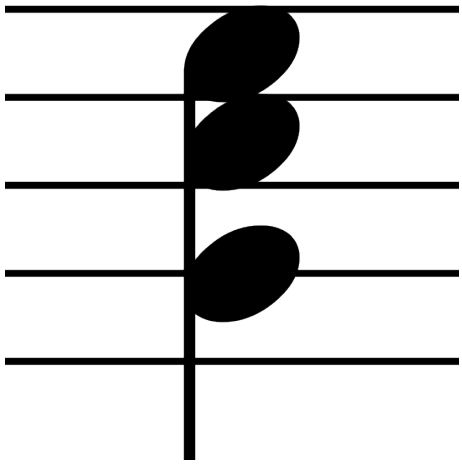
Concurrent Programming in Harmony



Robbert van Renesse

Contents

Chapter 1

On Concurrent Programming

Programming with concurrency is hard. On the one hand concurrency can make programs faster than sequential ones, but having multiple threads 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 threads are *interleaved*. Some bugs may occur only occasionally and may never occur when the program is instrumented to find them (so-called *Heisenbugs*—overhead caused by instrumentation leads to timing changes that makes such bugs less likely to occur).
- 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 threads 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 a particular test program. So, even if a bug is unlikely to occur, if the test *can* expose the bug it *will*. Moreover, 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.

If model checking does not prove a program correct, why is it useful? To answer that question, let us consider a sorting algorithm. 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 of 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. That said, 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 many correctness properties can be proved using *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], as well as Gerard Holzmann's work on Promela and SPIN [Hol11]. 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 threads 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 (although not very efficient or practical) 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 atomic *locks* for implemented critical sections.
- Chapter 9 presents Harmony modules that support locks and other synchronization primitives.
- Chapter 10 gives an introduction to building concurrent data structures.
- Chapter 11 discusses approaches to testing concurrent code in Harmony.
- Chapter 12 instead goes into how to find a bug in concurrent code using the Harmony output.
- Chapter 13 talks about threads having to wait for certain conditions. As examples, it presents the reader/writer lock problem and the bounded buffer problem.
- Chapter 14 presents *Split Binary Semaphores*, a general technique for solving synchronization problems.
- Chapter 15 talks about *starvation*: the problem that in some synchronization approaches threads may not be able to get access to a resource they need.
- Chapter 16 presents *monitors* and *condition variables*, another approach to thread synchronication.
- Chapter 17 describes *deadlock* where a set of threads are indefinitely waiting for one another to release a resource.
- Chapter 18 presents the *actor model* and *message passing* as an approach to synchronization.
- Chapter 19 describes *barrier synchronization*, useful in high-performance computing applications such as parallel simulations.
- Chapter 20 discusses how to handle interrupts, a problem closely related to—but not the same as—synchronizing threads.
- Chapter 21 introduces *non-blocking* or *wait-free* synchronization algorithms, which prevent threads waiting for one another more than a bounded number of steps.
- Chapter 22 presents a problem and a solution to the distributed systems problem of having two threads communicate reliably over an unreliable network.

Chapter 2

Introduction to Harmony

Harmony is a programming language that borrows much of Python's syntax. Like Python, Harmony is an imperative, dynamically typed, and garbage collected programming language. There are also some important differences:

- 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. On the other hand, Harmony supports pointers to objects and methods.

There are also less important 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

```
1  const N = 10
2
3  def triangle(n):   # computes the n'th triangle number
4      result = 0
5      for i in {1..n}:      # for each integer from 1 to n inclusive
6          result += i       # add i to result
7
8  x = choose({0..N})       # select an x between 0 and N inclusive
9  assert triangle(x) == ((x * (x + 1)) / 2)
```

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

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). (Harmony does not have iterators and in particular does not have a `range` operator.) 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):

```
$ harmony triangle.hny
#states 13
13 components, 0 bad states
No issues
$
```

(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
103 components, 0 bad states
No issues
$
```

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

```
1   shared = True
2
3   def f(): assert shared
4   def g(): shared = False
5
6   f()
7   g()
```

(a) [code/prog1.hny] Sequential

```
1   shared = True
2
3   def f(): assert shared
4   def g(): shared = False
5
6   spawn f()
7   spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

Concurrent programming, aka multithreaded programming, involves multiple threads running in parallel while sharing variables. Figure 3.1 shows two programs. Program (a) is sequential. It sets *shared* to `True`, asserts that *shared* = `True`, and finally sets *shared* to `False`. If you run the program through Harmony, it will not find any problems because there is only one execution possible and 1) in that execution the assertion does not fail and 2) the execution terminates. Program (b) is concurrent. It executes methods `f()` and `g()` in parallel. If method `g()` runs and completes before `f()`, then the assertion in `f()` will fail when `f()` runs. Harmony will find this problem. This problem is an example of non-determinism: methods `f()` and `g()` can run in either order.

Figure 3.2 presents a more subtle example that illustrates non-atomicity. The program initializes two shared variables: an integer *count* and an array *done* with two booleans. The program then spawns two threads. The first runs `incrementer(0)`; the second runs `incrementer(1)`.

Method `incrementer` takes a parameter called *self*. It increments *count* and sets *done*[*self*] to `True`. It then waits until the other thread is done. (**await** *c* is shorthand for **while not** *c*:

pass) and semantically the same. After that, method `incrementer` verifies that the value of *count* equals 2.

Note that although the threads 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 that particular execution as well. What would the value of *count* be at the end of that execution?

```
1   count = 0
2   done = [ False, False ]
3
4   def incrementer(self):
5       count = count + 1
6       done[self] = True
7       await done[1 - self]
8       assert count == 2
9
10  spawn incrementer(0)
11  spawn incrementer(1)
```

Figure 3.2: [code/Up.hny] Incrementing twice in parallel.

- 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 shared memory locations. So, to increment a value in memory, the machine must do at least three machine instructions. Conceptually:

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

When running multiple threads, 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 unpredictable ways. A program is correct if it works for any interleaving of threads. Harmony will try all possible interleavings of the threads executing machine instructions.

If the threads run one at a time, then *count* will be incremented twice and ends up being 2. However, the following is also a possible interleaving of `incrementer(0)` and `incrementer(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 1 to the value that it loaded (0), and stores 1 into *count*;
4. `incrementer(0)` adds 1 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, not the specification.

The exercises below have you try the same thing in Python. As you will see, the bug is not easily triggered when you run a Python version of the program. But in Harmony Murphy's Law applies: if something can go wrong, it will. Usually that is not a good thing, but in Harmony it is. It allows you to find bugs in your concurrent programs much more easily than with a conventional programming language.

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 3.3 is a Python version of Figure 3.2.

1. Run Figure 3.3 using Python. Does the assertion fail?
2. Using a script, run Figure 3.3 1000 times. For example, if you are using the bash shell (in Linux or Mac OS X, say), you can do the following:

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

```
import threading

count = 0
done = [ False, False ]

def incrementer(self):
    global count
    count = count + 1
    done[self] = True
    while not done[1 - self]:
        pass
    assert count == 2

threading.Thread(target=incrementer, args=(0,)).start()
threading.Thread(target=incrementer, args=(1,)).start()
```

Figure 3.3: [python/Up.py] Python implementation of Figure 3.2.

If you're using Windows, the following batch script does the trick:

```
FOR /L %%i IN (1, 1, 1000) DO python Up.py
PAUSE
```

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

3.2 Figure 3.4 is a version of Figure 3.3 that has each incrementer thread increment *count* `N` times. Run Figure 3.4 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? (Try powers of 10 for `N`.)

3.3 Can you think of a fix to Figure 3.2? 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. (Also, you do not get to use a *lock*, yet.)

```
import threading

N = 1000000
count = 0
done = [ False, False ]

def incrementer(self):
    global count
    for i in range(N):
        count = count + 1
    done[self] = True
    while not done[1 - self]:
        pass
    assert count == 2*N, count

threading.Thread(target=incrementer, args=(0,)).start()
threading.Thread(target=incrementer, args=(1,)).start()
```

Figure 3.4: [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 machine instructions, and in our case those 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 keys to values. Unlike Python, any Harmony value can be a key, including another dictionary. Dictionaries are written as $\{k_0 : v_0,\ k_1 : v_1,\ ...\}$. 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
```

The meaning of $d\ a\ b$ ... is $(((d\ a)\ b)$ ...). This notation is unfamiliar to Python programmers, 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 $d = \{$`.count` : 3$\}$, then you 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.

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 { `0:5, 1:False` }. The empty

tuple is written as `()`. As in Python, you can create singleton tuples by including a comma. For example, `(1,)` is a tuple consisting just of the number 1. Importantly, $(1) = 1 \neq (1,)$.

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 { `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 A provides details on all the values that Harmony currently supports.

Harmony Bytecode

A Harmony program is translated into HVM bytecode. To make it amenable to efficient model checking, 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 a Harmony value.
- Threads: Any thread can spawn an unbounded number of other threads and threads may terminate. Each thread has a program counter that indexes into the code, a stack of Harmony values, and two private *registers* that each contain a Harmony value.

One of the registers of a thread contains the local variables of the method that the thread is currently executing. It is saved and restored by method invocations. The other register is called **this** and is used for so-called *thread-local state*, which contains variables that can be used by all methods. The state of a thread is called a *context* (aka *continuation*): it contains the values of its program counter, stack, and registers. 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 threads can have the same context.

It may seem strange that there is only one memory location. However, this is not a limitation because Harmony values are unbounded trees. The shared memory is a dictionary that maps atoms (names of shared variables) to other Harmony values. We call this a *directory*. Thus, a directory represents the state of a collection of variables named 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 Harmony values, like a path name in the file system hierarchy. We call such a sequence an *address*. For example, in Figure 3.2 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. So, for example, the address of `done[0]` is the sequence [`.done`, 0]. An address is itself a Harmony value.

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

Figure 4.1: The first part of the HVM bytecode corresponding to Figure 3.2.

```
#states 32
Safety Violation
T0: __init__() [0-5,32-40] { count: 0, done: [ False, False ] }
T1: incrementer(0) [ 6- 9] { count: 0, done: [ False, False ] }
T2: incrementer(1) [ 6-18] { count: 1, done: [ False, True  ] }
T1: incrementer(0) [10-28] { count: 1, done: [ True,  True  ] }
Harmony assertion failed: 1
open file:///Users/rvr/github/harmony/harmony.html for more information
```

Figure 4.2: The text output of running Harmony on Figure 3.2.

Compiling the code in Figure 3.2 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 thread in the HVM is predominantly a *stack machine*, but it also has two registers. Like shared memory, the registers usually contain dictionaries so they can represent the values of multiple named variables. As mentioned before, one register contains a dictionary with the local variables of the method that the thread is currently executing, while the other contains thread-local state.

All instructions are atomically executed. The Harmony memory model is *sequentially consistent*: all accesses are in program order. Most instructions pop values from the stack or push values onto the stack. At first there is one thread, named `__init__`, which initializes the state. It starts executing at instruction 0 and keeps executing until it reaches the last instruction in the program. In this case, it executes instructions 0 through 5 first. The last instruction in that sequence is a `JUMP` instruction that sets the program counter to 32 (skipping over the code for `incrementer` method). The `__init__` thread then executes instructions 32 through 40 and finishes. Once initialization completes, any threads that were spawned (in this case `incrementer(0)` and `incrementer(1)`) can run.

At program counter 6 is the code for the `incrementer` method. All methods start with a `Frame` instruction and end with a `Return` instruction. Appendix D 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 5 of `Up.hny` is as follows (see Figure 4.1):

7. The `Load` instruction pushes the value of the *count* variable onto the stack.
8. The `Push` instruction pushes the constant 1 onto the stack of the thread.
9. `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.
10. The `Store` instruction pops a Harmony value (the sum of the *count* variable and 1) and stores it in the *count* variable.

You can think of Harmony as trying every possible interleaving of threads executing instructions. Figure 4.2 shows the output produced by running Harmony on the `Up.hny` program.

Harmony can report the following 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. Harmony will print a trace of the shortest bad execution that it found.
- `Non-terminating State`: Harmony found one or more states from which there does not exist an execution such that all threads terminate. Harmony will not only print the non-terminating state with the shortest trace, but also the list of threads at that state, along with their program counters.
- `Active Busy Waiting`: There are states in which some thread cannot make progress without the help of another thread but does not block;
- `Data Race`: There are states in which two or more threads concurrently access a shared variable, at least one of which is a store operation.

Harmony checks for these types of failure conditions in the given order: if there are multiple failure conditions, only the first is reported. *Active busy waiting* (Chapter 13) is not technically an indication of a synchronization problem, but instead an indication of an inefficient solution to a synchronization problem— one that uses up significantly CPU cycles. A *data race* (Chapter 9) may not be a bug either—whether or not it is might depend on the semantics of the underlying memory operations and are therefore generally undesirable.

Going back to our example, Harmony reports a safety violation. In particular, it reports that the assertion failed because *count* has value 1 instead of 2. Next it reports an execution that failed this assertion. The program got to the failed assertion in 4 "turns." The output has four columns:

1. A thread identifier;
2. The main method and argument of the thread;
3. The sequence of program counters of the HVM instructions that the thread executed;
4. The contents of the shared memory.

The four turns in the execution are as follows:

1. Thread `__init__` (with identifier T0) sets shared variable *count* to 0 and shared variable *done* to `[False, False]`.
2. Thread `incrementer(0)` (with identifier T1) executes instructions 6 through 9, loading the value of *count* but stopping just before storing 1 into *count*;
3. Thread `incrementer(1)` (with identifier T2) executes instructions 6 through 18, storing 1 into *count* and storing `True` into *done*[1];
4. Thread `incrementer(0)` continues execution, executing instructions 10 through 28 storing value 1 into *count* (instruction 10), storing `True` into *done*[0] (instructions 11 through 15) finding that *done*[1] is `True` (instructions 16 through 21), and finally detecting that the assertion is violated (instructions 22 through 28).

Issue: Safety violation

Turn	Thread	Instructions Executed	PC	Shared Variables: *count*	*done* 0	*done* 1
1	T0: __init__()		39	0	False	False
2	T1: incrementer(0)		10	0	False	False
3	T2: incrementer(1)		19	1	False	True
4	T1: incrementer(0)		27	1	True	True

code/Up.hny:8 assert count == 2

24	Load count
25	Push 2
26	2-ary ==
27	Assert
28	AtomicDec
29	ReadonlyDec
30	Return

Threads

ID	Status	Stack Trace		Stack Top
T0	terminated	__init__()		
T1	failed atomic read-only	incrementer(0)	self: 0	
		Harmony assertion failed		
T2	runnable	incrementer(1)	self: 1	0

Figure 4.3: The HTML output of running Harmony on Figure 3.2. There are four sections. The top shows the turns to the problematic state. Each turn shows a thread identifier, the method the thread is executing, a "timeline" containing a block for each instruction executed, the program counter of the thread, and the values of the shared variables at those turns. The middle section shows the line of code that is being executed. The bottom left shows the bytecode of the program. The bottom right shows the state of each thread. Each row in the stack trace of a thread shows the method that is being evaluated and the values of its method variables.

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.

In the top right, the HTML file contains the reported issue in red. Underneath it, a table shows the four turns in the execution. Instead of listing explicitly the program counters of the executed instructions, the HTML file contains a list of blocks for each executed instruction. We call this the *timeline*. You can click on such a block to see the state of the Harmony virtual machine just after executing the corresponding instruction. The turn that is being executed is highlighted in green. The table also lists the program counter of the thread at each turn, and the values of the shared variables.

Underneath the table it shows the line of Harmony code that is being executed in blue.

The bottom left shows the bytecode of the program being executed. It has alternating grey and white sections. Each section corresponds to a line of Harmony code. The instruction that is about to be executed, if any, is highlighted in red. (In this case, the state shown is a failed state and no instruction will be executed next.) If you hover the mouse over a machine instruction, it provides a brief explanation of what the instruction does.

The bottom right contains a table with the state of each thread. The thread that is executing is highlighted in green. Status information for a thread can include:

`runnable`: the thread is runnable but not currently running. In Harmony, threads are interleaved and so at most one thread is actually running;

`running`: the thread is currently executing instructions;

`terminated`: the thread has completed all its instructions;

`failed`: the thread has encountered an error, such as violating an assertion or divide by zero;

`blocked`: the thread cannot make progress until another thread has updated the shared state. For example, this occurs when one of the implementers is waiting for the other to set its *done* flag;

`atomic`: the thread is in *atomic* mode, not allowing other threads to be scheduled. This is, for example, the case when an assertion is being checked;

`read-only`: the thread is in *read-only* mode, not able to modify shared state. Assertions can execute arbitrary code including methods, but they are not allowed to modify the shared state.

The stack of each thread is subdivided into two parts: the *stack trace* and the *stack top*. A stack trace is a list of methods that are being invoked. In this case, the `incrementer` method does not invoke any other methods, and so the list is of length 1. For each entry in the stack trace, it shows the method name and arguments, as well as the variables of the method. The stack top shows the values on the stack beyond the stack trace.

When you load the HTML file, it shows the state after executing the last instruction. As mentioned above, you can go to any point in the execution by clicking on

