server model, some process may act as a *server* that clients can send requests to and receive responses from. In that case, there is a bounded buffer for each client/server pair. Clients produce requests and consume responses, while the server consumes requests and produces responses.

In general, we allow multiple producers and multiple consumers all sharing the same bounded buffer. Figure 14.1 gives a high-level description. 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 description 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. For this, 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

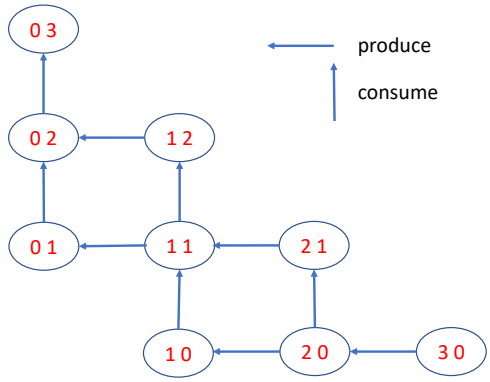

Figure 14.1: High-level state diagram specification of producers/consumers.

n_empty > 0 and 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 they 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.

Figure 14.3 invokes the code of Figure 14.2. We can run this through Harmony, but what would it test? There are no assert statements in the code. We can split the functionality of this code into two and test each separately. 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. To this this, one can write a script like the one in Figure 14.4.

Figure 14.5 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

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);
```

Figure 14.2: [code/BBsema.hny] Bounded Buffer implementation using semaphores.

```
import BBsema;

const NPRODS = 3;     # number of producers
const NCONSS = 3;     # number of consumers

for i in {1..NPRODS}:
   spawn produce(i);
;
for i in {1..NCONSS}:
   spawn consume();
;
```

Figure 14.3: [code/BBsematest.hny] Test program for Figure 14.2.

```
harmony -b -c NSLOTS=0 -c NPRODS=1 -c NCONSS=1 BBsematest.hny
harmony -b -c NSLOTS=1 -c NPRODS=0 -c NCONSS=1 BBsematest.hny
harmony    -c NSLOTS=1 -c NPRODS=1 -c NCONSS=0 BBsematest.hny
harmony    -c NSLOTS=1 -c NPRODS=1 -c NCONSS=1 BBsematest.hny
harmony -b -c NSLOTS=1 -c NPRODS=1 -c NCONSS=2 BBsematest.hny
harmony -b -c NSLOTS=1 -c NPRODS=2 -c NCONSS=0 BBsematest.hny
harmony    -c NSLOTS=1 -c NPRODS=2 -c NCONSS=1 BBsematest.hny
harmony    -c NSLOTS=1 -c NPRODS=2 -c NCONSS=2 BBsematest.hny
harmony -b -c NSLOTS=1 -c NPRODS=2 -c NCONSS=3 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=1 -c NCONSS=0 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=1 -c NCONSS=1 BBsematest.hny
harmony -b -c NSLOTS=2 -c NPRODS=1 -c NCONSS=2 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=2 -c NCONSS=0 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=2 -c NCONSS=1 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=2 -c NCONSS=2 BBsematest.hny
harmony -b -c NSLOTS=2 -c NPRODS=2 -c NCONSS=3 BBsematest.hny
harmony -b -c NSLOTS=2 -c NPRODS=3 -c NCONSS=0 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=3 -c NCONSS=1 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=3 -c NCONSS=2 BBsematest.hny
harmony    -c NSLOTS=2 -c NPRODS=3 -c NCONSS=3 BBsematest.hny
```

Figure 14.4: Script for testing producer/consumer synchronization.

```
import BBsema;

const NPRODS = 2;        # number of producers
const NCONSS = NPRODS;   # number of consumers

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 |= { first, second };
        ;
    ;
;
def main():
    await len(received) == (2 * NCONSS);
    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 14.5: [code/BBsemadata.hny] Test whether the bounded buffer delivers the right data in the right order.

```
const N = 10;

def gpuAlloc():
    result = choose(availGPUs);
    availGPUs -= { result };
;
def gpuRelease(gpu):
    availGPUs |= { gpu };
;
availGPUs = {1..N};
```

Figure 14.6: [code/gpu.hny] A thread-unsafe GPU allocator.

Exercises

14.1 Change the code in Figure 14.2 so that `produce()` fails if the buffer is full rather than block. `produce()` should return `False` if it fails and return `True` if it succeeds.

14.2 Optimize the code in Figure 14.2 for the case where there is only one consumer and one producer.

14.3 Implement a *bounded stack*. Method `push`(*item*) should block if the stack is full. Method `pop()` should block until the stack is non-empty and then return an item. Also implement test programs for your solution.

14.4 Implement a thread-safe *GPU allocator* by modifying Figure 14.6. There are `N` GPUs identified by the numbers 1 through `N`. Method `gpuAlloc()` returns the identifier of an available GPU, blocking if there is currently no available GPU. Method `gpuRelease`(*gpu*) releases the given GPU. It never needs to block.

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 condition, 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 releases the baton for some other process to pick up at a later time.

```
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 += 1;
        V(?mutex); P(?r_sema);
        r_waiting −= 1;
    ;
    r_entered += 1;
    V_one();
;
def release_rlock():
    P(?mutex);
    r_entered −= 1;
    V_one();
;
def acquire_wlock():
    P(?mutex);
    if (r_entered + w_entered) > 0:
        w_waiting += 1;
        V(?mutex); P(?w_sema);
        w_waiting −= 1;
    ;
    w_entered += 1;
    V_one();
;
def release_wlock():
    P(?mutex);
    w_entered −= 1;
    V_one();
;
mutex, r_sema, w_sema = Semaphore(1), Semaphore(0), Semaphore(0);
r_entered, r_waiting, w_entered, w_waiting = 0, 0, 0, 0;
```

Figure 15.1: [code/RWsbs.hny] Reader/Writer Lock using Split Binary Semaphores.

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 region, while a process that wants to write may have to wait until all readers have left the critical section or until a writer has left. 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 last semaphore 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 and 1 when not. 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. 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. For example, 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, passing the baton to the next reader. If there are writers waiting and there are no readers nor writers in the critical section, it vacates the semaphore associated with writers waiting, passing the baton to the next writer. If both conditions hold, it could vacate either one—it would not matter. However, it can only vacate one of the semaphores. In this implementation, it prefers to pass the baton to another reader. If none of the conditions hold, `V_one()` vacates *mutex*, leaving the batom for a future process to pick up.

Let us return now to `rlock_acquire`. In the case that the process, say p, 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 p by incrementing *r_sema*, p updates the state by decrementing *r_waiting* and incrementing *r_entered*. p then invokes `V_one()` to pass the baton to another waiting reader, if any, or to leave it for another process to pick up.

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*.

Exercises

15.1 Assume that you have only binary semaphores. Implement counting semaphores using the split binary semaphores technique.

15.2 With reader/writer locks, concurrency can be improved if a process *downgrades* its write lock to a read lock when its done writing but not done reading. Add a `downgrade` method to the code in Figure 15.1. How easy or hard would it be to do the same for the code in Figure 12.1? Note, you cannot simply release the write lock and then obtain the read lock, because the process needs to remain in the critical section.

15.3 Implement a solution to the producer/consumer problem (Figure 14.2) using split binary semaphores.

15.4 Cornell's campus features some one-lane bridges. On a one-lane bridge, cars can only go in one direction at a time. Consider northbound and southbound cars wanting to cross a one-lane bridge. The bridge allows arbitrary many cars, as long as they're going in the same direction. Implement a lock that observes this requirement using SBS. Write methods `nb_enter()` that a car must invoke before going northbound on the bridge and `nb_leave()` that the car must invoke after leaving the bridge. Similarly write `sb_enter()` and `sb_leave()` for southbound cars.

15.5 Extend the solution to Exercise 15.4 by implementing the requirement that at most 3 cars are allowed on the bridge.

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 keep entering, 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. We shall call such a solution *fair* (although it does not quite match the usual formal nor vernacular concepts of fairness).

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

```
import synch;

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 += 1; V(?mutex);
        P(?r_sema); r_waiting −= 1;
    ;
    r_entered += 1;
    V_one_r();     # only other readers can enter
;
def release_rlock():
    P(?mutex); 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 += 1; V(?mutex);
        P(?w_sema); w_waiting −= 1;
    ;
    w_entered = 1;
    V(?mutex);    # no other process can enter
;
def release_wlock():
    P(?mutex); w_entered = 0;
    V_one_r();     # other readers have right of way
;
mutex, r_sema, w_sema = Semaphore(1), Semaphore(0), Semaphore(0);
r_entered, r_waiting, w_entered, w_waiting = 0, 0, 0, 0;
```

Figure 16.1: [code/RWfair.hny] Reader/Writer Lock SBS implementation addressing fairness.