```
1   count = 0
2   entered = done = [ False, False ]
3
4   def incrementer(self):
5       entered[self] = True
6       if entered[1 - self]:        # if the other thread has already started
7           await done[1 - self]     # wait until it is done
8       count = count + 1
9       done[self] = True
10      await done[1 - self]
11      assert count == 2
12
13  spawn incrementer(0)
14  spawn incrementer(1)
```

Figure 4.4: [code/UpEnter.hny] Broken attempt at fixing the code of Figure 3.2.

one of the blocks in the timeline. There are also various handy keyboard shortcuts:

Right arrow:	go to the next instruction;
Left arrow:	go to the previous instruction;
Down arrow:	go to the next turn;
Up arrow:	go to the previous turn;
Enter (aka Return):	go to the next line of Harmony code;
0:	go to the initial state.

If you want to see an animation of the entire execution, one instruction at a time, you can first hit 0 and then hold down the right arrow. If you want to see it one line of Harmony code at a time, hold down the enter (aka return) key instead. If you hold down the down arrow key, the movie will go by very quickly.

Exercises

4.1 Figure 4.4 shows an attempt at trying to fix the code of Figure 3.2. 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 5 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.

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 thread is trying to do the same thing. This is called a *critical section* (aka critical region) [Dij65]: a set of instructions where only one thread 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 if during this time no other thread 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 not a problem as critical sections guarantee that no other thread will be able to see it. An implementation of a data structure that can be safely accessed by multiple threads and free of race conditions is called *thread-safe*.

```
1       def thread(self):
2           while True:
3               #enter critical section
4               @cs: assert atLabel(cs) == { (thread, self): 1 }
5               #exit critical section
6
7       spawn thread(0)
8       spawn thread(1)
```

Figure 5.1: [code/csbarebones.hny] A bare bones critical section with two threads.

```
1    def thread(self):
2        while choose({ False, True }):
3            #enter critical section
4            @cs: assert atLabel(cs) == { (thread, self): 1 }
5            #exit critical section
6
7    spawn thread(0)
8    spawn thread(1)
```

Figure 5.2: [code/cs.hny] Harmony model of a critical section.

A critical section is often modeled as threads 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 threads 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 thread 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 (method, argument) pairs of threads executing at that label. In our case there are two threads: `(thread, 0)` and `(thread, 1)`. A 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. 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 threads from being at label `@cs` at the same time. But it does so by preventing *any* thread 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 threads 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 threads that are trying to enter the critical section and (2) threads in the critical section always leave eventually, then eventually one thread in S will enter the critical section. We call this *progress*.

In order to detect violations of progress, and other liveness problems in algorithms in general, Harmony requires that every execution must be able to reach a state in which all threads have terminated. Clearly, even if mutual exclusion holds in Figure 5.1, the spawned threads never terminate. We will instead model threads in critical sections using the framework in Figure 5.2: a thread can *choose* to enter a critical section more than once, but it can also choose to terminate, even

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

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

without entering the critical section ever. (Recall that Harmony will try every possible execution, and so it will evaluate both choices.)

A thread 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 thread can own (or hold) at a time. A thread 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 5.3 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 thread waits until *lockTaken* is `False` and then sets it to `True` to indicate that the lock has been taken. The thread then executes the critical section. Finally the thread 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 5.4 shows the Harmony output. Thread 0 finds that the lock is available, but just before it stores `True` in *lockTaken* in instruction 9, thread 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 the critical section. Finally, thread 0 moves on, also stores `True` into *lockTaken* and also moves into the critical section. Thread 0 is the one that detects the problem. The *lockTaken* variable suffers from the same sort of race condition as the *count* variable in Figure 3.2: testing and setting the lock consists

Issue: Safety violation

Turn	Thread	Instructions Executed	PC	Shared Variables *lockTaken*
1	T0: __init__()		44	False
2	T1: thread(0)		11	False
3	T2: thread(1)		12	True
4	T1: thread(0)		28	True

code/naiveLock.hny:10 @cs: assert atLabel(cs) == { (thread, self): 1 }

24	3-ary DictAdd
25	Push 1
26	3-ary DictAdd
27	2-ary ==
28	Assert
29	AtomicDec
30	ReadonlyDec

Threads

ID	Status	Stack Trace		Stack Top
T0	terminated	__init__()		
T1	failed atomic read-only	thread(0)	self: 0	
		Harmony assertion failed		
T2	runnable	thread(1)	self: 1	

Figure 5.4: The HTML output of running Harmony on Figure 5.3.

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

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Issue: Non-terminating state

Turn	Thread	Instructions Executed	PC	Shared Variables: *flags* 0	*flags* 1
1	T0: __init__()		54	False	False
2	T1: thread(0)		16	True	False
3	T2: thread(1)		16	True	True

code/naiveFlags.hny:7 await not flags[1 - self]

12	Store
13	Push 1
14	LoadVar self
15	2-ary -
16	Load flags
17	Apply
18	JumpCond True 13

Threads

ID	Status	Stack Trace		Stack Top
T0	terminated	__init__()		
T1	blocked	thread(0)	self: 0	1
T2	blocked	thread(1)	self: 1	0

Figure 5.6: The HTML output of running Harmony on Figure 5.5.

```
1   turn = 0
2
3   def thread(self):
4       while choose({ False, True }):
5           # Enter critical section
6           turn = 1 - self
7           await turn == self
8
9           # Critical section
10          @cs: assert atLabel(cs) == { (thread, self): 1 }
11
12          # Leave critical section
13
14  spawn thread(0)
15  spawn thread(1)
```

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

of several instructions. It is thus possible for both threads to believe the lock is available and to obtain the lock at the same time.

Preventing multiple threads from updating the same variable, Figure 5.5 presents a solution based on each thread having a flag indicating that it is trying to enter the critical section. A thread can write its own flag and read the flag of its peer. After setting its flag, the thread waits until the other thread ($1 - self$) 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 threads being in the critical section at the same time.

To see why, first note the following invariant: if thread i is in the critical section, then *flags*[i] == `True`. Without loss of generality, suppose that thread 0 sets *flags*[0] at time t_0. Thread 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 thread 1 is not in the critical section at time t_1. Let t_2 be the time at which thread 0 sets *flags*[0] to `False`. Thread 0 is in the critical section sometime between t_1 and t_2. It is easy to see that thread 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 5.6), it turns out the solution does have a problem: if both try to enter the critical section at the same time, they may end up waiting for one another indefinitely. (This is a form of *deadlock*, which will be discussed in Chapter 17.) Thus the solution violates *progress*.

The final naïve solution that we propose is based on a variable called *turn*. Each thread politely lets the other thread have a turn first. When $turn = i$, thread i can enter the critical section, while thread $1 - i$ has to wait. An invariant of this solution is that while thread 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 the thread that has been waiting the longest to enter the critical section can go next.

Run the program through Harmony. It turns out that this solution also violates *progress*, albeit for a different reason: if thread i terminates instead of entering the critical section, thread $1 - i$, politely, ends up waiting indefinitely for its turn. Too bad, because it would have been a great solution if both threads try to enter the critical section ad infinitum.

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. If you get *active busy waiting* or *data race* reports, you probably found a correct solution; you'll learn later how to suppress those. Do not despair if you can't figure out how to develop a solution that satisfies both mutual exclusion and progress—as we will find out, it is possible but not obvious.

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 5.5 and Figure 5.7, and is presented in Figure 6.1. (The first line specifies that the *flags* and *turn* variables should have *sequential consistency*—it prevents Harmony from complaining about data races involving these variables, explained in Chapter 8.)

A thread first indicates its interest in entering the critical section by setting its flag. It then politely gives way to the other thread 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 thread continues to be polite, waiting until either the other thread is nowhere near the critical section ($flag[1 - self]$ == `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. It turns out that doing so is not easy. If you are interested in learning more about concurrent programming but not necessarily in how to prove concurrent programs correct, you may choose to skip the rest of this chapter. If you are still here, you 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 thread variables of each thread, including its program counter and the contents of its stack. Each time a thread executes a HVM machine instruction, the state changes (if only because the program counter of the thread changes). We call that a *step*. Steps in Harmony are atomic.

The HVM starts in an initial state in which there is only one thread 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 thread that can make a step. The other is when a thread 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 finite trace. A state is final when there are no threads left to make state changes.

```
1   sequential flags, turn
2
3   flags = [ False, False ]
4   turn = choose({0, 1})
5
6   def thread(self):
7     while choose({ False, True }):
8       # Enter critical section
9       flags[self] = True
10      turn = 1 - self
11      await (not flags[1 - self]) or (turn == self)
12
13      # critical section is here
14      @cs: assert atLabel(cs) == { (thread, self): 1 }
15
16      # Leave critical section
17      flags[self] = False
18
19  spawn thread(0)
20  spawn thread(1)
```

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

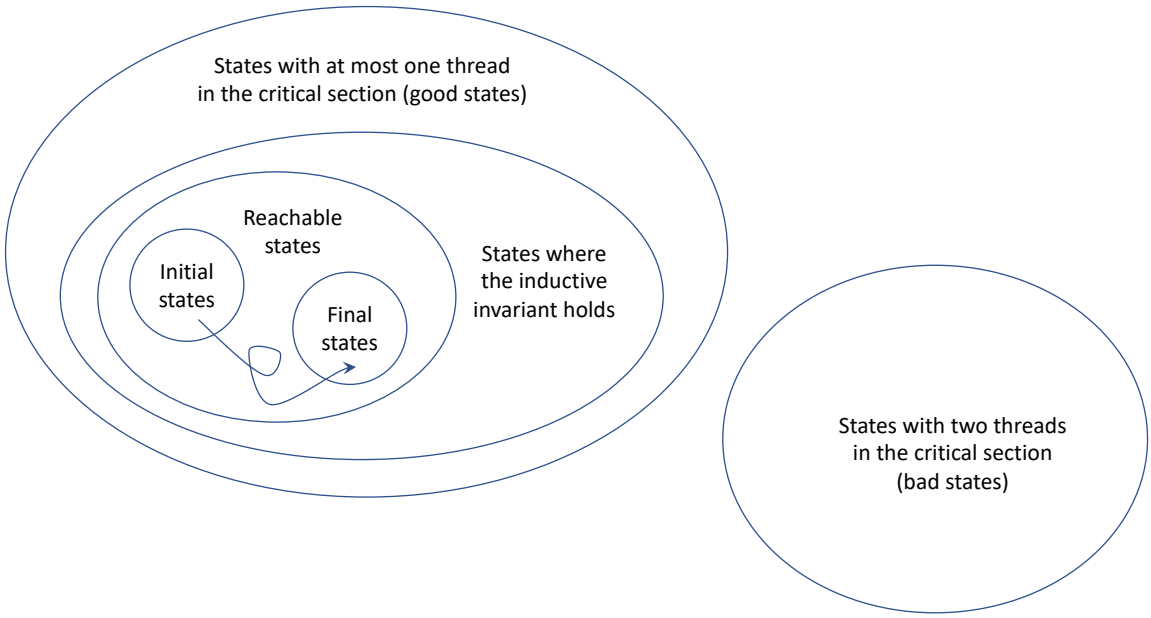


Figure 6.2: Venn diagram classifying all states and a trace.

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 threads executing, are examples of such predicates. For our purposes, it is useful to define a predicate that says that at most one thread 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 because mutual exclusion means that all reachable states are exclusive. 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 thread 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 define the set of reachable states inductively, 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 (including unreachable ones) and for any thread in the state, if the thread takes a step, then $\mathcal{I}$ also holds in the resulting state.

One candidate for such a predicate 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, exclusivity is not an *inductive* invariant. To see why, consider the following state s: let thread 0 be at label `@cs` and thread 1 be at the start of the **await** statement. Also, in state s, $turn = 1$. Now let thread 1 make a step. Because $turn = 1$, thread 1 will stop waiting and also enter the critical section, entering a state that is not exclusive. So, exclusivity is an invariant (holds over every reachable state, as demonstrated by Harmony), but not an inductive invariant. It will turn out that s is not reachable.

We are looking for an inductive invariant that *implies* exclusivity. In other words, the set of states where the inductive invariant holds must be a subset of the set of states where there is at most one thread in the critical section.

Let us begin with considering the following important property: $\mathcal{F}(i) = \texttt{thread}(i)@[8 \cdots 15] \Rightarrow flags[i]$, that is, if thread i is executing in lines 8 through 15, then $flags[i]$ is set. Although it does not, by itself, imply exclusivity, we can show that $\mathcal{F}(i)$ is an inductive invariant (for both threads 0 and 1). To wit, it holds in the initial state, because in the initial state thread i does not even exist yet. Now we have to show that if $\mathcal{F}(i)$ holds in some state, then $\mathcal{F}(i)$ also holds in a next state. Since only thread i ever changes $flags[i]$, we only need to consider steps by thread i. Since $\mathcal{F}(i)$ holds, there are two cases to consider:

1. states in which $flags[i] = \texttt{true}$
2. states in which $\neg\texttt{thread}(i)@[8 \cdots 15]$ (because false implies anything)

In each case we need to show that if thread i takes a step, $\mathcal{F}(i)$ still holds. In the first case, there is only one step that thread i can take that would set $flags[i]$ to false: the step in line 15. But executing the line would also take the thread out of lines $8 \cdots 15$, so $\mathcal{F}(i)$ continues to hold. In the second case (thread i is not executing in lines $8 \cdots 15$), the only step that would cause thread i to execute in lines $8 \cdots 15$ would be the step in line 7. But in that case $flags[i]$ would end up being true, so $\mathcal{F}(i)$ continues to hold as well. So, $\mathcal{F}(i)$ is an inductive invariant (for both threads 0 and 1).

While $\mathcal{F}(i)$ does not imply mutual exclusion, it does imply the following useful invariant: $\texttt{thread}(i)@cs \Rightarrow flags[i]$: when thread i is at the critical section, $flags[i]$ is set. This seems obvious from the code, but now you know how to prove it.

We need a stronger inductive invariant than $\mathcal{F}(i)$ to prove mutual exclusion. What else do we know when thread i is in the critical section? Let $\mathcal{C}(i) = \neg flags[1 - i] \vee turn = i$, that is, the condition on the **await** statement for thread i. In a sequential program, $\mathcal{C}(i)$ would clearly hold if thread i is in the critical section: $\texttt{thread}(i)@cs \Rightarrow \mathcal{C}(i)$. However, because thread $1 - i$ is executing concurrently, this property does not hold. In particular, suppose thread 0 is at the critical section, $flags[0] = \text{true}$, $turn = 1$, and thread 1 just finished the step in line 7, setting $flags[1]$ to true. Notice that $C(0)$ is now violated.

Instead, we will use the following property: $\mathcal{G}(i) = \texttt{thread}(i)@\texttt{cs} \Rightarrow \mathcal{C}(i) \vee \texttt{thread}(1-i)@\texttt{gate}$. That is, if thread i is at the critical section, then either $\mathcal{C}(i)$ holds or thread $1 - i$ is about to execute line 8. Figure 6.3 formalizes $\mathcal{G}(i)$ in Harmony. The label `@gate` refers to the step that sets $turn$ to $1 - i$. You can run Figure 6.3 to determine that $\mathcal{G}(i)$ is an invariant for $i = 0, 1$. Moreover, if $\mathcal{F}(i)$ and $\mathcal{G}(i)$ both hold for $i = 0, 1$, then mutual exclusion holds. We can show this using proof by contradiction. Suppose mutual exclusion is violated and thus both threads are in

```
1   sequential flags, turn
2
3   flags = [ False, False ]
4   turn = choose({0, 1})
5
6   def thread(self):
7       while choose({ False, True }):
8           # Enter critical section
9           flags[self] = True
10          @gate: turn = 1 - self
11          await (not flags[1 - self]) or (turn == self)
12
13          # Critical section
14          @cs: assert (not flags[1 - self]) or (turn == self)
15                  or (atLabel(gate) == {(thread, 1 - self): 1})
16
17          # Leave critical section
18          flags[self] = False
19
20  spawn thread(0)
21  spawn thread(1)
```

Figure 6.3: [code/PetersonInductive.hny] Peterson's Algorithm with Inductive Invariant

the critical section. By $\mathcal{F}$ it must be the case that both *flags* are true. By $\mathcal{G}$ and the fact that neither thread is at label `@gate`, we know that both $C(0)$ and $C(1)$ must hold. This then implies that $turn = 0 \wedge turn = 1$, providing the desired contradiction.

We claim that $\mathcal{G}(i)$ is an inductive invariant. First, since neither thread in in the critical section in the initial state, it is clear that $\mathcal{G}(i)$ holds in the initial state. Without loss of generality, suppose $i = 0$ (a benefit from the fact that the algorithm is symmetric for both threads). We still have to show that if we are in a state in which $\mathcal{G}(0)$ holds, then any step will result in a state in which $\mathcal{G}(0)$ still holds.

First consider the case that thread 0 is at label `@cs`. If thread 0 were to take a step, then in the next state thread 0 would be no longer at that label and $\mathcal{G}(0)$ would hold trivially over the next state. Therefore we only need to consider a step by thread 1. From $\mathcal{G}$ we know that one of the following three cases must hold before thread 1 takes a step:

1. *flags*[1] = `False`;
2. *turn* = 0;
3. thread 1 is at label `@gate`.

Let us consider each of these cases. We have to show that if thread 1 takes a step, then one of those cases must hold after the step. In the first case, if thread 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 thread 1 will be at label `@gate` (in which case the third case will hold). We know that thread 1 never sets *turn* to 1, so if the second case holds before the step, it will also hold after the step. Finally, if thread 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.

Now consider the case where thread 0 is not in the critical section, and therefore $\mathcal{G}(0)$ holds trivially because false implies anything. There are three cases to consider:

1. Thread 1 takes a step. But then thread 0 is still not in the critical section and $\mathcal{G}(0)$ continues to hold;
2. Thread 0 takes a step but still is not in the critical section. Then again $\mathcal{G}(0)$ continues to hold.
3. Thread 0 takes a step and ends up in the critical section. Because thread 0 entered the critical section, we know that *flags*[1] = `False` or *turn* == 0 because of the `await` condition. And hence $\mathcal{G}(0)$ continues to hold in that case as well.

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 inductive invariants 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.

Even though they are not strictly necessary, we encourage you to include invariants in your Harmony code. They can provide important insights into why the code works.

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 thread i is in the critical section, then $\mathcal{C}(i)$ (i.e., $\mathcal{G}$ without the disjunct that thread $1 - i$ 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 solution to the mutual exclusion problem. It is similar to the one in Figure 5.5, but has a thread *back out and try again* if it finds that the other thread is either trying to enter the critical section or already has. Compare this algorithm with Peterson's. Why does Harmony complain about *active busy waiting*?

6.2 Can you find one or more inductive invariants 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

thread X:                          thread 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 threads?

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 threads so they only try to enter the critical section a bounded number of times.

```
1   sequential flags
2
3   flags = [ False, False ]
4
5   def thread(self):
6       while choose({ False, True }):
7           # Enter critical section
8           flags[self] = True
9           while flags[1 - self]:
10              flags[self] = False
11              flags[self] = True
12
13          # Critical section
14          @cs: assert atLabel(cs) == { (thread, self): 1 }
15
16          # Leave critical section
17          flags[self] = False
18
19  spawn thread(0)
20  spawn thread(1)
```

Figure 6.4: [code/csonebit.hny] Mutual exclusion using a flag per thread.

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 too 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*, itself a Harmony value. 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

```
1   def P_enter(pm, pid):
2       pm→flags[pid] = True
3       pm→turn = 1 - pid
4       await (not pm→flags[1 - pid]) or (pm→turn == pid)
5
6   def P_exit(pm, pid):
7       pm→flags[pid] = False
8
9   def P_mutex():
10      result = { .turn: choose({0, 1}), .flags: [ False, False ] }
11
12  #### The code above can go into its own Harmony module ####
13
14  sequential mutex
15  mutex = P_mutex()
16
17  def thread(self):
18      while choose({ False, True }):
19          P_enter(?mutex, self)
20          @cs: assert atLabel(cs) == { (thread, self): 1 }
21          P_exit(?mutex, self)
22
23  spawn thread(0)
24  spawn thread(1)
```

Figure 7.1: [code/PetersonMethod.hny] Peterson's Algorithm accessed through methods.

```
const FIFO = False

def CLOCK(n):
    result = { .entries: [None,] * n, .recent: {}, .hand: 0, .misses: 0 }

def ref(clock, x):
    if x not in clock→entries:
        while clock→entries[clock→hand] in clock→recent:
            clock→recent -= {clock→entries[clock→hand]}
            clock→hand = (clock→hand + 1) % len(clock→entries)
        clock→entries[clock→hand] = x
        clock→hand = (clock→hand + 1) % len(clock→entries)
        clock→misses += 1
    if not FIFO:
        clock→recent |= {x}

clock3, clock4, refs = CLOCK(3), CLOCK(4), [ ]

const VALUES = { 1..5 }

let last = {}:
    for i in {1..100}:
        let x = i if i < 5 else choose(VALUES - last):
            refs = refs + [x,]
            ref(?clock3, x); ref(?clock4, x)
            assert(clock4.misses <= clock3.misses)
            last = {x}
```

Figure 7.2: [code/clock.hny] Harmony program that finds page replacement anomalies.

that contains Peterson's Algorithm's shared memory state: a turn variable and two flags. Both methods `P_enter` and `P_exit` take two arguments: a pointer to a mutex and the thread identifier (0 or 1). *pm*→`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 usable by multiple applications.

Finally, methods can have local variables. Different from Python, Harmony local variables must be explicitly bound within a scope. You have already seen examples: the arguments to the method, the variable *result*, and the bound variable (or variables) within a `for` statement. The `let` statement allows declaring additional local variables. For example: `let` $x = 3$: y += x adds 3 to the global variable y. See Section G.2 for more information.

If you are ready to learn about how locks are implemented in practice, you can now skip the rest of this chapter. But if you would like to see a cool example of using the concepts introduced in this chapter, hang on for a sequential Harmony program that finds anomalies in page replacement

algorithms. In 1969, Béláady et al. published a paper [BNS69] that showed that making a cache larger does not necessarily lead to a higher hit ratio. He showed this for a scenario using a FIFO replacement policy when the cache is full. The program in Figure 7.2 will find exactly the same scenario if you define `FIFO` to be **True**. Moreover, if you define `FIFO` to be **False**, it will find a scenario for the CLOCK replacement policy [Cor69], often used in modern operating systems.

In this program, `CLOCK` maintains the state of a cache (in practice, typically pages in memory). The set `recent` maintains whether an access to the cache for a particular reference was recent or not. (It is not used if `FIFO` is **True**.) The integer `misses` maintains the number of cache misses. Method `ref`(*ck*, *x*) is invoked when *x* is referenced and checked against the cache *ck*.

The program declares two caches: one with 3 entries (*clock3*) and one with 4 entries (*clock4*). The interesting part is in the last block of code. It runs every sequence of references of up to 100 entries, using references in the range 1 through 5. Note that all the constants chosen in this program (3, 4, 5, 100) are the result of some experimentation—you can try other ones. To reduce the search space, the first four references are pinned to 1, 2, 3, and 4. Further reducing the search space, the program never repeats the same reference twice in a row (using the local variable `last`).

The two things to note here is the use of the **choose** expression and the **assert** statement. Using `choose` we are able to express searching through all possible strings of references without a complicated nested iteration. Using `assert` we are able to express the anomaly we are looking for.

In case you want to check if you get the right results. For `FIFO`, the program finds (in a few minutes) the same anomaly that Bélády et al. found: 1 2 3 4 1 2 5 1 2 3 4 5. For the `CLOCK` algorithm the program actually finds a shorter reference string: 1 2 3 4 2 1 2 5 1 2.

Chapter 8

Spinlock

Figure 5.3 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 threads 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 threads. Worse, Peterson relies on load and store operations to be executed atomically, but this may not be the case. There are a variety of possible reasons for this.

- Variables may have more bits than the processor's data bus. For example, variables may have 32 bits, but the data bus may only have 16 bits. Thus to store or load a variable takes two 16-bit operations each. Take, for example, a variable that has value 0xFFFFFFFF, and consider a concurrent load and store operation on the variable. The store operation wants to clear the variable, but because it takes two store operations on the bus, the load operation may return either 0xFFFF0000 or 0x0000FFFF, a value that the variable never was supposed to have. This is the case even if the processor supports a 32-bit load or store machine instruction: on the data bus it is still two operations.

- Modern processors sometimes re-orders load and store operations (out-of-order execution) for improved performance. On a sequential processor, the re-ordering is not a problem as the processor only re-orders operations on independent memory locations. However, as Exercise 6.3 showed, Peterson's algorithm breaks down if operations are re-ordered. Some memory caches can also cause non-atomic behavior of memory when shared among multiple cores.

- Even compilers, in their code generation, may make optimizations that can reorder operations, or even eliminate operations, on variables. For example, a compiler may decide that it is unnecessary to read the same variable more than once, because how could it possibly change if there are no store operations?

Peterson's algorithm relies on a *sequential consistent memory model* and hence the **sequential** statement: without it Harmony will complain about data races. More precisely, the **sequential** statement says that the program relies on memory load and store instructions operating on the indicated variables to be performed sequentially, and that this order should be consistent with the order of operations invoked on each thread. The default memory models of C and Java are not