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.

SBSs allow much control over which thread runs next and is therefore a good starting point for developing fair synchronization algorithms. In this chapter, we will present two SBS-based fair reader/writer lock implementations. The first, in Figure 16.1, is based on Figure 15.1. There are two differences:

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

2. There are two versions of `V_one()`. In particular, when the last reader leaves the critical section, it preferentially passes the baton to a waiting writer rather than to a waiting reader. Similarly, when a 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 cannot starve one another. And since contention for a semaphore is resolved probabilistically, each reader and each writer will eventually have a chance to enter the critical section almost surely.

Figure 16.2 and Figure 16.3 present a different approach in which processes are allowed into the critical region in a purely First Come First Served manner. When a reader arrives, it is allowed in only if there is no writer in the critical section or waiting to enter. When a writer arrives, it is allowed in only if there is no reader or writer in the critical section (and therefore also not waiting to enter). To maintain the arrival order, the solution uses a semaphore per process and a queue. The `acquire_` methods take a pointer to the semaphore of the process as argument. Each entry in the queue is a pair consisting of a type of access (`.read` or `.write`) and a pointer to the semaphore of the processs that is waiting. Method `V_one` considers semaphores to vacate in the order in which they appear in the queue.

Exercises

16.1 Write a fair solution to the one-lane bridge problem of Exercise 15.4. If you want to use the queue method, you can change the signature of the methods to enter the bridge, adding an argument that contains a pointer to a semaphore.

```
import synch;
import list;

def V_one():
    if queue == []:
        V(?mutex);
    else:
        let h = head(queue):
            if (h[0] == .read) and (w_entered == 0): V(h[1]);
            elif (h[0] == .write) and ((r_entered + w_entered) == 0): V(h[1]);
            else: V(?mutex);
            ;
        ;
    ;
;
def acquire_rlock(psema):
    P(?mutex);
    if (w_entered > 0) or (queue != []):
        queue = append(queue, (.read, psema)); V(?mutex);
        P(psema); queue = tail(queue);
    ;
    r_entered += 1;
    V_one();
;
def release_rlock():
    P(?mutex); r_entered −= 1; V_one();
;
def acquire_wlock(psema):
    P(?mutex);
    if (r_entered + w_entered) > 0:
        queue = append(queue, (.write, psema)); V(?mutex);
        P(psema); queue = tail(queue);
    ;
    w_entered = 1;
    V_one();
;
def release_wlock():
    P(?mutex); w_entered = 0; V_one();
;
mutex = Semaphore(1);
r_entered, w_entered = 0, 0;
queue = [];
```

Figure 16.2: [code/RWqueue.hny] Reader/Writer Lock SBS implementation using a queue.

```
import RWqueue;

const NPROCS = 3;

def process(self):
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            acquire_rlock(?semas[self]);
            @rcs: assert atLabel.wcs == dict{};
            release_rlock();
        else:                    # .write
            acquire_wlock(?semas[self]);
            @wcs: assert (atLabel.wcs == dict{ nametag(): 1 }) and
                    (atLabel.rcs == dict{})
                ;
            release_wlock();
        ;
    ;
;
semas = [ Semaphore(0) for i in {0..NPROCS–1} ];
for i in {0..NPROCS–1}:
    spawn process(i);
;
```

Figure 16.3: [code/RWqtest.hny] Test program for Figure 16.2.

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 Harmony, are as follows:

```
def wait(c):
    V(?mutex);
    P(c);
;
def signal(c):
    V(c);
    P(?mutex);
;
```

`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` and `V`.

In the late 70s, researchers at Xerox PARC, where among others the desktop and Ethernet were invented, 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.

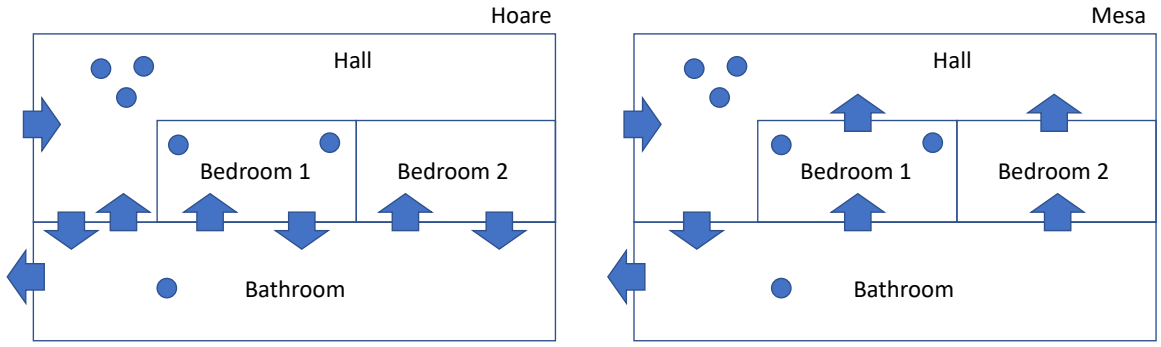

Figure 17.1: High-level depictions of Hoare and Mesa monitors. The hall is where processes wait to enter the bathroom (the critical section). The bedrooms illustrate two condition variables. The circles are processes.

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 variables that a process can wait on. Like in Hoare monitors, the `wait` operation releases 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 (assuming there even is such a process). Instead, p holds onto the baton until it leaves the monitor. At that point, any process that was notified will have a chance to obtain the baton, but they compete with other processes also trying to obtain the baton.

Figure 17.1 illustrates the difference with a drawing. Here a bathroom represents the critical section, allowing only one process at a time. The bedrooms represent condition variables. There is some condition associated with each bedroom. In the hall are processes waiting to enter the critical section.

On the left is a Hoare monitor. When some process p is in the bathroom, it can invoke **signal** on some non-empty bedroom associated with a condition that holds. This select a process q in that bedroom. As a result of invoking **signal**, p goes back into the hall and q enters the bathroom.[1] Importantly, when q enters the bathroom, the condition that q was waiting for is guaranteed to hold.

On the right is a Mesa monitor. The only way into the bathroom is through the hall. When process p invokes **notify** on a bedroom, it selects a process q in that bedroom, if any. Process q goes into the hall, joining any other processes that are also waiting to enter the bathroom. If there are no processes in that bedroom, **notify** is a no-op. Importantly, in either case p remains in the bathroom. Finally, when p leaves the bathroom, one of the processes in the hall can enter 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 the condition it was waiting for still 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.

[1] Per Brinch Hansen' monitors have a **continue** operation instead of **signal**. The difference is that processes that invoke **continue** leave the monitor rather than returning to the hall.

Besides notifying processes, Mesa monitors provide another important option: 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 the specified bedroom into the hall. Mesa also provides an operation called `notifyAll` (aka `broadcast`) that, when applied to a bedroom, lets all processes in that bedroom into the hall. This is not possible with Hoare monitors: when needing to wake up a set of processes, only one can be signaled at a time. Each of those processes then has to signal the next in a baton-passing style until all processes have been signaled.

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`. In addition, Java 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.

The Harmony `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 17.2 shows the implementation of condition variables in the `synch` module. `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 uses 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 using, once again, an implementation of reader/writer locks. Figure 17.3 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 above: 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 there is always a **while** loop around any invocation of `wait` containing the negation of the condition that the process is waiting for. Many implementation of condition variables depend on this, and optimized implementations of condition variables often allow so-called "spurious wakeups," where `wait` may sometimes return even if the conditon variable 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`(?*rcond*) 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

```
import bag;

def Condition(lk):
    result = dict{ .lock: lk, .waiters: bagEmpty() };
;
def wait(c):
    let lk, blocked = c→lock, True
    let 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 17.2: [modules/synch.hny] Implementation of condition variables in the `synch` module.

```
import synch;

def acquire_rlock():
    lock(?rwlock);
    while nwriters > 0:
        wait(?rcond);
    ;
    nreaders += 1;
    unlock(?rwlock);
;
def release_rlock():
    lock(?rwlock);
    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 17.3: [code/RWcv.hny] Reader/Writer Lock using Mesa-style condition variables.

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.

Much of the complexity of programming with Mesa condition variables is in figuring out when to invoke `notify` and when to invoke `notifyAll`. As a rule of thumb: be conservative—it is better to wake up too many processes than too few. Waking up too many processes may lead to some inefficiency, but waking up too few may cause the application to get stuck. Harmony can be particularly helpful here, as it examines every corner case. You can safely try to replace each `notifyAll` with `notify` and see if every possible execution of the the application still terminates.