```
1   const N = 3
2
3   shared = False
4   private = [ True, ] * N
5
6   invariant len(x for x in [shared,] + private where not x) <= 1
7
8   def tas(s, p):
9       atomic:
10          !p = !s
11          !s = True
12
13  def clear(s):
14      atomic:
15          assert !s
16          !s = False
17
18  def thread(self):
19      while choose({ False, True }):
20          # Enter critical section
21          while private[self]:
22              tas(?shared, ?private[self])
23
24          # Critical section
25          @cs: assert (not private[self]) and (atLabel(cs) == { (thread, self): 1 })
26
27          # Leave critical section
28          private[self] = True
29          clear(?shared)
30
31  for i in {0..N-1}:
32      spawn thread(i)
```

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

sequentially consistent. The unfortunately named `volatile` keyword in Java has a similar function as Harmony's `sequential` keyword. Like many constructions in Java, its `volatile` keyword was borrowed from C and C++. However, in C and C++ they do *not* provide sequential consistency, and one cannot implement Peterson's algorithm in C or C++ directly.

For proper synchronization, multi-core processors provide so-called *atomic 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 atomic instructions besides loading and storing variables, 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 the **atomic** statement. We can use **atomic** statement blocks to implement a wide variety of atomic operations. For example, we could fix the program in Figure 3.2 by constructing an atomic increment operation for a counter, like so:

```
1    def atomic_inc(ptr):
2        atomic: !ptr += 1
3
4    count = 0
5    atomic_inc(?count)
```

To support implementing locks, 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 thread. 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` threads. The approach is called *spinlock*, because each thread 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 each thread 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 thread i is executing between lines $18 \cdots 22$ (which includes 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 thread can be in the critical section at the same time. For those who read through the previous chapter: $\mathcal{I}_1 \wedge \forall i : \mathcal{I}_2(i)$ 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). 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 because we can assume, by the induction hypothesis, that *private*[i] is `True` just before exiting the critical section. Invariant $\mathcal{I}_2(i)$ is obvious as (i) thread i only proceeds to the critical section if *private*[i] is `False`, and (ii) no other thread modifies *private*[i].

Harmony can check these invariants as well. $\mathcal{I}_2(i)$ is checked by the **assert** statement. 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 premise is that there is a thread at that label. However, we would like to check $\mathcal{I}_1$ in *every state* (after the variables have been initialized). Harmony supports checking such invariants using the `invariant` keyword. The expression counts the number of `False` values and checks that the result is less than or equal to 1. Harmony checks the expression in every reachable state.

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.

Chapter 9

Blocking

In Figure 8.1 we have shown a solution based on a shared variable and a private variable for each thread. The private variables themselves are actually implemented as shared variables, but they are accessed only by their respective threads. A thread usually does not keep explicit track of whether it has a lock or not, because it is usually implied by the control flow of the program—a thread implicitly *knows* that when it is executing in a critical section it has the lock. 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 5.3, but uses test-and-set to check and set the lock variable atomically. The lock is also cleared in an **atomic** block to prevent a data race. This approach is general for any number of threads.

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 thread can execute. The latter allows multiple threads 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 atomic primitives such as 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, as mentioned before, **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. That way, when a thread makes modifications to the data structure that take multiple steps, other threads will not see the intermediate inconsistent states of the data structure.

When there is a bug in a program because some code omitted obtaining a lock before accessing a shared data structure, that is known as a *data race*. More precisely, a data race happens when

```
1   lock = False
2
3   def tas(s):
4       atomic:
5           result = !s
6           !s = True
7
8   def clear(s):
9       atomic:
10          assert !s
11          !s = False
12
13  def thread():
14      while choose({ False, True }):
15          await not tas(?lock)
16          @cs: assert atLabel(cs) == { (thread, ()): 1 }
17          clear(?lock)
18
19  for i in {1..10}:
20      spawn thread()
```

Figure 9.1: [code/csTAS.hny] Fixed version of Figure 5.3 using test-and-set.

```
1     def tas(lk):
2         atomic:
3             result = !lk
4             !lk = True
5
6     def BinSema(acquired):
7         result = acquired
8
9     def Lock():
10        result = BinSema(False)
11
12    def acquire(binsema):
13        await not tas(binsema)
14
15    def release(binsema):
16        atomic:
17            assert binsema
18            !binsema = False
19
20    def held(binsema):
21        result = !binsema
```

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

```
1   from synch import Lock, acquire, release
2
3   sequential done
4
5   count = 0
6   countlock = Lock()
7   done = [ False, False ]
8
9   def thread(self):
10      acquire(?countlock)
11      count = count + 1
12      release(?countlock)
13      done[self] = True
14      await done[1 - self]
15      assert count == 2
16
17  spawn thread(0)
18  spawn thread(1)
```

Figure 9.3: [code/UpLock.hny] Program of Figure 3.2 fixed with a lock.

there is a state in which multiple threads are trying to access the same variable, and at least one of those accesses updates the variable. In many environments, including C and Java programs, the behavior of concurrent load and store operations have tricky or even undefined semantics. One should therefore avoid data races, which is why Harmony reports them even though Harmony has sequentially consistent memory.

Harmony does not report data races in two cases. First, using the **sequential** statement, you can specify that concurrent access to the specified variables is intended. Second, if the accesses are within an **atomic** statement, then they are not considered part of a data race.

Locks are a special case of *binary semaphores*. Like locks, binary semaphores are either in an *acquired* state or a *released* state. Threads trying to acquire an already acquired binary semaphore have to wait until the binary semaphore is released. It is illegal to release a binary semaphore that is in a released state. A lock is a binary semaphore that, by convention, is initialized to a released state, and is always released by the same thread that acquired it. A binary semaphore can be initialized to an acquired state and can be released by threads other than the thread that last acquired the binary semaphore.

Harmony has a module called `synch` that includes support for binary semaphores and 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 3.2. Notice that the module hides the implementation of binary semaphores and locks. The `synch` module includes a variety of other useful synchronization primitives, which will be discussed in later chapters. Also notice the use of the sequentially consistent *done* variable for testing purposes.

```
1   import list
2
3   def BinSema(acquired):
4       result = { .acquired: acquired, .suspended: [ ] }
5
6   def Lock():
7       result = BinSema(False)
8
9   def acquire(binsema):
10      atomic:
11          if binsema→acquired:
12              stop binsema→suspended[len binsema→suspended]
13              assert binsema→acquired
14          else:
15              binsema→acquired = True
16
17  def release(binsema):
18      atomic:
19          assert binsema→acquired
20          if binsema→suspended == [ ]:
21              binsema→acquired = False
22          else:
23              go (list.head(binsema→suspended)) ()
24              binsema→suspended = list.tail(binsema→suspended)
25
26  def held(binsema):
27      result = binsema→acquired
```

Figure 9.4: [modules/synchS.hny] The binary semaphore interface in the `synchS` module uses suspension.

We call a thread *blocked* if a thread cannot change the state or terminate unless another thread changes the state first. A thread trying to acquire a test-and-set spinlock held by another thread is a good example of a thread being blocked. The only way forward is if the other thread releases the spinlock. A thread that is in an infinite loop is also considered blocked.

In most operating systems, threads are virtual (as opposed to "raw CPU cores") and can be suspended until some condition changes. For example, a thread that is trying to acquire a lock can be suspended until the lock is available. In Harmony, a thread can suspend itself and save its context (state) in a shared variable. Recall that the context of a thread contains its program counter, stack, and registers. A context is a regular (if complex) Harmony value. The syntax of the expression that a thread executes to suspend itself is as follows:

stop *s*

This causes the context of the thread to be saved in *s* and the thread to be no longer running. Another thread can revive the thread using the **go** statement:

go *C R*;

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

There is a second version of the `synch` module, called `synchS`, that uses suspension instead of spinlocks. Figure 9.4 shows the same interface as Figure 9.2 implemented using suspension. The interface is implemented as follows:

- A binary semaphore maintains both a boolean indicating whether the binary semaphore is currently acquired and a list of contexts of threads that want to acquire the binary semaphore.
- `acquire`() acquires the binary semaphore if available and suspends the invoking thread if not. In the latter case, the context of the thread is added to the end of the list of contexts. Note that **stop** is called within an **atomic** statement—this is the only exception to an atomic statement running to completion. While the thread is running no other threads can run, but when the thread suspends itself other threads can run.
- `release`() checks to see if any threads are waiting to acquire the binary semaphore. If so, it uses the `head` and `tail` methods from the `list` module (see Appendix C) to resume the first thread that got suspended and to remove its context from the list.

Selecting the first thread 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 binary semaphore implementations as it prevents *starvation*: every thread gets a chance to acquire the binary semaphore (assuming every thread eventually releases it). Chapter 15 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 the model checker 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 threads wait to acquire a binary semaphore, 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.

```
x, y = 0, 100

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

spawn checker()
spawn setX(50)
```

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

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 `tryAcquire`(b) as an additional interface for both the `synch` and `synchS` modules. This interface is like `acquire`(b) but never blocks. It returns `True` if the binary semaphore was available (and now acquired) or `False` if the binary semaphore was already acquired. Hint: you do not have to change the existing code.

9.4 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 9.6 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. 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. Fix the code in Figure 9.6. 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.

```
from synch import Lock, acquire, release

const N_ACCOUNTS = 2
const N_CUSTOMERS = 2
const N_ATMS = 2
const MAX_BALANCE = 1

accounts = [ { .lock: Lock(), .balance: choose({0..MAX_BALANCE})}
                        for i in {1..N_ACCOUNTS} ]

invariant min(accounts[acct].balance for acct in {0..N_ACCOUNTS–1}) >= 0

def atm_check_balance(acct):    # return the balance on acct
    acquire(?accounts[acct].lock)
    result = accounts[acct].balance
    release(?accounts[acct].lock)

def atm_withdraw(acct, amount): # withdraw amount from acct
    acquire(?accounts[acct].lock)
    accounts[acct].balance –= amount
    result = True               # return success
    release(?accounts[acct].lock)

def customer(atm, acct, amount):
    let bal = atm_check_balance(acct):
        if amount <= bal:
            atm_withdraw(acct, amount)

for i in {1..N_ATMS}:
    spawn customer(i, choose({0..N_ACCOUNTS–1}),
                    choose({0..MAX_BALANCE}))
```

Figure 9.6: [code/atm.hny] Withdrawing money from an ATM.

Chapter 10

Concurrent Data Structures

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

- $q =$ `Queue`(): allocate a new queue;
- `put`(q, v): add v to the tail of the queue;
- $r =$ `get`(q): returns $r =$ `None` if q is empty or $r = v$ if v was at the head of the queue.

See Figure 10.1 for a simple demonstration 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. (`None` is a special address value that is not the address of any memory location.)

`queue.Queue()` returns a new queue object consisting of a `None` head and tail pointer and a lock. `queue.put`(q, v) and `queue.get`(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.

An important thing to note in Figure 10.1 is Lines 7 and 8. It would be incorrect to replace these by:

> **assert** `queue.get`(q) in { **None**, 1, 2 }

The reason is that `queue.get()` changes the state by acquiring a lock, but the expressions in **assert** statements (or **invariant** statements) are not allowed to change the state. In general, when calling methods in **assert** or **invariant** statements, one has to be convinced that those methods cannot change the state in any way.

Figure 10.3 shows another concurrent queue implementation [MS96]. It is well-known, but what is not often realized is that it requires sequentially consistent memory, which is not said

```
import queue

def sender(q, v):
   queue.put(q, v)

def receiver(q):
   let v = queue.get(q):
      assert v in { None, 1, 2 }

demoq = queue.Queue()
spawn sender(?demoq, 1)
spawn sender(?demoq, 2)
spawn receiver(?demoq)
spawn receiver(?demoq)
```

Figure 10.1: [code/queuedemo.hny] Using a concurrent queue.

explicitly in the paper. As a result, the algorithm is not guaranteed to work correctly with most modern programming languages and computer hardware. But it is still useful to study it. The implementation uses separate locks for the head and the tail, allowing a `put` and a `get` operation to proceed concurrently. To avoid contention between the head and the tail, the queue uses a dummy node at the head of the linked list. Except initially, the dummy node is the last node that was dequeued. Note that neither the `head` nor `tail` pointer are ever `None`. The problem is when the queue is empty and there are concurrent `get` and `put` operations. They obtain separate locks and then concurrently access the `next` field in the dummy node—a data race with undefined semantics in most environments.

A queue has the nice property that usually only the head or the tail is accessed. However, in many data structures it is necessary to "walk" the data structure, an operation that can take significant time. In such a case, a single lock (known as a "big lock") for the entire data structure might restrict concurrency to an unacceptable level. To reduce the granularity of locking, each node in the data structure must be endowed with its own lock instead.

Figure 10.4 implements an ordered linked list of integers without duplicates. (Figure 10.5 contains test code.) Values can be added using `insert` or deleted using `remove`. Method `contains` checks if a particular value is in the list.

The list has two dummy "book-end" nodes with values `-inf` and `inf` (similar to the Python `math.inf` constant). 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.

Each node has a lock, a value, and *next*, a pointer to the next node (which is `None` for the final `inf` node). The `_find(lst, v)` helper method first 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

```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def Queue():
5       result = { .head: None, .tail: None, .lock: Lock() }
6
7   def put(q, v):
8       let node = malloc({ .value: v, .next: None }):
9           acquire(?q→lock)
10          if q→tail == None:
11              q→tail = q→head = node
12          else:
13              q→tail→next = node
14              q→tail = node
15          release(?q→lock)
16
17  def get(q):
18      acquire(?q→lock)
19      let node = q→head:
20          if node == None:
21              result = None
22          else:
23              result = node→value
24              q→head = node→next
25              if q→head == None:
26                  q→tail = None
27              free(node)
28      release(?q→lock)
```

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

```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def Queue():
5       let dummy = malloc({ .value: (), .next: None }):
6           result = { .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() }
7
8   def put(q, v):
9       let node = malloc({ .value: v, .next: None }):
10          acquire(?q→tllock)
11          q→tail→next = node
12          q→tail = node
13          release(?q→tllock)
14
15  def get(q):
16      acquire(?q→hdlock)
17      let dummy = q→head
18      let node = dummy→next:
19          if node == None:
20              result = None
21              release(?q→hdlock)
22          else:
23              result = node→value
24              q→head = node
25              release(?q→hdlock)
26              free(dummy)
```

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

```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def _node(v, n):     # allocate and initialize a new list node
5       result = malloc({ .lock: Lock(), .value: v, .next: n })
6
7   def _find(lst, v):
8       let before = lst:
9           acquire(?before→lock)
10          let after = before→next:
11              acquire(?after→lock)
12              while after→value < v:
13                  release(?before→lock)
14                  before = after
15                  after = before→next
16                  acquire(?after→lock)
17              result = (before, after)
18
19  def LinkedList():
20      result = _node(-inf, _node(inf, None))
21
22  def insert(lst, v):
23      let before, after = _find(lst, v):
24          if after→value != v:
25              before→next = _node(v, after)
26          release(?after→lock)
27          release(?before→lock)
28
29  def remove(lst, v):
30      let before, after = _find(lst, v):
31          if after→value == v:
32              before→next = after→next
33              release(?after→lock)
34              free(after)
35          else:
36              release(?after→lock)
37          release(?before→lock)
38
39  def contains(lst, v):
40      let before, after = _find(lst, v):
41          result = after→value == v
42          release(?after→lock)
43          release(?before→lock)
```

Figure 10.4: [code/linkedlist.hny] Linked list with fine-grained locking.