While Hoare moniters embody the split binary semaphore approach, Mesa monitors are much closer to the busy-waiting approach. When comparing the busy-waiting version of reader/writer locks in Figure 11.1 and the Mesa version in Figure 17.3, you will see important similarities. Busy-waiting is "fixed" by placing a **wait** statement in the waiting loop. The Mesa code then has to be careful to invoke **notify** or **notifyAll** in the right places.

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

Exercises

17.1 Implement a "try lock" using Mesa condition variables (see also Exercise 9.3). It should have the following API:

1. *tl* = `TryLock`() # *allocate a try lock*
2. `tlLock`(?*tl*) # *acquire a try lock*
3. `tlTrylock`(?*tl*) # *attempt to acquire a try lock*
4. `tlUnlock`(?*tl*) # *release a try lock*

`tlTrylock` should not wait. Instead it should return `True` if the lock was successfully acquired and `False` if the lock was not available.

17.2 Implement semaphores using Mesa condition variables.

17.3 Write a new version of the GPU allocator in Exercise 14.4 using Mesa condition variables. In this version, a process is allowed to allocate a set of GPUs and release a set of GPUs that it has allocated. Method `gpuAllocSet(n)` should block until n GPUs are available, but it should grant them as soon as they are available. Method `gpuReleaseSet(s)` takes a set of GPU identifiers as argument. A process does not have to return all the GPUs it allocated at once.

17.4 The specification in the previous question makes the solution unfair. Explain why this is so. Then change the specification and the solution so that it is fair.

Chapter 18

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 philosopher likes to take breaks from eating to think for a while. To do so, the philosopher puts down both forks. Each philosopher repeat this procedure. Dijkstra had them repeating this for ever, but for the purposes of this book, philosophers can leave the table when they're not eating.

Figure 18.1 implements the dining philosophers in Harmony, using a process for each philosopher and a lock for each fork. If you run it, Harmony 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?
- Does it matter in what order the philosophers lay down their forks?

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 away 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 22). Havender proposed the following techniques that avoid the remaining three conditions [Hav68]:

```
import synch;

const N = 5;

def diner(which):
    let left, right = (which, (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 18.1: [code/Diners.hny] Dining Philosophers.

```
import synch;

const N = 5;

def diner(which):
    let left, right = (which, (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 18.2: [code/DinersCV.hny] Dining Philosophers that grab both forks at the same time.

- *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 18.2 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 negation of the condition that the diner is waiting and consists of two disjuncts. Within the **while** statement, there is an **if** statement for each disjunct. 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:

```
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 Harmony 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 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:

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

This completes all the Havender methods. There is, however, another approach, which is sometimes called deadlock *avoidance* instead of deadlock *prevention*. 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 18.3 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 the set of processes and what the maximum number of resources is that each process wants to allocate, making them generally quite impractical.

Exercises

18.1 Figure 18.4 shows an implementation of a bank with various accounts and transfers between those accounts. Unfortunately, running the test reveals that it sometimes leaves unterminated processes. Can you fix the problem?

18.2 Add a method `total()` to the solution of the previous question that computes the total over all balances. It needs to obtain a lock on all accounts. Make sure that it cannot cause deadlock.

18.3 Implement an extra process that checks, once, if the total of the balances is the same as that at the beginning (using an assertion). Is once enough to determine that no money gets lost or create, or should it do so continually?

```
import synch;

const N = 5;

def diner(which):
    let left, right = (which, (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 18.3: [code/DinersAvoid.hny] Dining Philosophers that carefully avoid getting into a deadlock scenario.

```
import synch;

const MAX = 2;
const NACCOUNTS = 2;
const NPROCESSES = 2;

def transfer(a1, a2, amount):
    lock(?accounts[a1].lock);
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount;
        lock(?accounts[a2].lock);
        accounts[a2].balance += amount;
        unlock(?accounts[a2].lock);
        result = True;
    else:
        result = False;
    ;
    unlock(?accounts[a1].lock);
;
def process():
    let a1 = choose({0..NACCOUNTS–1})
    let a2 = choose({0..NACCOUNTS–1} – { a1 }):
        transfer(a1, a2, choose({0..MAX}));
    ;
;
accounts = [ dict{ .lock: Lock(), .balance: choose({0..MAX}) }
             for i in {0..NACCOUNTS–1} ]
;
for i in {1..NPROCESSES}:
    spawn process();
;
```

Figure 18.4: [code/bank.hny] Bank accounts.

Chapter 19

Actors and Message Passing

Some programming languages favor a different way of implementing synchronization using so-called *actors* [HBS73]. Actors are processes that have only private memory and communicate through *message passing*. See Figure 19.1 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. As message queues are FCFS (First-Come-First-Served), starvation is prevented.

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:

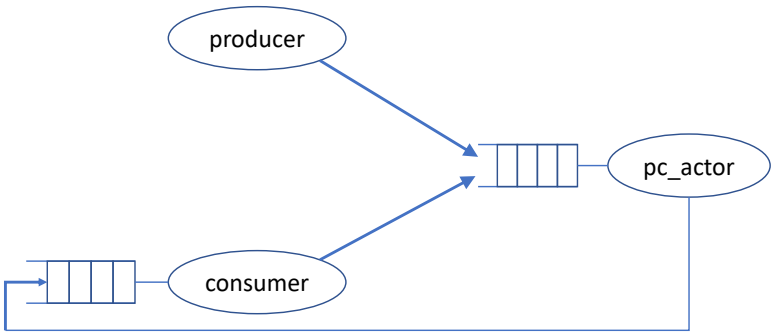

Figure 19.1: Depiction of three actors. The producer does not receive messages.

```
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 |= { req };
                    else:
                        let r = choose(requests):
                            assert r.type == .consume;
                            enqueue(r.queue, req.item);
                            requests -= { r };
                        ;
                    ;
                    balance += 1;
                else:
                    assert req.type == .consume;
                    if balance <= 0:
                        requests |= { req };
                    else:
                        let r = choose(requests):
                            assert r.type == .produce;
                            enqueue(req.queue, r.item);
                            requests -= { r };
                        ;
                    ;
                    balance -= 1;
                ;
            ;
            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 19.2: [code/actor.hny] A producer/consumer actor.

```
import actor;

const NITEMS = 3;

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(?pc_queue, i);
    spawn consume(?pc_queue, ?queues[i]);
;
```

Figure 19.3: [code/actortest.hny] Test code for producer/consumer actor.

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

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 19.2 shows a Harmony solution to the "multiple producer / multiple consumer problem" (with an unbounded buffer) using an actor (as illustrated in Figure 19.1). `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 Harmony all processes are required to terminate. In a more realistic implementation, the `pc_actor` would be in an infinite loop awaiting requests.

The shared memory consists entirely of message queues. 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 19.3 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.

Exercises

19.1 A popular model is the *client/server* model. Here a single actor implements some service, like computing the square of a number. A client can send a message containing an integer to the server, and the server returns a response message containing the square of that integer. Implement this.

19.2 Actors and message queues are good for building pipelines. Develop a pipeline that computes Mersenne primes (primes that are one less than a power of two). Write four actors:

1. an actor that generates a sequence of integers 1 through `N`;
2. an actor that receives integers and forwards only those that are prime;
3. an actor that receives integers and forwards only those that are one less than a power of two;
4. an actor that receives integers but otherwise ignores them.

Configure two versions of the pipeline, one that first checks if a number is prime and then if it is one less than a power of two, the other in the opposite order. Which do you think is better?

Chapter 20

Interrupts

Processes can be *interrupted*. An interrupt is a notification of some event such as a keystroke, a timer expiring, the reception of a network packet, the completion of a disk operation, and so on. We distinguish *interrupts* and *exceptions*. An exception is caused by the process executing an invalid machine instruction such as divide-by-zero. An interrupt is caused by some peripheral device and can be handled in Harmony. In other words: an interrupt is a notification, while an exception is an error.

Harmony allows modeling interrupts using the `trap` statement:

trap *handler argument*

invokes *handler argument* at some later, unspecified time. Only one of these asynchronous events can be outstanding at a time; a new call to **trap** overwrites any outstanding one. Figure 20.1 gives an example of how **trap** might be used. Here, the *main*() process loops until the interrupt has occurred and the *done* flag has been set.

But now consider Figure 20.2. The difference with Figure 20.1 is that both the *main*() and *handler*() methods increment *count*. This is not unlike the example we gave in Figure 3.1, except that only a single process is involved now. And, indeed, it suffers from a similar race condition; run it through Harmony to see for yourself. If the interrupt occurs after *main*() reads *count* (and thus still has value 0) but before *main()* writes the updated value 1, then the interrupt handler will also read value 0 and write value 1. We say that the code in Figure 20.2 is not *interrupt-safe* (as opposed to not being *thread-safe*).

You would be excused if you wanted to solve the problem using locks, similar to Figure 9.3. Figure 20.3 shows how one might go about this. But locks are intended to solve synchronization issues between multiple processes. If you run the code through Harmony, you will find that the code may not terminate. The issue is that a process can only acquire a lock once. If the interrupt happens after *main*() acquires the lock but before *main*() releases it, the *handler*() method will block trying to acquire the lock, even though it is being acquired by the same process that already holds the lock.

Instead, the way one fixes interrupt-safety issues is through disabling interrupts temporarily. In Harmony, this can be done by setting the *interrupt level* of a process to `True` using the **setintlevel** interface. Figure 20.4 illustrates how this is done. Note that it is not necessary to change the

```
def handler():
    count += 1;
    done = True;
;
def main():
    trap handler();
    await done;
    assert count == 1;
;
count = 0;
done = False;
spawn main();
```

Figure 20.1: [code/trap.hny] How to use **trap**.

```
def handler():
    count += 1;
    done = True;
;
def main():
    trap handler();
    count += 1;
    await done;
    assert count == 2;
;
count = 0;
done = False;
spawn main();
```

Figure 20.2: [code/trap2.hny] A race condition with interrupts.

```
import synch;

def handler():
    lock(?countlock);
      count += 1;
    unlock(?countlock);
    done = True;
;
def main():
    trap handler();
    lock(?countlock);
      count += 1;
    unlock(?countlock);
    await done;
    assert count == 2;
;
countlock = Lock();
count = 0;
done = False;
spawn main();
```

Figure 20.3: [code/trap3.hny] Locks do not work with interrupts.

```
import synch;

def handler():
    count += 1;
    done = True;
;
def main():
    trap handler();
    setintlevel(True);
      count += 1;
    setintlevel(False);
    await done;
    assert count == 2;
;
mutex = Lock();
count = 0;
done = False;
spawn main();
```

Figure 20.4: [code/trap4.hny] Disabling and enabling interrupts.

interrupt level during servicing an interrupt, because it is automatically set to `True` upon entry to the interrupt handler and restored to `False` upon exit. It is important that the *main*() code re-enables interrupts after incrementing *count*. What would happen if *main*() left interrupts disabled?

setintlevel(*il*) sets the interrupt level to *il* and returns the prior interrupt level. Returning the old level is handy when writing interrupt-safe methods that can be called from ordinary code as well as from an interrupt handler. Figure 20.5 shows how one might write a interrupt-safe method to increment the counter.

It will often be necessary to write code that is both interrupt-safe *and* thread-safe. As you might expect, this involves both managing locks and interrupt levels. To increment *count*, the interrupt level must be *True* and *countlock* must be held. Figure 20.6 gives an example of how this might be done. One important rule to remember is that a process should disable interrupts *before* attempting to acquire a lock.

Try moving `lock`() to the beginning of the `increment` method and `unlock`() to the end of `increment` and see what happens. While Harmony will only report one faulty run, this incorrect code can lead to the assertion failing as well as processes getting blocked indefinitely.

(Another option is to use synchronization techniques that do not use locks. See Chapter 22 for more information.)

There is another important rule to keep in mind. Just like locks should never be held for long, interrupts should never be disabled for long. With locks the issue is to maximize concurrent performance. For interrupts the issue is fast response to asynchronous events. Because interrupts may be disabled only briefly, interrupt handlers must run quickly and cannot wait for other events. So it is ok to invoke synchronization calls such as `V` and `notify`, but calls such as `P` and `wait` should

```
import synch;

def increment():
    let prior = setintlevel(True):
        count += 1;
        setintlevel(prior);
    ;
;
def handler():
    increment();
    done = True;
;
def main():
    trap handler();
    increment();
    await done;
    assert count == 2;
;
mutex = Lock();
count = 0;
done = False;
spawn main();
```

Figure 20.5: [code/trap5.hny] Example of an interrupt-safe method.

```
import synch;

def increment():
    let prior = setintlevel(True):
        lock(?countlock);
            count += 1;
        unlock(?countlock);
        setintlevel(prior);
    ;
;
def handler(self):
    increment();
    done[self] = True;
;
def process(self):
    trap handler(self);
    increment();
    await done[self];
;
def main(self):
    await all(done);
    assert count == 4, count;
;
count = 0;
countlock = Lock();
done = [ False, False ];
spawn process(0);
spawn process(1);
spawn main();
```

Figure 20.6: [code/trap6.hny] Code that is both interrupt-safe and thread-safe.

only be used if it is certain that they will not block for long. Informally, interrupt handlers can be *producers* but not *consumers* of synchronization events.

Exercises

20.1 The `produce` method in Figure 14.2 cannot be used in interrupt handlers for two reasons: (1) it is not interrupt-safe, and (2) it may block for a long time if the buffer is full. Yet, it would be useful if, say, a keyboard interrupt handler could place an event on a shared queue. Similar to Exercise 14.1, implement a new method `iproduce`(*item*) that does not block. Instead, it should return `False` if the buffer is full and `True` if the item was successfully enqueued. The method also needs to be interrupt-safe.

Chapter 21

Alternating Bit Protocol

A *distributed system* is a concurrent system in which a collection of processes communicate by message passing, much the same as in the actor model. The most important difference between distributed and concurrent systems is that the former takes *failures* into account, including failures of processes and failures of shared memory. In this chapter, we will consider two actors, Alice and Bob. Alice wants to send a sequence of application messages to Bob, but the underlying network may lose messages. The network does not re-order messages: when sending messages m_1 and m_2 in that order, then if both messages are received, m_1 is received before m_2. Also, the network does not create messages out of nothing: if message m is received, then message m was sent.

It is useful to create an abstract network that reliably sends messages between processes, much like the FIFO queue in the `synch` module. For this, we need a network protocol that Alice and Bob can run. In particular, it has to be the case that if Alice sends application messages $m_1, ..., m_n$ in that order, then if Bob receives an application message m, then $m = m_i$ for some i and Bob will already have received application messages $m_1, ..., m_i$ (safety). Also, if the network is fair and Alice sends application message m, then eventually Bob should deliver m (liveness).

The *Alternating Bit Protocol* is suitable for our purposes. We assume that there are two unreliable network channels: one from Alice to Bob and one from Bob to Alice. Alice and Bob each maintain a zero-initialized *sequence number*, *s_seq* and *r_seq* resp. Alice sends a network message to Bob containing an application message as *payload* and Alice's sequence number as *header*. When Bob receives such a network message, Bob returns an *acknowledgment* to Alice, which is a network message containing the same sequence number as in the message that Bob received.

In the protocol, Alice keeps sending the same network message until she receives an acknowledgment with the same sequence number. This is called *retransmission*. When she receives the desired sequence number, Alice increments her sequence number. She is now ready to send the next message she wants to send to Bob. Bob, on the other hand, waits until he receives a message matching Bob's sequence number. If so, Bob *delivers* the payload in the message and increments his sequence number. Because of the network properties, a one-bit sequence number suffices.

We can model each channel as a variable that either contains a network message or nothing (we use `()` in the model). Let *s_chan* be the channel from Alice to Bob and *r_chan* the channel from Bob to Alice. `net_send`(*pchan*, *m*, *reliable*) models sending a message m to `!`*pchan*, where *pchan* is either ?*s_chan* or ?*r_chan*. The method places either m (to model a successful send) or `()` (to

```
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, blocked = dict{ .seq: s_seq, .payload: payload }, 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 21.1: [code/abp.hny] Alternating Bit Protocol.

```
import abp;

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 21.2: [code/abptest.hny] Test code for alternating bit protocol.

model loss) in `!`*pchan*. The use of the *reliable* flag will be explained later. `net_recv`(*pchan*) models checking `!pchan` for the next message.

Method `app_send`(m) retransmits m until an acknowledgment is received. Method `app_recv`(*reliable*) returns the next successfully received message. Figure 21.2 shows how the methods may be used to send and receive a stream of `NMSGS` messages reliably. It has to be bounded, because model checking requires a finite model.

Only the last invocation of `app_recv`(*reliable*) is invoked with *reliable* == `True`. It causes the last acknowledgment to be sent reliably. It allows the receiver (Bob) to stop, as well as the sender (Alice) once the last acknowledgment has been received. Without something like this, either the sender may be left hanging waiting for the last acknowledgment, or the receiver waiting for the last message.

Exercises

21.1 Exercise 19.1 explored the *client/server model*. It is popular in distributed systems as well. Develop a protocol for a single client and server using the same network model as for the ABP protocol. Hint: the response to a request can contain the same sequence number as the request.

21.2 Generalize the solution in the previous exercise to multiple clients. Each client is uniquely identified. You may either use separate channel pairs for each client, or solve the problem using a single pair of channels.

Chapter 22

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 executing the critical section. Also, using synchronization primitives in interrupt handlers is tricky to get right (Chapter 20) and might run too long. In this chapter, we will have a look at how one can develop concurrent code in which processes do not have to wait for other processes to complete their ongoing operations.

As a first example, we will revisit the producer/consumer problem. The code in Figure 22.1 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;
- `exch`(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 `exch()` 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. Unfortunately, the solution is not practical for the following reasons:

- The *items* array must be of infinite size if an unbounded number of items may be produced;
- Each slot in the array is only used once, which is inefficient;
- the `inc` and `exch` interlock instructions are not universally available on existing processors.