```
from linkedlist import *

mylist = LinkedList()

def thread1():
    insert(mylist, 1)
    let x = contains(mylist, 1):
        assert x

def thread2(v):
    insert(mylist, v)
    remove(mylist, v)

spawn thread1()
spawn thread2(0)
spawn thread2(2)
```

Figure 10.5: [code/lltest.hny] Test code for Figure 10.4.

to climbing a tree hand-over-hand. Using `_find` the `insert`, `remove`, and `contains` methods are fairly straightforward.

Like the queue in Figure 10.2, 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. 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 point for the `insert` operation coincides exactly with the update of the *before*.`next` pointer. The linearization point of a `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

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

10.2 Add a method `length`(q) to Figure 10.2 that returns the length of the given queue. The complexity of the method should be $O(1)$, which is to say that you should maintain the length of the queue as a field member and update it in `put` and `get`.

10.3 Write a method `check`(q) that checks the integrity of the queue in Figure 10.2. In particular, it should check the following integrity properties:

- If the list is empty, q→`tail` should be `None`. Otherwise, the last element in the linked list starting from q→`head` should equal q→`head`. Moreover, q→`tail`→*next* should be `None`;

- The length field that you added in Exercise 10.3 should equal the length of the list.

Method `check`(q) should not obtain a lock; instead add the following line just before releasing the lock in `put` and `get`:

```
assert check()
```

10.4 Add a method `remove`(q, v) to Figure 10.2 that removes all occurrences of v, if any, from queue q.

10.5 The test program in Figure 10.1 is a not thorough test program. Design and implement a test program for Figure 10.1. Make sure you *test* the test program by trying it out against some buggy queue implementations.

10.6 Create a thread-safe sorted binary tree. Implement a module `bintree` with methods

10.7 Add methods to the data structure in Figure 10.4 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?

10.8 Create a thread-safe sorted binary tree. Implement a module `bintree` with methods `BinTree`() to create a new binary tree, `insert`(t, v) that inserts v into tree t, and `contains`(t, v) that checks if v is in tree t. Use a single lock per binary tree.

10.9 Create a binary tree that uses, instead of a single lock per tree, a lock for each node in the tree.

Chapter 11

Testing

Testing is a way to increase confidence in the correctness of an implementation. Figure 10.1 demonstrates how concurrent queues may be used, but it is not a very thorough test program for an implementation such as the one in Figure 10.2 and does little to increase our confidence in its correctness. To wit, if `get()` always returned 1, the program would find no problems. In this chapter, we will look at approaches to testing concurrent code.

The framework in Figure 5.1 works well for thoroughly testing mutual exclusion and progress in critical sections, but depends on `atLabel`, an unusual operator specific to Harmony that is mostly useful for testing mutual exclusion. It turns out that we do not need labels to express this. Figure 11.1 tests mutual exclusion and progress for critical sections implemented using locks. The test uses a simple atomic counter that is incremented before entering the critical section and decremented after leaving it. Mutual exclusion holds if the counter never goes over 1.

Test programs themselves should be tested. Just because a test program works with a particular implementation does not mean the implementation is correct—it may be that the implementation is incorrect but the test program does not have enough coverage to find any bugs in the implementation. In the case of testing mutual exclusion with atomic counters, you will want to see if you can use this method to find the problems in Figure 5.3, Figure 5.5, and Figure 5.7. Conversely, a test program may be broken in that it finds bugs that do not exist. So, in this case, you also want to check if an atomic counter solution works with known correct solutions such as Figure 6.1 and Figure 8.1 if possible.

As with critical sections, when testing a concurrent data structure we need a specification for its correctness. A good place to start is thinking about a *sequential* specification for queues. A specification is simply a program, written at a high level of abstraction. Figure 11.2 shows a sequential specification of a queue in Harmony (exploiting some methods from the `list` module described in Section C.4).

First we can check if the queue implementation in Figure 10.2 meets the sequential queue specification in Figure 11.2. To check if the queue implementation meets the specification, we need to see if any sequence of queue operations in the implementation matches a corresponding sequence in the specification. We say that the implementation and the specification have the same *behavior*. Figure 11.3 presents a test program that does exactly this, for sequences of up to `NOPS` queue operations. It maintains two queues:

implq: the queue of the implementation;

```
1   import synch
2
3   lock = synch.Lock()
4
5   count = 0
6   invariant 0 <= count <= 1
7
8   def thread():
9       synch.acquire(?lock)
10
11      atomic: count += 1
12
13      # critical section is here
14      assert count == 1
15
16      atomic: count -= 1
17
18      synch.release(?lock)
19
20  for i in {1..100}:
21      spawn thread()
```

Figure 11.1: [code/cslock.hny] Critical section testing without labels.

```
1   import list
2
3   def Queue():
4       result = [ ]
5
6   def put(q, v):
7       !q = list.append(!q, v)
8
9   def get(q):
10      if !q == [ ]:
11          result = None
12      else:
13          result = list.head(!q)
14          !q = list.tail(!q)
```

Figure 11.2: [code/queuespec.hny] Sequential queue specification.

```
1   import queue, queuespec
2
3   const NOPS = 4
4   const VALUES = { 1..NOPS }
5
6   implq = queue.Queue()
7   specq = queuespec.Queue()
8
9   for i in {1..NOPS}:
10      let op = choose({ .get, .put }):
11          if op == .put:
12              let v = choose(VALUES):
13                  queue.put(?implq, v)
14                  queuespec.put(?specq, v)
15          else:
16              let v = queue.get(?implq)
17              let w = queuespec.get(?specq):
18                  assert v == w
```

Figure 11.3: [code/qtestseq.hny] Sequential queue test.

specq: the queue of the specification

For each operation, the code first decides whether to perform a `get` or `put` operation. In the case of a `put` operation, the code also decides which value to append to the queue. All operations are performed on both the queue implementation and the queue specification. In the case of `get`, the results of the operation on both the implementation and specification are checked against one another.

As with any other test program, Figure 11.3 may not trigger extant bugs, but it nonetheless inspires reasonable confidence that the queue implementation is correct, at least sequentially. The higher `NOPS`, the higher the confidence. It is possible to write similar programs in other languages such as Python, but the `choose` expression in Harmony makes it relatively easy to explore all corner cases. For example, a common programming mistake is to forget to update the `tail` pointer in `get()` in case the queue becomes empty. Normally, it is a surprisingly tricky bug to find. You can comment out those lines in Figure 10.2 and run the test program—it should easily find the bug and explain exactly how the bug manifests itself, adding confidence that the test program is reasonably thorough.

The test program also finds some common mistakes in using locks, such as forgetting to release a lock when the queue is empty, but it is not designed to find concurrency bugs in general. If you remove all `acquire()` and `release()` calls from Figure 10.2, the test program will not (and should not) find any errors. The next step is to test if the queue implementation meets the *concurrent* queue specification or not. But we have not yet shown what the concurrent queue specification is.

We briefly mentioned the notion of *linearizability* in Chapter 10. Basically, we want a concurrent queue to behave consistently with a sequential queue in that all `put` and `get` operations should

```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock(), .time: 0 }

def _linpoint(q):
    atomic:
        this.qtime = q→time
        q→time += 1

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→tail == None:
            q→tail = q→head = node
        else:
            q→tail→next = node
            q→tail = node
        _linpoint(q)
        release(?q→lock)

def get(q):
    acquire(?q→lock)
    let node = q→head:
        if node == None:
            result = None
        else:
            result = node→value
            q→head = node→next
            if q→head == None:
                q→tail = None
            free(node)
    _linpoint(q)
    release(?q→lock)
```

Figure 11.4: [code/queuelin.hny] Queue implementation with linearization points.

```
1   import queuelin, queuespec
2
3   const NOPS = 4
4   const VALUES = { 1..NOPS }
5
6   sequential qtime
7   qtime = 0
8
9   implq = queuelin.Queue()
10  specq = queuespec.Queue()
11
12  def thread():
13      let op = choose({ .get, .put }):
14          if op == .put:
15              let v = choose(VALUES):
16                  queuelin.put(?implq, v)
17                  await qtime == this.qtime
18                  queuespec.put(?specq, v)
19          else:
20              let v = queuelin.get(?implq):
21                  await qtime == this.qtime
22                  let w = queuespec.get(?specq):
23                      assert v == w
24      atomic:
25          qtime += 1
26
27  for i in {1..NOPS}:
28      spawn thread()
```

Figure 11.5: [code/qtestconc.hny] Concurrent queue test.

appear to happen in a total order. Moreover, we want to make sure that if some `put` or `get` operation o_1 finished before another operation o_2 started, then o_1 should appear to happen before o_2 in the total order. If these two conditions are met, then we say that the concurrent queue implementation is linearizable.

In general, if a data structure is protected by a single lock and every operation on that data structure starts with acquiring the lock and ends with releasing the lock, it will automatically be linearizable. The queue implementation in Figure 10.2 does not quite match this pattern, as the `put` operation allocates a new node before acquiring the lock. However, in this case that is not a problem, as the new node has no dependencies on the queue when it is allocated.

Still, it would be useful to check in Harmony that Figure 10.2 is linearizable. To do this, instead of applying the operations sequentially, we want the test program to invoke the operations concurrently, consider all possible interleavings, and see if the result is consistent with an appropriate sequential execution of the operations.

Harmony provides support for testing linearizability, but requires that the programmer identifies what are known as *linearization points* in the implementation that indicate exactly *which* sequential execution the concurrent execution must align with. Figure 11.4 is a copy of Figure 10.2 extended with linearization points. For each operation (`get` and `put`), the corresponding linearization point must occur somewhere between acquiring and releasing the lock. Each linearization point execution is assigned a logical timestamp. Logical timestamps are numbered $0, 1, ...$ To do so, we have added a counter (`time`) to the `Queue`. Method `_linpoint` saves the current counter in **this**.`qtime` and increments the counter. The **this** dictionary maintains *thread-local state* associated with the thread (Chapter 4)—it contains variables that can be accessed by any method in the thread.

Given the linearization points, Figure 11.5 shows how linearizability can be tested. The test program is similar to the sequential test program (Figure 11.3) but starts a thread for each operation. The operations are executed concurrently on the concurrent queue implementation of Figure 11.4, but they are executed sequentially on the sequential queue specification of Figure 11.2. To that end, the test program maintains a global time variable *qtime*, and each thread waits until the timestamp assigned to the last concurrent queue operation matches *qtime* before invoking the sequential operation in the specification. Afterward, it atomically increments the shared *qtime* variable. This results in the operations being executed sequentially against the sequential specification in the same order of the linearization points of the concurrent specification.

If you want a purely black box test for testing a queue implementation, then adding linearization points is not possible. We will present a few test programs that try to identify incorrect behavior of a black box queue implementation in the presence of concurrency.

Figure 11.6 shows a program that checks concurrent `put` operations using `N` threads. Thread `putter(`v`)` adds v to the queue. Thread `main()` waits until all `putter` threads have finished, and then dequeues `N` items, verifying that they form the set $\{0..\texttt{N}-1\}$.

Similarly, Figure 11.7 shows a program that checks concurrent `get` operations using `N` threads. The queue is initialized with the sequence $0..\texttt{N}-1$. Thread `getter(`v`)` removes an entry from the queue and makes sure that no other thread got the same entry. Thread `main()` waits until all `getter` threads have finished and checks to make sure that the queue is empty.

Figure 11.8 checks a mixture of concurrent `get` and `put` operations. It starts `N` `getter` threads and `N` `putter` threads. Thread `main` waits until all have finished, and then checks to make sure that whatever is left on the queue was not also consumed by some `getter` thread. Finally, it makes sure that the queue is empty once all the values that were added to it by the `putter` threads have been consumed.

```
1   import queue
2
3   const N = 3
4
5   sequential putcount
6
7   testq = queue.Queue()
8   putcount = 0
9
10  def putter(v):
11      queue.put(?testq, v)
12      atomic:
13          putcount += 1
14
15  def main():
16      await putcount == N
17      let gotten = {}:
18          while gotten != {0..N–1}:
19              let v = queue.get(?testq):
20                  assert v not in gotten
21                  gotten |= {v}
22      let v = queue.get(?testq):
23          assert v == None
24
25  for i in {0..N–1}:
26      spawn putter(i)
27  spawn main()
```

Figure 11.6: [code/qtest1.hny] Concurrent `put()`.

```
1   import queue
2
3   const N = 3
4
5   sequential gotten
6
7   testq = queue.Queue()
8   for i in {0..N–1}:
9      queue.put(?testq, i)
10  gotten = {}
11
12  def getter():
13     let v = queue.get(?testq):
14        atomic:
15           assert v not in gotten
16           assert v != None
17           gotten |= {v}
18
19  def main():
20     await gotten == {0..N–1}
21     let v = queue.get(?testq):
22        assert v == None
23
24  for i in {0..N–1}:
25     spawn getter()
26  spawn main()
```

Figure 11.7: [code/qtest2.hny] Concurrent `get()`.

```
1   import queue
2
3   const N = 3
4
5   sequential putcount, getcount
6
7   testq = queue.Queue()
8   gotten = {}
9   putcount = getcount = 0
10
11  def putter(v):
12      queue.put(?testq, v)
13      atomic:
14          putcount += 1
15
16  def getter():
17      let v = queue.get(?testq):
18          atomic:
19              assert v not in gotten
20              if v != None:
21                  gotten |= {v}
22              getcount += 1
23
24  def main():
25      await (getcount == N) and (putcount == N)
26      while gotten != {0..N-1}:
27          let v = queue.get(?testq):
28              assert v not in gotten
29              gotten |= {v}
30      let v = queue.get(?testq):
31          assert v == None
32
33  for i in {0..N-1}:
34      spawn putter(i)
35      spawn getter()
36  spawn main()
```

Figure 11.8: [code/qtest3.hny] Mix of concurrent `get()` and `put()` operations.

```
1   import queue
2
3   q1 = q2 = queue.Queue()
4   queue.put(?q1, 1)
5   queue.put(?q2, 2)
6
7   def getter(q, v):
8       let x = queue.get(q):
9           assert x == v
10
11  spawn getter(?q1, 1)
12  spawn getter(?q2, 2)
```

Figure 11.9: [code/qtest4.hny] Test with multiple queues.

A common mistake that people make is to use a globally shared variable in the implementation of a concurrent data structure. The `get()` method of the queue implementation uses a local variable *node*, declared using a **let** statement, but it would be easy to mistakenly use a global variable *node* instead. If there is only one queue, this is not an issue. So, it is important to test not just one queue at a time, but also multiple queues. Figure 11.9 gives an example that tests the `get` method for use of global variables. Even if the assertion does not fail, Harmony might find a data race if there are global variables shared between different queues.

While having five different test programs is fine, it would be nice if we can combine them all in one test program. This has to be done with some care, as running the concurrent test programs simultaneously would lead to a state space explosion. Instead, we can leverage the **choose** expression. Figure 11.10 shows how one might go about this. Harmony will run all the tests, but the tests do not interfere with one another.

Exercises

11.1 Write a sequential specification for Figure 10.4. (Hint: it implements a set of integers.) Write a Harmony program that checks if Figure 10.4 meets the sequential specification.

11.2 Extend Figure 10.4 with linearization points and check if the implementation is linearizable.

11.3 Check if the extended queue implementation of Exercise 10.4 is linearizable.

```
    def seq_test():
        pass

    def conc_test1():
        pass

    def conc_test2():
        pass

    def conc_test3():
        pass

    seq_test()
    let test = choose({ conc_test1, conc_test2, conc_test3 }):
        test()
```

Figure 11.10: [code/multitest.hny] Running a sequential test followed by several concurrent tests.

Chapter 12

Debugging

So you wrote a Harmony program and Harmony reports a bug. Often you may just be able to figure it out by staring at the code and going through some easy scenarios, but what if you don't? The output of Harmony can be helpful in that case.

Figure 12.1 contains an attempt at a queue implementation where the queue is implemented by a linked list, with the first node being a `dummy` node to prevent data races. Each node in the list contains a lock. The `put`() method walks the list until it gets to the last node, each time acquiring the lock to access the node's fields. When `put`() gets to the last node in the list, it appends a new one. The `get`() method locks the first (dummy) node, removes the second from the list and frees it. The method returns the value from the removed node.

Let us run the code through some of the test programs in the last chapter. Harmony does not detect any issues with the sequential test in Figure 11.3. The concurrent `put` test of Figure 11.6 as well as the concurrent `get` test of Figure 11.7 also find no problems. However, when we run the new queue code through the test in Figure 11.8, Harmony finds a problem.

Figure 12.2 shows the Harmony output of running the test in Figure 11.8 against the queue code in Figure 12.1. There is quite a bit of information in the Harmony output, and it's important to learn to navigate through it. Let's start with looking at the red text. Harmony found a safety violation (something bad happened during one of the possible executions), and in particular `putter`(1) (thread T2) was trying to dereference `alloc`.*pool*[0].`lock` in turn 5.

The `alloc` module maintains a shared array *pool* that it uses for dynamic allocation. Apparently `putter`(1) tried to access *pool*[0], but it does not exist, meaning that either it was not yet allocated, or it had been freed since it was allocated. When we look at the top half of the figure, we see that in fact `putter`(0) (T1) allocated *pool*[0] in turn 2, while `getter`() (T4) released it in turn 4.

So how did we get there? We can start by single stepping through the actions of `putter`(0) by clicking on its first block. By hitting return repeatedly, we can go through the lines of code that it is executing. Doing so, we can see that it allocates *pool*[0] and uses that as the node to contain value 0 and add this node to the queue *qtest*. At the end of its turn, `putter`(0) has just finished `put`(0). Looking at the state in the top right, everything looks good. The first node (*testq*) points to the allocated node with the value 0 in it, and its `next` pointer is **None**. The locks on both nodes are released. So far so good.

```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def Queue():
5       result = { .next: None, .value: None, .lock: Lock() }
6
7   def put(q, v):
8       let node = malloc({ .next: None, .value: v, .lock: Lock() }):
9           while q != None:
10              acquire(?q→lock)
11              let n = q→next:
12                  if n == None:
13                      q→next = node
14                  release(?q→lock)
15                  q = n
16
17  def get(q):
18      acquire(?q→lock)
19      if q→next == None:
20          result = None
21      else:
22          let node = q→next:
23              q→next = node→next
24              result = node→value
25              free(node)
26      release(?q→lock)
```

Figure 12.1: [code/queuebug.hny] A buggy queue implementation.

Issue: Safety violation

Turn	Thread	Instructions Executed	PC	Shared Variables: alloc.next	alloc.pool.0.lock	alloc.pool.0.next	alloc.pool.0.value	alloc.pool.1.lock	alloc.pool.1.next	alloc.pool.1.value	getcount	gotten	putcount	testq.lock	testq.next	testq.value
1	T0: __init__()		1081	0							0	{ }	0	False	None	None
2	T1: putter(0)		968	1	False	None	0				0	{ }	0	False	?alloc.pool[0]	None
3	T2: putter(1)		502	2	False	None	0	False	None	1	0	{ }	0	False	?alloc.pool[0]	None
4	T4: getter()		554	2				False	None	1	0	{ }	0	True	None	None
5	T2: putter(1)		504	2				False	None	1	0	{ }	0	True	None	None

modules/synch.hny:6 result = !lk

500	Jump 781
501	Frame tas lk
502	AtomicInc
503	LoadVar lk
504	Load
505	StoreVar result
506	LoadVar lk
507	Push True
508	Store
509	AtomicDec

Threads

ID	Status	Stack Trace	Stack Top
T0	terminated	__init__()	
T1	runnable	putter(0) this: ( ), v: 0	
T2	failed atomic	putter(1) v: 1 put(?testq, 1) node: ?alloc.pool[1], q: ?alloc.pool[0], v: 1 acquire(?alloc.pool[0].lock) binsema: ?alloc.pool[0].lock tas(?alloc.pool[0].lock) lk: ?alloc.pool[0].lock, this: ( ) Load: unknown address ?alloc.pool[0].lock	
T3	runnable	putter(2)	
T4	runnable	getter() get(?testq) q: ?testq, result: 0 release(?testq.lock) binsema: ?testq.lock, this: ( )	
T5	runnable	getter()	
T6	runnable	getter()	
T7	runnable	main()	

Figure 12.2: Harmony output of running Figure 11.8 against Figure 12.1.

Next `putter`(1) (thread T2) takes a turn. It gets to line 10 in Figure 12.1 where it's trying to acquire $q \rightarrow$`lock`. If we look at the bottom right, we see that q points to *pool*[0], the node that has value 0 in it. Again, so far so good.

But before `putter`(1) obtains the lock on *pool*[0], `getter`() (thread T4) starts running. `getter`() acquires the lock on *qtest*, which still points to *pool*[0]. `getter` then extracts the value from *pool*[0] and then releases it. At this point *pool*[0] no longer exists, but then `putter`(1) starts running again from where it left off. It was about to acquire the lock on *pool*[0], but now that node no longer exists.

To fix the code, we have to use hand-over-hand locking (Chapter 10). Figure 12.3 shows an implementation that uses the same data structure as Figure 12.1 but uses hand-over-hand locking, both for `put`() and for `get`(). It passes all tests.

```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def Queue():
5       result = { .next: None, .value: None, .lock: Lock() }
6
7   def put(q, v):
8       let node = malloc({ .next: None, .value: v, .lock: Lock() }):
9           acquire(?q→lock)
10          let n = q→next:
11              while n != None:
12                  acquire(?n→lock)
13                  release(?q→lock)
14                  q = n
15                  n = n→next
16              q→next = node
17          release(?q→lock)
18
19  def get(q):
20      acquire(?q→lock)
21      if q→next == None:
22          result = None
23      else:
24          let node = q→next:
25              acquire(?node→lock)
26              q→next = node→next
27              result = node→value
28              release(?node→lock)
29              free(node)
30      release(?q→lock)
```

Figure 12.3: [code/queuefix.hny] Queue implementation with hand-over-hand locking.

Chapter 13

Conditional Waiting

Critical sections enable multiple threads to easily share data structures whose modification requires multiple steps. A critical section only allows one thread to execute the code of the critical section at a time. Therefore, when a thread arrives at a critical section, the thread blocks until there is no other thread in the critical section.

Sometimes it is useful for a thread to block waiting for additional conditions. For example, when dequeuing from an empty shared queue, it may be useful for the thread to block until the queue is non-empty instead of returning an error. The alternative would be *busy waiting* (aka *spin-waiting*), where the thread repeatedly tries to dequeue an item until it is successful. Doing so wastes CPU cycles and adds contention to queue access. A thread that is busy waiting until the queue is non-empty cannot make progress until another thread enqueues an item. However, the thread is not considered *blocked* because it is changing the shared state by repeatedly acquiring and releasing the lock. We distinguish *passive busy waiting* and *active busy waiting*. A process that is waiting for a condition without changing the state (like in a spinlock) is passively busy waiting. A process that is waiting for a condition while changing the state (such as repeatedly trying to dequeue an item, which requires acquiring a lock) is actively busy waiting.

We would like to find a solution to *conditional waiting* so that a thread blocks until the condition holds—or at least most of the time. Before we do so, we will give two classic examples of synchronization problems that involve conditional waiting: *reader/writer locks* and *bounded buffers*.

13.1 Reader/Writer Locks

Locks are useful when accessing a shared data structure. By preventing more than one thread 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 threads are simply reading the data structure. In many applications, reads are the majority of all accesses, and read operations do not conflict with one another. 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 acquire the lock. This is called a *reader/writer lock* [CHP71]. A reader/writer lock is an object whose abstract (and opaque) state contains two integer counters:

```
1   import RW
2
3   rw = RW.RWlock()
4
5   def thread():
6      while choose({ False, True }):
7         if choose({ .read, .write }) == .read:
8            RW.read_acquire(?rw)
9            @rcs: assert atLabel(wcs) == ()
10           RW.read_release(?rw)
11        else:                    # .write
12           RW.write_acquire(?rw)
13           @wcs: assert (atLabel(wcs) == { (thread, ()): 1 }) and
14                        (atLabel(rcs) == ())
15           RW.write_release(?rw)
16
17  for i in {1..3}:
18     spawn thread()
19
```

Figure 13.1: [code/RWtest.hny] Test code for reader/writer locks.

1. *nreaders*: the number of readers
2. *nwriters*: the number of writers

satisfying the following invariant:

$$(nreaders \geq 0 \wedge nwriters = 0) \vee (nreaders = 0 \wedge 0 \leq nwriters \leq 1)$$

There are four operations on a reader/writer lock *rw*:

- `read_acquire`(*rw*): waits until $nwriters = 0$ and then increments *nreaders*;
- `read_release`(*rw*): decrements *nreaders*;
- `write_acquire`(*rw*): waits until $nreaders = nwriters = 0$ and then sets *nwriters* to 1;
- `write_release`(*rw*): sets *nwriters* to 0.

Figure 13.1 shows how reader/writer locks operations may be tested. Similar to ordinary locks, a thread is restricted in how it is allowed to invoke these operations. In particular, a thread can only release a reader/writer lock for reading if it acquired it for reading and the same for writing. Moreover, a thread is only allowed the acquire a reader/writer lock once.

13.2 Bounded Buffer

A *bounded buffer* (aka *ring buffer*) is a queue with the usual `put/get` interface, but implemented using a circular buffer of a certain length and two pointers: the *tail* points where new items are enqueued and the *head* points where items are dequeued. If the buffer is full, the enqueuer must wait; if the buffer is empty, the dequeuer must wait. This problem is known as the "Producer/Consumer Problem" and was proposed by Dijkstra [Dij72]. Multiple producers and multiple consumers may all share the same bounded buffer.

The producer/consumer pattern is common. Threads may be arranged in *pipelines*, where each upstream thread is a producer and each downstream thread is a consumer. Or threads 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 thread 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.

Chapter 14

Split Binary Semaphores

The Split Binary Semaphore (SBS) approach is a general technique for implementing conditional waiting. It was originally proposed by Tony Hoare and popularized by Edsger Dijkstra [Dij79]. SBS is an extension of a critical section that is protected by a single binary semaphore. If there are n *waiting conditions*, then SBS uses $n + 1$ binary semaphores to protect the critical section. An ordinary critical section has no waiting conditions. For example, a bounded buffer has two waiting conditions:

1. consumers waiting for the buffer to be non-empty;
2. producers waiting for an empty slot in the buffer.

So, it will require 3 binary semaphores if the SBS technique is applied.

Think of each of these semaphores as a gate that a thread must go through in order to enter the critical section. A gate is either open or closed. Initially, exactly one gate, the main gate, is open. Each of the other gates, the *waiting gates*, is associated with a waiting condition. When a gate is open, one thread can enter the critical section, closing the gate behind it.

When leaving the critical section, the thread must open exactly one of the gates, but it does not have to be the gate that it used to enter the critical section. In particular, when a thread leaves the critical section it should check for each waiting gate if its waiting condition holds and if there are threads trying to get through the gate. If there is such a gate, it must select one and open that gate. If there is no such gate, it must open the main gate.

Finally, if a thread is executing in the critical section and needs to wait for a particular condition, it leaves the critical section and waits for the gate associated with that condition to open.

Note that the following invariants hold:

- At any time, at most one gate is open;
- If some gate is open, then no thread is in the critical section. Equivalently, if some thread is in the critical section, all gates are closed;
- At any time, at most one thread is in the critical section.