```
const MAX_ITEMS = 3;

def inc(pcnt):
    atomic:
        result = !pcnt;
        !pcnt += 1;
    ;
;
def exch(pv):
    atomic:
        result = !pv;
        !pv = ();
    ;
;
def produce(item):
    items[inc(?back)] = (item,);
;
def consume():
    result = ();
    while result == ():
        let range, i = back, 0:
            while (i < range) and (result == ()):
                result = exch(?items[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 22.1: [code/hw.hny] Non-blocking queue.

However, in the literature there are many examples of practical non-blocking (aka *wait-free*) synchronization algorithms.

There are also various algorithms where read-only access is wait-free but updates do require a lock—these are useful in situations where most accesses are read-only and the performance of updates is less critical. Figure 22.2 and Figure 22.3 implements an ordered linked list of integers without duplicates. There are two update operations: values can be added using `lst_insert` or deleted using `lst_remove`. There is also a read-only operation: `lst_contains` checks if a particular value is in the list.

The read-only operation `lst_contains` is wait_free: it scans through the list without having to obtain a lock. Nonetheless, the implementation of the list is *linearizable* [HW90], a strong notion of consistency that makes it appear as if each of the operations executes atomically at some point between their invocation and return.

The list has two "book-end" nodes with values `-inf` and `inf` (similar to the Python `math.inf` constant). Each node has a lock, a value, and *next*, a pointer to the next node (which is `None` for the final `inf` node). The `lst_find`(*lst*, *v*) method is a helper function for update methods that finds and locks two consecutive nodes *before* and *after* such that *before*.`data.value` $< v \leq$ *after*.`data.value`. It does so by performing something called *hand-over-hand locking*. It first locks the first node, which is the `-inf` node. Then, iteratively, it obtains a lock on the next node and release the lock on the last one, and so on, similar to climbing a tree hand-over-hand.

Having the two adjoining nodes locked, implementing `lst_insert` and `lst_remove` is straightforward. The `lst_contains` method can simply scan through the list because the *next* pointers are updated atomically in such a way that they always point to a legal suffix of the list. An invariant of the algorithm is that at any point in time the list is "valid," starting with a `-inf` node and ending with a `inf` node.

Determining if an implementation of a concurrent data structure is linearizable involves finding what are known as the *linearization points* of the operations in an execution. These are the unique points in time at which an operation appears to execute atomically. The linearization points for the `lst_insert` and `lst_remove` operations coincide exactly with the update of the *before*.`next` pointer. The linearization point of a `lst_contains` method execution depends on whether the value is found or not. If found, it coincides with retrieving the pointer to the node that has the value. If not found, it coincides with retrieving the pointer to the `inf` node.

Exercises

22.1 Add read-only methods to the data structure in Figure 22.2 and Figure 22.3 that report the size of the list, the minimum value in the list, the maximum value in the list, and the sum of the values in the list. Are they linearizable? If so, what are their linearization points?

22.2 A *seqlock* consists of a lock and a version number. An update operation acquires the lock, increments the version number, makes the changes to the data structure, and then releases the lock. A read-only operation does not use the lock. Instead, it retrieves the version number, reads the data structure, and then checks if the version number has changed. If so, the read-only operation is retried. Use a seqlock to implement a bank much like Exercise 18.1, with one seqlock for the entire bank (i.e., no locks on individual accounts). Method `transfer` is an update operation; method `total` is a read-only operation. Explain how a seqlock can lead to starvation.

```
import synch;
import alloc;

def lst_node(v, n):     # allocate and initialize a new list node
    result = malloc(dict{ .lock: Lock(), .value: v, .next: n });
;
def lst_contains(lst, v):   # check if v is in the list
    let n = lst:
        while n→value < v:
            n = n→next;
        ;
        result = n→value == v;
    ;
;
```

Figure 22.2: [code/lst1.hny] List with non-blocking read operation, part 1.

```
import lst1;

def lst_find(lst, v):   # find 'neighboring' nodes of v in list
    let before = lst let after = before→next:
        lock(?before→lock);
        lock(?after→lock);
        while after→value < v:
            unlock(?before→lock);
            before = after;
            after = before→next;
            lock(?after→lock);
        ;
        result = (before, after);
    ;
;
def lst_insert(lst, v):
    let before, after = lst_find(lst, v):
        if after→value != v:
            before→next = lst_node(v, after);
        ;
        unlock(?before→lock);
        unlock(?after→lock);
    ;
;
def lst_remove(lst, v):
    let before, after = lst_find(lst, v):
        if after→value == v:
            before→next = after→next;
            unlock(?after→lock);
            free(after);
        else:
            unlock(?after→lock);
        ;
        unlock(?before→lock);
    ;
;
mylist = lst_node(−inf, lst_node(inf, None));
lst_insert(mylist, 3);
assert lst_contains(mylist, 3);
lst_remove(mylist, 3);
assert not lst_contains(mylist, 3);
```

Figure 22.3: [code/lst2.hny] List with non-blocking read operation, part 2.

Chapter 23

Barrier Synchronization

Barrier synchronization is a problem that comes up in high-performance parallel computing, used, among others, for scalable simulation. A barrier is almost the opposite of a critical section: the intention is to get a group of processes to run some code at the same time, instead of having them execute it one at a time. More precisely, with barrier synchronization the processes execute in rounds. Between each round there is a so-called *barrier* where processes wait until all processes have completed the previous round, before they start the next one. For example, in an iterative matrix algorithm, the matrix may be cut up into fragments. During a round, the processes run concurrently, one for each fragment. The next round is not allowed to start until all processes have completed processing their fragment.

Figure 23.1 shows a high-level depiction of barrier synchronization with three and 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 23.2 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