```
1   from synch import BinSema, acquire, release
2
3   def RWlock():
4       result = {
5               .nreaders: 0, .nwriters: 0, .mutex: BinSema(False),
6               .r_gate: { .sema: BinSema(True), .count: 0 },
7               .w_gate: { .sema: BinSema(True), .count: 0 }
8           }
9
10  def release_one(rw):
11      if (rw→nwriters == 0) and (rw→r_gate.count > 0):
12          release(?rw→r_gate.sema)
13      elif ((rw→nreaders + rw→nwriters) == 0) and (rw→w_gate.count > 0):
14          release(?rw→w_gate.sema)
15      else:
16          release(?rw→mutex)
17
18  def read_acquire(rw):
19      acquire(?rw→mutex)
20      if rw→nwriters > 0:
21          rw→r_gate.count += 1; release_one(rw)
22          acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
23      rw→nreaders += 1
24      release_one(rw)
25
26  def read_release(rw):
27      acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
28
29  def write_acquire(rw):
30      acquire(?rw→mutex)
31      if (rw→nreaders + rw→nwriters) > 0:
32          rw→w_gate.count += 1; release_one(rw)
33          acquire(?rw→w_gate.sema); rw→w_gate.count −= 1
34      rw→nwriters += 1
35      release_one(rw)
36
37  def write_release(rw):
38      acquire(?rw→mutex); rw→nwriters −= 1; release_one(rw)
```

Figure 14.1: [code/RWsbs.hny] Reader/Writer Lock using Split Binary Semaphores.

The main gate is implemented by a binary semaphore, initialized in the released state (signifying that the gate is open). The waiting gates each consist of a pair: a counter that counts how many threads are waiting behind the gate and a binary semaphore initialized in the acquired state (signifying that the gate is closed).

We will illustrate the technique using the reader/writer problem. Figure 14.1 shows code. The first step is to enumerate all waiting conditions. In the case of the reader/writer problem, there are two: a thread that wants to read may have to wait for a writer to leave the critical section, while a thread that wants to write may have to wait until all readers have left the critical section or until a writer has left. The state of a reader/writer lock thus consists of the following:

- *nreaders*: the number of readers in the critical section;
- *nwriters*: the number of writers in the critical section;
- *mutex*: the main gate binary semaphore;
- *r_gate*: the waiting gate used by readers, consisting of a binary semaphore and the number of readers waiting to enter;
- *w_gate*: the waiting gate used by writers, similar to the readers' gate.

Each of the `read_acquire`, `read_release`, `write_acquire`, and `write_release` methods must maintain this state. First they have to acquire the *mutex* (i.e., enter the main gate). After acquiring the *mutex*, `read_acquire` and `write_acquire` each must check to see if it has to wait. If so, it increments the count associated with its respective gate, opens a gate (using method `release_one`), and then blocks until its waiting gate opens up.

`release_one()` is the function that a thread uses when leaving the critical section. It must check to see if there is a waiting gate that has threads waiting behind it and whose condition is met. If so, it selects one and opens that gate. In the given code, `release_one()` first checks the readers' gate and then the writers' gate, but the other way around would have worked as well. If neither waiting gate qualifies, then `release_one()` has to open the main gate (i.e., release *mutex*).

Let us examine `read_acquire` more carefully. First the method acquires the *mutex*. Then, in the case that the thread finds that there is a writer in the critical section ($nwriters > 0$), it increments the counter associated with the readers' gate, leaves the critical section (`release_one`), and then tries to acquire the semaphore associated with the waiting gate. This will cause it to block until some thread opens that gate.

Now 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 `write_release`:

1. After acquiring *mutex*, the writer decrements *nwriters*, which must be 1 at this time, and thus becomes 0.
2. It then calls `release_one()`. `release_one()` finds that there are no writers in the critical section and there are two readers waiting. It therefore releases not *mutex* but the readers' gate's binary semaphore.
3. One of the waiting readers can now re-enter the critical section. When it does, the reader decrements the gate's counter (from 2 to 1) and increments *nreaders* (from 0 to 1). The reader finally calls `release_one()`.

4. Again, `release_one()` finds that there are no writers and that there are readers waiting, so again it releases the readers' semaphore.

5. The second reader can now enter the critical section. It decrements the gate's count from 1 to 0 and increments *nreaders* from 1 to 2.

6. Finally, the second reader calls `release_one()`. This time `release_one()` does not find any threads waiting, and so it releases *mutex*. There are now two threads that are holding the reader/writer lock.

Exercises

14.1 Several of the instantiations of `release_one()` in Figure 14.1 can be replaced by simply releasing *mutex*. Which ones?

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

14.3 Implement a solution to the producer/consumer problem using split binary semaphores.

14.4 Using busy waiting, implement a "bound lock" that allows up to `M` threads to acquire it at the same time.[1] 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 threads working at the same time might exhaust some resource such as a cache.)

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

14.6 Write a test program for bound locks that checks that up to `M` threads can acquire the bound lock at the same time.

14.7 Implement a thread-safe *GPU allocator* by modifying Figure 14.2. 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.

14.8 With reader/writer locks, concurrency can be improved if a thread *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 14.1. (Similarly, you may want to try to implement an *upgrade* of a read lock to a write lock. Why is this problematic?)

14.9 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 `OLBlock()` to create a new "one lane bridge" lock, `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.

[1] A bound lock is a restricted version of a *counting* semaphore.

```
1     const N = 10
2
3     availGPUs = {1..N}
4
5     def gpuAlloc():
6         result = choose(availGPUs)
7         availGPUs -= { result }
8
9     def gpuRelease(gpu):
10        availGPUs |= { gpu }
```

Figure 14.2: [code/gpu.hny] A thread-unsafe GPU allocator.

14.10 Extend the solution to Exercise 14.9 by implementing the requirement that at most n cars are allowed on the bridge. Add n as an argument to `OLBlock`.

Chapter 15

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 thread both acquiring a reader/writer lock. The *progress* property is an example of a *liveness property*—guaranteeing that something good eventually happens. Informally (and inexactly), progress states that if no threads are in the critical section, then some thread that wants to enter can.

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

While spinlocks suffer from starvation, it is a uniform random process and each thread 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 solution that we presented in Chapter 14. Consider this scenario: there are two readers and one writer. One reader is in the critical section while the writer is waiting. Now the second reader tries to enter and is able to. The first reader leaves. We are now in a similar situation as the initial state with one reader in the critical section and the writer waiting, but it is not the same reader. Unfortunately for the writer, this scenario can repeat itself indefinitely. So, even if neither reader was in the critical section all of the time, and the second reader arrived well after the writer, the writer never had a chance.

```
from synch import BinSema, acquire, release

def RWlock():
    result = {
            .nreaders: 0, .nwriters: 0, .mutex: BinSema(False),
            .r_gate: { .sema: BinSema(True), .count: 0 },
            .w_gate: { .sema: BinSema(True), .count: 0 }
        }

def read_acquire(rw):
    acquire(?rw→mutex)
    if (rw→nwriters > 0) or (rw→w_gate.count > 0):
        rw→r_gate.count += 1; release(?rw→mutex)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    if rw→r_gate.count > 0:
        release(?rw→r_gate.sema)
    else:
        release(?rw→mutex)

def read_release(rw):
    acquire(?rw→mutex)
    rw→nreaders −= 1
    if (rw→w_gate.count > 0) and (rw→nreaders == 0):
        release(?rw→w_gate.sema)
    else:
        release(?rw→mutex)

def write_acquire(rw):
    acquire(?rw→mutex)
    if (rw→nreaders + rw→nwriters) > 0:
        rw→w_gate.count += 1; release(?rw→mutex)
        acquire(?rw→w_gate.sema); rw→w_gate.count −= 1
    rw→nwriters += 1
    release(?rw→mutex)

def write_release(rw):
    acquire(?rw→mutex)
    rw→nwriters −= 1
    if rw→r_gate.count > 0:
        release(?rw→r_gate.sema)
    elif rw→w_gate.count > 0:
        release(?rw→w_gate.sema)
    else:
        release(?rw→mutex)
```

Figure 15.1: [code/RWfair.hny] Reader/Writer Lock SBS implementation addressing fairness.

SBSs allow much control over which type of thread runs next and is therefore a good starting point for developing fair synchronization algorithms. Figure 15.1 is based on Figure 14.1, but there are two important 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. Instead of a one-size-fits-all `release_one` method, each method has a custom way of selecting which gate to open. In particular, `read_release` prefers the write gate, while `write_release` prefers the read gate.

The net effect of this is that if there is contention between readers and writers, then readers and writers end up alternating entering the critical section. While readers can still starve other readers and writers can still starve other writers, readers can no longer starve writers nor vice versa. Other fairness is based on the fairness of semaphores themselves.

Exercises

15.1 Write a fair solution to the one-lane bridge problem of Exercise 14.9.

Chapter 16

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. A monitor also has special variables called *condition variables*, one per waiting condition. There are two operations on condition variables: `wait` and `signal`.

Harmony does not have language support for Hoare monitors but it has a module called `hoare`. Figure 16.1 shows its implementation. A monitor uses a hidden split binary semaphore. The mutex semaphore is acquired when entering a monitor and released upon exit. Each condition variable maintains a binary semaphore and a counter for the number of threads waiting on the condition. `wait` increments the condition's counter, releases the monitor mutex, blocks while trying to acquire the condition's semaphore, and upon resuming decrements the counter—in much the same way as we have seen for split binary semaphores (SBS). `signal` checks to see if the condition's count is non-zero, if so releases the condition's semaphore, and then blocks by trying to acquire the mutex again.

Figure 16.2 presents a bounded buffer implemented using Hoare monitors. It is written in much the same way you would if using the SBS technique (see Exercise 14.3). However, there is no `release_one` method. Instead, one can conclude that `put` guarantees that the queue will be non-empty, and `signal` will check if there are any threads waiting for this event. If so, `signal` will pass control to one such thread and, unlike `release_one`, re-enter the critical section afterwards by acquiring the *mutex*.

Implementing a reader/writer lock with Hoare monitors is not quite so straightforward, unfortunately. When a writer releases the lock, it has to choose whether to signal a reader or another writer. For that it needs to know if there is a reader or writer waiting. The simplest solution would be to peek at the counters inside the respective condition variables, but that breaks the abstraction. The alternative is for the reader/writer implementation to keep track of that state explicitly, which complicates the code. Also, it requires a deep understanding of the SBS method to remember to place a call to `signal` in the `read_acquire` method that releases additional readers that may be waiting to acquire the lock.

```
1   import synch
2
3   def Monitor():
4       result = synch.Lock()
5
6   def enter(mon):
7       synch.acquire(mon)
8
9   def exit(mon):
10      synch.release(mon)
11
12  def Condition():
13      result = { .sema: synch.BinSema(True), .count: 0 }
14
15  def wait(cond, mon):
16      cond→count += 1
17      exit(mon)
18      synch.acquire(?cond→sema)
19      cond→count -= 1
20
21  def signal(cond, mon):
22      if cond→count > 0:
23          synch.release(?cond→sema)
24          enter(mon)
```

Figure 16.1: [modules/hoare.hny] Implementation of Hoare monitors.

```
1   import hoare
2
3   def BB(size):
4       result = {
5               .mon: hoare.Monitor(),
6               .prod: hoare.Condition(), .cons: hoare.Condition(),
7               .buf: { x:() for x in {1..size} },
8               .head: 1, .tail: 1,
9               .count: 0, .size: size
10          }
11
12  def put(bb, item):
13      hoare.enter(?bb→mon)
14      if bb→count == bb→size:
15          hoare.wait(?bb→prod, ?bb→mon)
16      bb→buf[bb→tail] = item
17      bb→tail = (bb→tail % bb→size) + 1
18      bb→count += 1
19      hoare.signal(?bb→cons, ?bb→mon)
20      hoare.exit(?bb→mon)
21
22  def get(bb):
23      hoare.enter(?bb→mon)
24      if bb→count == 0:
25          hoare.wait(?bb→cons, ?bb→mon)
26      result = bb→buf[bb→head]
27      bb→head = (bb→head % bb→size) + 1
28      bb→count -= 1
29      hoare.signal(?bb→prod, ?bb→mon)
30      hoare.exit(?bb→mon)
```

Figure 16.2: [code/BBhoare.hny] Bounded Buffer implemented using a Hoare monitor.

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 and mostly easier to use in practice.

As in Hoare monitors, there is a hidden mutex associated with each Mesa monitor, and the mutex must be acquired upon entry to a method and released upon exit. Mesa monitors also have condition variables that a thread 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 thread p invokes `notify`, it does not immediately pass control to a thread that is waiting on the corresponding condition (if there is such a thread). Instead, p remains in the critical section until it leaves the monitor explicitly (by calling either `release` or `wait`). At that point, any thread that was notified will have a chance to enter the critical section, but they compete with other threads trying to enter the critical section.

Basically, there is just one gate to enter the critical section, instead of a main gate and a gate per waiting condition. This is a very important difference. In Hoare monitors, when a thread enters through a waiting gate, it can assume that the condition associated with the waiting gate still holds because no other thread can run in between. Not so with Mesa monitors: by the time a thread that was notified enters through the main gate, other threads may have entered first and falsified the condition. So, in Mesa, threads always have to check the condition again after resuming from the `wait` operation. This is accomplished by wrapping each `wait` operation in a **while** statement that loops until the condition of interest becomes valid. A Mesa monitor therefore is more closely related to busy waiting than to split binary semaphores.

Mesa monitors also allow notifying multiple threads. For example, a thread can invoke `notify` twice—if there are two or more threads waiting on the condition variable, two will be resumed. Operation `notifyAll` (aka `broadcast`) notifies *all* threads that are waiting on a condition. Signaling multiple threads is not possible with Hoare monitors because with Hoare monitors control must be passed immediately to a thread that has been signaled, and that can only be done if there is just one such thread.

The so-called "Mesa monitor semantics" or "Mesa condition variable semantics" have become more popular than Hoare monitor semantics and have been adopted by all major programming languages. That said, few programming languages provide full syntactical support for monitors, instead opting to support monitor semantics through library calls. 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 entirely through libraries.

Harmony provides support for Mesa monitors through the Harmony `synch` module. Figure 16.3 shows the implementation of condition variables in the `synch` module. `Condition`() creates a new condition variable. It is represented by a dictionary containing a bag of contexts of threads waiting on the condition variable. (The `synchS` library instead uses a list of contexts.)