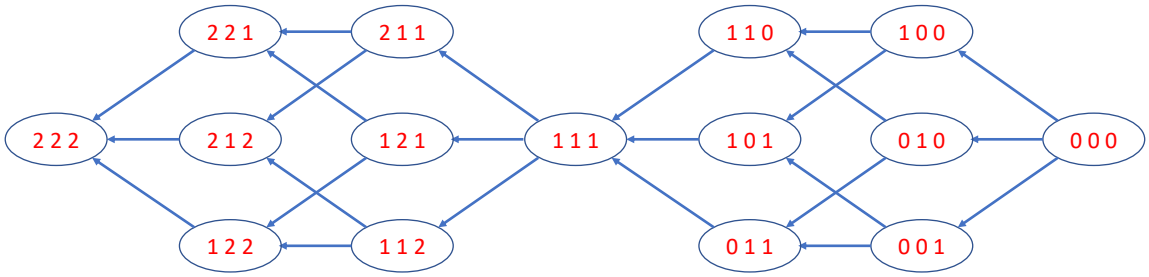

Figure 23.1: High-level state diagram specification of barrier synchronization with three processes and two rounds. The three numbers specify how many rounds each process has completed.

```
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} such that i != self:
            V(?sema[i]);
        ;
        for i in {0..N–1} such that i != self:
            P(?sema[self]);
        ;
        round[self] = (round[self] + 1);
        assert (max(round) – min(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 23.2: [code/barrier.hny] Barrier synchronization with semaphores.

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.

Exercises

23.1 See if you can implement barrier synchronization for `N` processes with just three semaphores. Busy waiting is not allowed. How about just two? (As always, the Little Book of Semaphores [Dow09] is a good resource for solving synchronization problems with semaphores. Look for the *double turnstile* solution.)

23.2 Implement barrier synchronization with Mesa condition variables. (You may want to use a double turnstile approach here as well.)

Bibliography

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

[Bir89] Andrew D. Birrell. An introduction to programming with threads. SRC report 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.

[Dij72] Edsger W. Dijkstra. EWD-329 information streams sharing a finite buffer. 1972.

[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.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[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.

[MS98] Maged M. Michael and Michael L. Scott. Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[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 Harmony values. Below is a complete list of Harmony value types with examples:

Boolean	`False`, `True`
Integer	`-inf, ..., -2, -1, 0, 1, 2, ..., inf`
Atom	`.example`, `.test1`, `.0x4A`
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], None`
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 Harmony 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 is represented as a tuple of its characters. Characters are one-character atoms, which can be expressed in hexadecimal unicode using the syntax `.0xXX`. A bag or multiset is a dictionary that maps a value to how many times it occurs in the bag.

All Harmony 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. `None` is the empty list of atoms. Addresses are lexicographically ordered accordingly;
- contexts (Appendix F) are ordered first by name tag, then by program counter, then by hash.

Appendix B

List of Operators

Harmony currently supports the following operators:

e `==` *e*, *e* `!=` *e*	two Harmony values of the same type
e `<` *e*, *e* `<=` *e*, *e* `>` *e*, *e* `>=` *e*	any two Harmony values
e `and` *e* `and ...`	two or more booleans
e `or` *e* `or ...`	two or more booleans
`not` *e*	a boolean
`-`*e*, `~`*e*	an integer
e `+` *e* `+ ...`	two or more integers, tuples, or lists
e `-` *e*	two integers or sets
e `&` *e* `& ...`	two or more integers or sets
e `|` *e* `| ...`	two or more integers or sets
e `^` *e* `^ ...`	two or more integers or sets
e `*` *e*	two integers or an integer and a list
e `/` *e*, *e* `//` *e*	two integers (integer division)
e `%` *e*, *e* `mod` *e*	two integers (division remainder)
e `**` *e*, *e* `<<` *e*, *e* `>>` *e*	two integers
{*e*..*e*}	two integers
e `[not] in` *e*	first is any Harmony value, second is a set or dict
e `if` *e* `else` *e*	middle *e* must be a boolean
`!`*e*	an address
`choose` *e*	a set
`min` *e*, `max` *e*, `any` *e*, `all` *e*, `len` *e*	a set or a dict
`keys` *e*	a dictionary (includes tuple and list)
`bagsize` *e*	a bag
`atLabel` *e*	atom (corresponding to a label)
`nametag` *e*	*e* must be `()`
`processes` *e*	*e* must be `()`
`setintlevel` *e*	*e* is a boolean
`stop` *lv*	*lv* is an lvalue
`?`*lv*	*lv* is an lvalue
`lambda` *a*`:` *e* `end`	*a* is a bounded variable, *e* an expression

`+` concatenates when applied to lists or tuples. `-` computes set difference when applied to two sets. `&` computes the intersection when applied to sets. `|` computes the union when applied to sets. `^` computes $(s \cup t) \backslash (s \cap t)$ when applied to sets s and t. $\{x..y\}$ computes the set consisting of the integers x through y inclusive. It returns the empty set if $x > y$. If d is a dictionary, e `in` d tests if e is in the set of *values* of d, not the set of *keys* (this is different from Python).

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*.

Harmony also supports the following comprehensions:

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

In each case, s must be a set or a dictionary. Appendix H provides more information on comprehensions.

Appendix C

List of Statements

Harmony currently supports the following statements:

lv = [lv =]... *e*	*lv* is an lvalue and *e* is an expression
lv `[op]=` *e*	`op` is one of `+`, `-`, `*`, `/`, `//`, `%`, `\&`, `|`, `\^`, `and`, and `or`
`pass`	do nothing
`del` *lv*	delete
`assert` *b* `[,` *e*`]`	*b* is a boolean. Optionally report value of expression *e*
`const a =` *e*	`a` is a bounded variable, *e* is a constant expression
`def m` *a*`:  S`	`m` is an identifier, *a* a bounded variable, `S` a list of statements
`let` *a* `=` *e* `[let ...]:  S`	*a* is a bounded variable, *e* is an expression, `S` a list of statements
`if` *b*`:  S else:  S`	*b* is a boolean, `S` a list of statements
`while` *b*`:  S`	*b* is a boolean, `S` a list of statements
`await` *b*	*b* is a boolean
`for` *a* `in` *e*`:  S`	*a* is a bounded variable, *e* is a set, `S` a list of statements
`atomic:  S`	`S` a list of statements
`spawn m` *e* `[,` *t*`]`	`m` is a method, *e* is an expression, *t* is a tag (an expression)
`trap m` *e*	`m` is a method and *e* is an expression
`go` *c* *e*	*c* is a context, *e* is an expression
`import m`	`m` identifies a module

Appendix D

List of Modules

D.1 The `alloc` module

The `alloc` module supports thread-safe (but not interrupt-safe) dynamic allocation of shared memory locations. There are just two methods:

`malloc(`v`)`	return a pointer to a memory location initialized to v
`free(`p`)`	free an allocated memory location p

The usage is similar to `malloc` and `free` in C. `malloc`() is specified to return `None` when running out of memory, although this is an impossible outcome in the current implementation of the module.

D.2 The `bag` module

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

D.3 The `list` module

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 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
`reversed(`t`)`	reverse a list
`sorted(`t`)`	sorted set or list
`set(`t`)`	convert values of a dict or list into a set
`list(`t`)`	convert set into a list
`values(`t`)`	convert values of a dict into a list sorted by key
`items(`t`)`	convert dict into (key, value) list sorted by key
`enumerate(`t`)`	like Python enumerate
`sum(`t`)`	return the sum of all elements in t
`list2bag(`t`)`	convert list t into a bag
`listQsort(`t`)`	sort list t using quicksort

D.4 The `set` module

The `set` module implements the following methods:

`issubset(`s`, `t`)`	returns whether s is a subset of t
`issuperset(`s`, `t`)`	returns whether s is a superset of t

For Python programmers: note that $s <= t$ does not check if s is a subset of t when s and t are sets, as "<=" implements a total order on all Harmony values including sets.

D.5 The `synch` module

The `synch` module provides the following methods:

`tas(`lk`)`	test-and-set on `!`lk
`Lock()`	return a lock object
`lock(`lk`)`	acquire `!`lk
`unlock(`lk`)`	release `!`lk
`Condition(`lk`)`	return a condition variable on `!`lk
`wait(`c`)`	wait on condition variable `!`c
`notify(`c`)`	notify a process waiting on condition variable `!`c
`notifyAll(`c`)`	notify all processes waiting on condition variable `!`c
`Semaphore(`cnt`)`	return a semaphore object initialized to cnt
`P(`$sema$`)`	procure `!`$sema$
`V(`$sema$`)`	vacate `!`$sema$
`Queue()`	return a synchronized queue object
`dequeue(`q`)`	return next element of q, blocking if empty
`enqueue(`q, $item$`)`	add $item$ to a

Appendix E

List of Machine Instructions

Address	compute address from two components
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
Cut	pop set or dict and push minimum and remainder
Del [v]	delete shared variable v
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	start method `m` with arguments a, initializing variables.
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 the value of a shared variable onto the stack
LoadVar [v]	push the value of a process variable onto the stack
Move i	move stack element at offset i to top of 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
Return	pop return address, push `result`, and restore program counter
Set	pop n and n elements and push a set of the elements
SetIntLevel	pop e, set interrupt level to e, and push old interrupt level
Spawn	pop tag, argument, and method and spawn a new context
Split	pop tuple and push its elements
Stop [v]	save context into shared variable v and remove from context bag
Store [v]	pop a value from the stack and store it in a shared variable
StoreVar [v]	pop a value from the stack and store it in a process variable
Trap	pop interrupt argument and method

Clarifications:

- The `Address` instruction expects two values on the stack. The top value must be an address value, representing a dictionary The other value must be a key into the dictionary. The instruction then computes the address of the given key.
- Even though Harmony code does not allow taking addresses of process variables, both shared and process variables can have addresses.
- The `Load`, `LoadVar`, `Del`, `DelVar`, and `Stop` instructions have an optional variable name: if omitted the top of the stack must contain the address of the variable.
- `Store` and `StoreVar` instructions have an optional variable name. In both cases the value to be assigned is on the top of the stack. If the name is omitted, the address is underneath that value on the stack.
- 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 (`Jump` instructions are not allowed to jump to their own locations), 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	an integer value pointing into the code
frame pointer	an integer value pointing into the stack
atomic	if non-zero, the process is in atomic mode
stack	a list of Harmony values
register	a dictionary mapping atoms (names of variables) to values
stopped	a boolean indicating if the context is stopped
failure	if not None, string that describes how the process failed

Details:

- The name in a name tag is the name of the method that the process is executing;
- The tag in a name tag the argument to the method by default. Optionally `spawn` allows the tag to be specified explicitly;
- The frame pointer points to the current *stack frame*, which consists of the caller's frame pointer and variables, the argument to the method, an "invocation type atom" (`normal`, `interrupt`, or `process`), and the return address (in case of `normal`).
- A process terminates when it reaches the `Return` instruction of the top-level method (when the stack frame is of type `process`) or when it hits an exception. Exceptions include divide by zero, reading a non-existent key in a dictionary, accessing a non-existent variable, infinite loops of microsteps, 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 Harmony program.

Appendix G

The Harmony Virtual Machine

The Harmony Virtual Machine (HVM) has the following state:

code	a list of HVM machine instructions
labels	a dictionary of atoms to program counters
variables	a dictionary 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
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 HVM machine instruction by a context. Each micro step generates a new state. When there are multiple contexts, the HVM 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 HVM 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 Harmony program results in a graph where the nodes are Harmony 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.

Consecutive macro steps by the same process are called *mega steps*. Each state maintains the shortest path to it from the initial state in terms of mega steps. The diameter of the graph is the length of the longest path found.

If some states have a problem, the state with the shortest path is reported. Problematic states include states that experienced exceptions. If there are no exceptions, Harmony computes the strongly connected components (SCCs) of the graph (the number of such components are printed

as part of the output). The sink SCCs should each consist of a terminal state without any processes. If not, again the state with the shortest path is reported.

If there are no problematic states, Harmony reports "no issues found" and outputs in the HTML file the state with the longest path.

Appendix H

Harmony Language Details

The Harmony language borrows heavily from Python. However, there are some important differences that we will describe in this chapter.

H.1 Harmony is not object-oriented

Python is object-oriented, but Harmony is not. This can lead to some unexpected differences. For example, consider the following code:

```
1   x = [ 1, 2 ];
2   y = x;
3   x[0] = 3;
4   assert y[0] == 1;
```

In Python, lists are objects. Thus x and y point to the same list, and the assertion would fail if executed by Python. In Harmony, lists are values. So when x is updated in Line 3, it does not affect the value of y. The assertion succeeds. Harmony supports references to values (Chapter 7), allowing programs to implement shared objects.

Because Harmony does not have objects, it also does not have object methods. However, Harmony methods and lambdas are program counter constants. These constants can be added to dictionaries. For example, in Figure 7.1 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.

H.2 Constants, Global and Local Variables

Each (non-reserved) identifier in a Harmony program refers to either a constant, a global variable, or a local variable. Constants are declared using `const` statements. Those constants are computed at compile-time.

Local variables all declared. They can be declared in `def` statements (i.e., arguments), `let` statements, and in `for` loops. Also, each method has an implicitly declared *result* variable. Local variables are tightly scoped and cannot be shared between processes. While in theory one method can be declared within another, they cannot share variables either. All other variables are global and must be initialized before any processes are spawned.

While arguments to a method and variables in `for` loops can be modified, we discourage it for improved code readability.

H.3 Operator Precedence

In Harmony, there is no syntactic difference between applying an argument to a function or an index to a dictionary. Both use the syntax $a\ b\ c\ \ldots$. We call this *application*, and application is left-associative. So $a\ b\ c$ is interpreted as $(a\ b)\ c$: b is applied to a, and then c is applied to the result. For readability, it may help to write $a(b)$ for function application and $a[b]$ for indexing. In case b is an atom, you can also write $a.b$ for indexing.

There are three classes of precedence. Application has the highest precedence. So `!`$a\ b$ is interpreted as `!`$(a\ b)$ and $a\ b$ `+` $c\ d$ is interpreted as $(a\ b)$ `+` $(c\ d)$. Unary operators are have the next highest precedence, and the remaining operators have the lowest precedence. So $-2+3$ evaluates to 1, not -5.

Associative operators ($+$, $*$, $|$, &, ^, **and**, **or**) are interpreted as general n-ary operators, and you are allowed to write $a+b+c$. However, $a-b-c$ is illegal, as is any combination of operators with an arity larger than one, such as $a+b<c$. In such cases you have to add parentheses or brackets to indicate what the intended evaluation order is, such as $(a+b)<c$.

In almost all expressions, subexpressions are evaluated left to right. So $a[b]+c$ first evaluates a, then b (and then applies b to a), and then c. The one exception is the expression a **if** c **else** b, where c is evaluated first. In that expression, only a or b is evaluated depending on the value of c. In the expression a **and** b **and** ..., evaluation is left to right but stops once one of the subexpressions evaluates to **False**. Similarly for **or**, where evaluation stops once one of the subexpressions evaluates to `True`.

As an aside: the expression a **not in** b is equivalent to **not** (a **in** b). Harmony generalizes this construct for any pair of a unary (except '$-$') and a binary operator. In particular, a **not and** b is the same as **not** (a **and** b). For those familiar with logic gates, **not and** is the equivalent of `NAND`.

H.4 Tuples, Lists, and Pattern Matching

Harmony's tuples and, equivalently, lists, are just special cases of dictionaries. They can be bracketed either by '(' and ')' or by '[' and ']', but the brackets are often optional. Importantly, with a singleton list, the one element must be followed by a comma. So the statement x `= 1,;` assigns a singleton tuple (or list) to x.

Because tuples and lists are dictionaries, the `del` statement is different than in Python. For example, if x = [.a, .b, .c], then `del` x[1] results in x = `dict`{ 0:.a, 2:.c }, *not* x = [.a, .c]. Harmony also does not support special slicing syntax like Python. To modify lists, use the `subseq` method in the `list` module (Section D.3).

Harmony allows pattern matching against nested tuples in various language constructs. The following are the same in Python and Harmony:

- x, `= 1,`: assigns 1 to x;
- x`,` y `= 1, (2, 3)`: assigns 1 to x and (2, 3) to y;
- x`, (`y`,` z`) = 1, (2, 3)`: assigns 1 to x, 2 to y, and 3 to z;
- x`, (`y`,` z`) = 1, 2;` generates an runtime error because 2 cannot be matched with (y, z);
- x[0]`,` x[1] `=` x[1]`,` x[0]`;` swaps the first two elements of list x.

As in Python, pattern matching can also be used in `for` statements. For example:

```
for key, value in [ (1, 2), (3, 4) ]: ...
```

Harmony (but not Python) also allows pattern matching in defining and invoking methods. For example, you can write:

```
def f[a, (b, c)]: ...
```

and then call `f[1, (2, 3)]`. Note that the more familiar: `def` g(a) defines a method g with a single argument a. Invoking g(1, 2) would assign the tuple (1, 2) to a. This is not consistent with Python syntax. For single argument methods, you may want to declare as follows: `def` g(a,). Calling g(1,) assigns 1 to a, while calling g(1, 2) would result in a runtime error as (1, 2) cannot be matched with (a,).

Pattern matching can also be used in `const` and `let` statements.

H.5 For Loops and Comprehensions

While Harmony does not support general iterators such as Python does, Harmony allows iterating over sets and dictionaries (and thus lists and tuples). The details are a little different from Python:

- When iterating over a set, the set is always traversed in order (see Appendix A for how Harmony values are ordered);
- In case of a dictionary, the iteration is over the *values* of the dictionary, but in the order of the keys. In the case of lists, this works much the same as in Python, but in the case of general dictionaries, Python iterates over the keys rather than the values;
- If you want to iterate over the keys of a dictionary d, use `for` k `in keys` d;
- The `list` module (Section D.3) provides methods `values`(), `items`(), `enumerate`(), and `reversed`() for other types of iteration supported by Python.

```
import stack;

stack = Stack();
stack_push(?stack, 1);
stack_push(?stack, 2);
v = stack_pop(?stack);
assert v == 2;
stack_push(?stack, 3);
v = stack_pop(?stack);
assert v == 3;
v = stack_pop(?stack);
assert v == 1;
```

Figure H.1: [code/stacktest.hny] Testing a stack implementation.

Harmony supports nesting and filtering in `for loops`. For example:

```
for i in { 1..10 } for j in { 1..10 } such that i < j: ...
```

Harmony also supports set, list, and dictionary comprehensions. Set and list comprehensions are similar to Python, except that filtering uses the keywords `such that` instead of `if`. Dictionary comprehensions are a little different. The dictionary comprehension `dict`{ f(i) `for` i `in` I } creates a dictionary of i:f(i) entries.

H.6 Dynamic Allocation

Harmony supports various options for dynamic allocation. By way of example, consider a stack. Figure H.1 presents a test program for a stack. We present four different stack implementations to illustrate options for dynamic allocation:

Figure H.2 uses a single list to represent the stack. It is updated to perform `push` and `pop` operations;

Figure H.3 also uses a list but, instead of updating the list, it replaces the list with a new one for each operation;

Figure H.4 represents a stack as a recursively nested tuple (v, f), where v is the element on top of the stack and r is a stack that is the remainder;

Figure H.5 implements a stack as a linked list with nodes allocated using the `alloc` module.

While the last option is the most versatile (it allows cyclic data structures), Harmony does not support garbage collection for memory allocated this way and so allocated memory that is no longer in use must be explicitly released using `free`.

```
def Stack():
    result = [];
;
def stack_push(st, v):
    (!st)[len(!st)] = v;
;
def stack_pop(st):
    let n = len(!st) - 1:
        result = (!st)[n];
        del (!st)[n];
    ;
;
```

Figure H.2: [code/stack1.hny] Stack implemented using a dynamically updated list.

```
import list;

def Stack():
    result = [];
;
def stack_push(st, v):
    !st += [v,];
;
def stack_pop(st):
    let n = len(!st) - 1:
        result = (!st)[n];
        !st = subseq(!st, 0, n);
    ;
;
```

Figure H.3: [code/stack2.hny] Stack implemented using static lists.

```
def Stack():
    result = ();
;
def stack_push(st, v):
    (!st) = (v, !st);
;
def stack_pop(st):
    let (top, rest) = !st:
        result = top;
        !st = rest;
    ;
;
```

Figure H.4: [code/stack3.hny] Stack implemented using a recursive tuple data structure.

```
import alloc;

def Stack():
    result = None;
;
def stack_push(st, v):
    !st = malloc(dict{ .value: v, .rest: !st });
;
def stack_pop(st):
    let node = !st:
        result = node→value;
        !st = node→rest;
        free(node);
    ;
;
```

Figure H.5: [code/stack4.hny] Stack implemented using a linked list.

H.7 Comments

Harmony supports the same commenting conventions as Python. In addition, Harmony supports nested multi-line comments of the form `(* comment *)`.

Acknowledgments

I received considerable help and inspiration from various people while writing this book.

First and foremost I would like to think my student Haobin Ni with whom I've had numerous discussions about the design of Harmony. Haobin even contributed some code to the Harmony compiler.

Most of what I know about concurrent programming I learned from my colleague Fred Schneider. He suggested I write this book after demonstrating Harmony to him.

Leslie Lamport introduced me to using model checking to test properties of a concurrent system. My experimentation with using TLC on Peterson's Algorithm became an aha moment for me.

I first demonstrated Harmony to the students in my CS6480 class on systems and formal verification and received valuable feedback from them.

I would also like to thank my proofreaders Anneke van Renesse and Zhuoyu Xu.

Finally, I would like to thank my family who had to suffer as I obsessed over writing the code and the book during the turbulent months of May and June 2020.

Index

acknowledgment, 107
actor model, 96
address, 17, 35
alloc module, 125
alternating bit protocol, 107
atLabel operator, 22
atom, 14
atomic statement, 38
atomicity, 4

bag, 15
bag module, 125
barrier synchronization, 115
binary semaphore, 69
blocked process, 45
blocking queue, 96
bounded buffer, 68
broadcast, 85
busy waiting, 57
bytecode, 14

choose operator, 8
circular buffer, 68
client/server model, 68
comprehensions, 136
condition variable, 83
conditional critical section, 74
constant, 8
context, 15
continuation, 15
corner case, 5
counting semaphore, 69
critical region, 21
critical section, 21

data race, 43
deadlock, 89
deadlock avoidance, 93
determinism, 4
dictionary, 14
dining philosopher, 89
directory, 15
distributed system, 107
double turnstile, 117
dynamic allocation, 49

exception, 100

failure, 107
fairness, 78
formal verification, 5

go statement, 45

hand-over-hand locking, 112
Harmony method, 35
Harmony Virtual Machine, 14
Heisenbug, 4
HVM, 14

import statement, 37
inductive invariant, 30
interleaving, 10
interlock instruction, 38
interrupt, 100
interrupt-safety, 100
invariant, 5, 30

label, 22
linearizable, 112
linearization point, 112
list module, 125
liveness property, 22

lock, 23, 43, 47
lvalue, 123

machine instruction, 9
macro step, 131
Mesa, 83
message passing, 96
model checking, 4
module, 37, 47
monitor, 83
multiple conditions, waiting on, 92
multiset, 15
mutual exclusion, 22

name tag, 15
nametag operator, 22
network, 107
non-blocking synchronization, 110
non-determinism, 28
notify, 84
notifyAll, 85

P, 63
pattern matching, 135
Peterson's Algorithm, 28
pipeline, 68
pointer, 35
process, 9
process variable, 28
procure, 63
producer/consumer problem, 68
program counter, 15
progress, 22
property, 78
protocol, 107

race condition, 10
reachable state, 28
reader/writer lock, 54
register, 15
retransmission, 107
ring buffer, 68

safety property, 22
semaphore, 63
seqlock, 112
sequence number, 107
sequential, 4
set module, 126
shared variable, 4
signal, 83
spawn statement, 15
spin waiting, 57
spinlock, 38
split binary semaphore, 74
stack machine, 17
starvation, 47, 78
state, 28
state diagram, 23
step, 28
stop expression, 45
synch module, 43, 126
synchronized queue, 96

tag, 15
TAS, 38
test, 4
test-and-set, 38
thread, 4, 22
thread safety, 22
Time Of Check Time Of Execution, 41
TOCTOE, 41
trace, 28

unlock, 47

V, 63
vacate, 63
virtual machine, 14

wait, 83
wait-free synchronization, 112

Glossary

actor model is a concurrency model where there are no shared variables, only processes with private variables that communicate through message passing. 96

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. 4

barrier synchronization is when a set of processes execute in rounds, waiting for one another to complete each round. 115

blocked process is a process that cannot change the state or terminate or can only do so after another process changes the state first. For example, a process that is waiting for a lock to become available. 45

busy waiting (aka spin-waiting) is when a process waits in a loop for some application-defined condition instead of blocking. 57

concurrent execution (aka parallel execution) is when there are multiple processes executing and their machine instructions are interleaved in an unpredictable manner. 4

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. 83

conditional critical section is a critical section with, besides mutual exclusion, additional conditions on when a process is allowed to enter the critical section. 74

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. 15

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. 21

deadlock is when there are two or more processes waiting indefinitely for one another to release a resource. 89

determinism is when the outcome of an execution is uniquely determined by the initial state. 4

fairness is when each process eventually can access each resource it needs to access with high probability. 78

interlock instruction a machine instruction that involves multiple memory load and/or store operations, executed atomically. 38

invariant is a binary predicate over states that must hold for every reachable state of a process. 5

linearizable is a consistency condition for concurrent access to an object, requiring that each access must appear to execute atomically sometime between the invocation of the access and its completion. 112

lock an object that can be owned by at most one process at a time. Useful for implementing mutual exclusion. 23

machine instruction is an atomic operation on the Harmony virtual machine, executed by a process. 9

model checking is a formal verification method that explores all possible executions of a program, which must have a finite number of states. 4

monitor is a programming language paradigm that supports mutual exclusion as well as waiting for resources to become available. 83

mutual exclusion is the property that two processes never enter the same critical section. 22

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. 110

process is code in execution. We do not make the distinction between processes and threads. A process has a current context and updates its context every time it executes a machine instruction. 9

process variable is a variable that is private to a single process and stored in its register. 28

producer/consumer problem is a synchronization problem whereby one or more producing processes submit items and one or more consuming processes 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. 68

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. 78

race condition describes when multiple processes access shared state concurrently, leading to undesirable outcomes. 10

reader/writer lock is a lock on a resource that can be held by multiple processes if they all only read the resource. 54

semaphore is a counter that can be atomically incremented and decremented, but blocks the process until the counter is larger than zero first. 63

sequential execution is when there is just one process executing, as opposed to concurrent execution. 4

shared variable is a variable that is stored in the memory of the Harmony virtual machine and shared between multiple processes, as opposed to a process variable. 4

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. 38

stack machine is a model of computing where the state of a process is kept on a stack. Harmony uses a combination of a stack machine and a register-based machine. 17

starvation is when a process cannot make progress because it is continuously losing a competition with other processes to get access to a resource. 78

state an assignment of values to variables. In a Harmony virtual machine, this includes the contents of its shared memory and the set of contexts. 28

step is the execution of a machine instruction by a process, updating its state. Harmony distinguishes micro steps (machine instructions) and macro step (a sequence of instructions with at most one effect visible by other processes). 28

thread safety is when the implementation of a data structure allows concurrent access with well-defined semantics. 21

trace is a sequence of steps, starting from an initial state. An infinite trace is also called a *behavior*. 28