```
1     def Condition():
2        result = bag.empty()
3
4     def wait(c, lk):
5        let blocked, cnt, ctx = True, 0, get_context():
6           atomic:
7              cnt = bag.count(!c, ctx)
8              bag.add(c, ctx)
9              !lk = False
10          while blocked:
11             atomic:
12                if (not !lk) and (bag.count(!c, ctx) <= cnt):
13                   !lk = True
14                   blocked = False
15
16    def notify(c):
17       atomic:
18          if !c != bag.empty():
19             bag.remove(c, bag.bchoose(!c))
20
21    def notifyAll(c):
22       !c = bag.empty()
```

Figure 16.3: [modules/synch.hny] Implementation of condition variables in the `synch` module.

```
from synch import *

def RWlock():
    result = {
            .nreaders: 0, .nwriters: 0, .mutex: Lock(),
            .r_cond: Condition(), .w_cond: Condition()
        }

def read_acquire(rw):
    acquire(?rw→mutex)
    while rw→nwriters > 0:
        wait(?rw→r_cond, ?rw→mutex)
    rw→nreaders += 1
    release(?rw→mutex)

def read_release(rw):
    acquire(?rw→mutex)
    rw→nreaders -= 1
    if rw→nreaders == 0:
        notify(?rw→w_cond)
    release(?rw→mutex)

def write_acquire(rw):
    acquire(?rw→mutex)
    while (rw→nreaders + rw→nwriters) > 0:
        wait(?rw→w_cond, ?rw→mutex)
    rw→nwriters = 1
    release(?rw→mutex)

def write_release(rw):
    acquire(?rw→mutex)
    rw→nwriters = 0
    notifyAll(?rw→r_cond)
    notify(?rw→w_cond)
    release(?rw→mutex)
```

Figure 16.4: [code/RWcv.hny] Reader/Writer Lock using Mesa-style condition variables.

`wait` adds the context of the thread to the bag. This increments the number of threads in the bag with the same context. `wait` then loops 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 threads with that context to resume and re-acquire the lock associated with the monitor. `notifyAll` empties out the entire bag, allowing all threads in the bag to resume.

To illustrate how Mesa condition variables are used in practice, we demonstrate using an implementation of reader/writer locks. Figure 16.4 shows the code. *mutex* is the shared lock that protects the critical region. There are two condition variables: readers wait on *r_cond* and writers wait on *w_cond*. The implementation also keeps track of the number of readers and writers in the critical section.

Note that `wait` is always invoked within a **while** loop that checks for the condition that the thread is waiting for. It is *imperative* that there is always a **while** loop around any invocation of `wait` containing the negation of the condition that the thread is waiting for. Many implementation of Mesa 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. As a rule of thumb, one should always be able to replace `wait` by `unlock` followed by `lock`. This turns the solution into a busy-waiting one, inefficient but still correct.

In `read_release`, notice that `notify`(?*w_cond*) is invoked when there are no readers left, *without* checking if there are writers waiting to enter. This is ok, because calling `notify` is a no-op if no thread is waiting.

`write_release` executes `notifyAll`(?*r_cond*) as well as `notify`(?*w_cond)*. Because we do not keep track of the number of waiting readers or writers, we have to conservatively assume that all waiting readers can enter, or, alternatively, up to one waiting writer can enter. So `write_release` 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 14.1), with Mesa monitors all readers are awakened 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 threads are waiting, either all the readers get to enter next or one of the writers gets to enter next. (If you want to prevent waking up both readers and a writer, then you can keep track of how many threads are waiting in the code.)

When using Mesa condition variable you have to be careful to invoke **notify** or **notifyAll** in the right places. 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 threads than too few. In case of doubt, use `notifyAll`. Waking up too many threads 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 each and every corner case. You can try to replace each `notifyAll` with `notify` and see if every possible execution of the application still terminates.

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

Exercises

16.1 Implement a solution to the bounded buffer problem using Mesa condition variables.

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

1. *tl* = `TryLock`() # *create a try lock*
2. `acquire`(?*tl*) # *acquire a try lock*
3. `tryAcquire`(?*tl*) # *attempt to acquire a try lock*
4. `release`(?*tl*) # *release a try lock*

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

16.3 Write a new version of the GPU allocator in Exercise 14.7 using Mesa condition variables. In this version, a thread 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. It returns a set of n GPU identifiers. Method `gpuReleaseSet(s)` takes a set of GPU identifiers as argument. A thread does not have to return all the GPUs it allocated at once. (You may want to try implementing this with Split Binary Semaphores. It is not as easy.)

16.4 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.

16.5 Bonus problem: Figure 16.5 shows an iterative implementation of the Qsort algorithm, and Figure 16.6 an accompanying test program. The array to be sorted is stored in shared variable *textqs*.`arr`. Another shared variable, *testqs*.`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 threads" that should replace the `qsort` code. Each worker loops until *todo* is empty and sorts the ranges that it finds until then. The `main` thread needs to wait until all workers are done.

```
def Qsort(arr):
    result = { .arr: arr, .todo: { (0, len(arr) - 1) } }

def swap(p, q):                 # swap !p and !q
    !p, !q = !q, !p;

def partition(qs, lo, hi):
    result = lo
    for i in {lo..hi - 1}:
        if qs→arr[i] <= qs→arr[hi]:
            swap(?qs→arr[result], ?qs→arr[i])
            result += 1
    swap(?qs→arr[result], ?qs→arr[hi]);

def sortrange(qs, range):
    let lo, hi = range let pivot = partition(qs, lo, hi):
        if (pivot - 1) > lo:
            qs→todo |= { (lo, pivot - 1) }
        if (pivot + 1) < hi:
            qs→todo |= { (pivot + 1, hi) }

def sort(qs):
    while qs→todo != {}:
        let range = choose(qs→todo):
            qs→todo -= { range }
            sortrange(qs, range)
    result = qs→arr
```

Figure 16.5: [code/qsort.hny] Iterative qsort() implementation.

```
import qsort, bag

const NITEMS = 4

a = [ choose({1..NITEMS}) for i in {1..choose({1..NITEMS})} ]
testqs = qsort.Qsort(a)
sa = qsort.sort(?testqs)
assert all(sa[i - 1] <= sa[i] for i in {1..len(sa)-1}) # sorted?
assert bag.fromList(a) == bag.fromList(sa); # is it a permutation?
```

Figure 16.6: [code/qsorttest.hny] Test program for Figure 16.5.

Chapter 17

Deadlock

When multiple threads are synchronizing access to shared resources, they may end up in a *deadlock* situation where one or more of the threads 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 every 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 repeats this procedure. Dijkstra had them repeating this for ever, but for the purposes of this book, philosophers can leave the table when they are not using any forks.

Figure 17.1 implements the dining philosophers in Harmony, using a thread for each philosopher and a lock for each fork. If you run it, Harmony complains that the execution may not be able to terminate, with all five threads 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 thread at a time:
2. *Hold and Wait*: each thread holds resources it already allocated while it waits for other resources that it needs;
3. *No Preemption*: resources cannot be forcibly taken away from threads that allocated them;
4. *Circular Wait*: there exists a directed circular chain of threads, 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 in general (although, for some resources it is possible, as demonstrated in Chapter 21). Havender proposed the following techniques that avoid the remaining three conditions [Hav68]:

```
from synch import Lock, acquire, release

const N = 5

forks = [Lock(),] * N

def diner(which):
   let left, right = (which, (which + 1) % N):
      while choose({ False, True }):
         acquire(?forks[left])
         acquire(?forks[right])
         # dine
         release(?forks[left])
         release(?forks[right])
         # think

for i in {0..N–1}:
   spawn diner(i)
```

Figure 17.1: [code/Diners.hny] Dining Philosophers.

- *No Hold and Wait*: a thread must request all resources it is going to need at the same time;
- *Preemption*: if a thread 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 17.2 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:

```
1   import synch
2
3   const N = 5
4
5   mutex = synch.Lock()
6   forks = [False,] * N
7   conds = [synch.Condition(),] * N
8
9   def diner(which):
10      let left, right = (which, (which + 1) % N):
11          while choose({ False, True }):
12              synch.acquire(?mutex)
13              while forks[left] or forks[right]:
14                  if forks[left]:
15                      synch.wait(?conds[left], ?mutex)
16                  if forks[right]:
17                      synch.wait(?conds[right], ?mutex)
18              assert not (forks[left] or forks[right])
19              forks[left] = forks[right] = True
20              synch.release(?mutex)
21              # dine
22              synch.acquire(?mutex)
23              forks[left] = forks[right] = False
24              synch.notify(?conds[left]);
25              synch.notify(?conds[right])
26              synch.release(?mutex)
27              # think
28
29  for i in {0..N–1}:
30      spawn diner(i)
```

Figure 17.2: [code/DinersCV.hny] Dining Philosophers that grab both forks at the same time.

```
    while forks[left]:
        synch.wait(?conds[left], ?mutex)
    while forks[right]:
        synch.wait(?conds[right], ?mutex)
```

- 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 threads to back out. While this could be done, this invariably leads to a busy waiting solution where a thread 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 thread waiting for the next thread 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:
        synch.acquire(?forks[left])
        synch.acquire(?forks[right])
    else:
        synch.acquire(?forks[right])
        synch.acquire(?forks[left])
```

or like so:

```
    synch.acquire(?forks[min(left, right)])
    synch.acquire(?forks[max(left, right])
```

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. 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 17.3 demonstrates this concept. It uses a *counting semaphore* to restrict the number of diners at any time to four. A counting semaphore is like a binary semaphore, but can be acquired a given number of times. It is supported by the `synch` module. The `P` or "procure" operation acquires a counting semaphore. That is, it tries to decrement the semaphore, blocking while the semaphore has a value of 0. The `V` or "vacate" operation increments the semaphore.

This avoidance technique 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

```
from synch import *

const N = 5

forks = [Lock(),] * N
sema = Semaphore(N - 1)     # can be procured up to N-1 times

def diner(which):
   let left, right = (which, (which + 1) % N):
      while choose({ False, True }):
         P(?sema)              # procure counting semaphore
         acquire(?forks[left])
         acquire(?forks[right])
         # dine
         release(?forks[left])
         release(?forks[right])
         V(?sema)              # vacate counting semaphore
         # think

for i in {0..N-1}:
   spawn diner(i)
```

Figure 17.3: [code/DinersAvoid.hny] Dining Philosophers that carefully avoid getting into a deadlock scenario.

is that one needs to know ahead of time the set of threads and what the maximum number of resources is that each thread wants to allocate, making them generally quite impractical.

Exercises

17.1 The solution in Figure 17.2 can be simplified by, instead of having a condition variable per fork, having a condition variable per diner. It is the same number of condition variables, but you will not need to have **if** statements nested inside the **while** loop waiting for the forks. See if you can figure it out.

17.2 Figure 17.4 shows an implementation of a bank with various accounts and transfers between those accounts. Unfortunately, running the test reveals that it sometimes leaves unterminated threads. Can you fix the problem?

17.3 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.

17.4 Add an invariant that checks that the total of the balances never changes. Note that the invariant only holds if none of the locks are held.

```
from synch import Lock, acquire, release

const MAX_BALANCE = 2
const N_ACCOUNTS = 2
const N_THREADS = 2

accounts = [ { .lock: Lock(), .balance: choose({0..MAX_BALANCE})}
                        for i in {1..N_ACCOUNTS} ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
        result = True
    else:
        result = False
    release(?accounts[a1].lock)

def thread():
    let a1 = choose({0..N_ACCOUNTS-1})
    let a2 = choose({0..N_ACCOUNTS-1} - { a1 }):
        transfer(a1, a2, choose({1..MAX_BALANCE}))

for i in {1..N_THREADS}:
    spawn thread()
```

Figure 17.4: [code/bank.hny] Bank accounts.

Chapter 18

Actors and Message Passing

Some programming languages favor a different way of implementing synchronization using so-called *actors* [HBS73, Agh86]. Actors are threads that have only private memory and communicate through *message passing*. See Figure 18.1 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 thread owns a particular piece of data and other threads access it by sending request messages to the thread and optionally waiting for response messages. Each thread handles one message at a time, serializing all access to the data it owns. As message queues are FIFO (First-In-First-Out), 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:

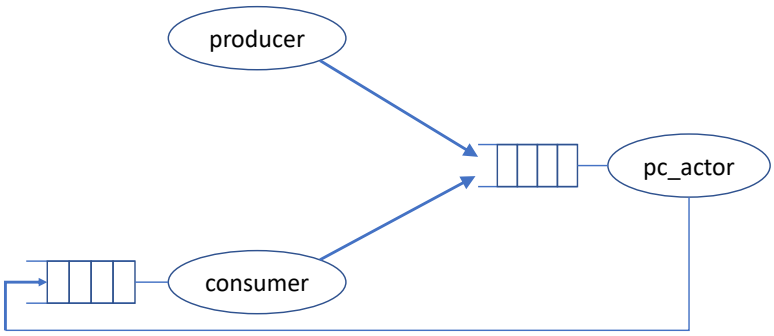


Figure 18.1: Depiction of three actors. The producer does not receive messages.

```
import synch

const NCLIENTS = 3

server_queue = synch.Queue()

def server():
   let counter = 0:
      while counter < NCLIENTS:
         let q = synch.get(?server_queue):   # await request
            synch.put(q, counter)            # send response
            counter += 1

spawn server()

sequential done
done = [False,] * NCLIENTS

def client(client_queue):
   synch.put(?server_queue, client_queue)      # send request
   let response = synch.get(client_queue):     # await response
      done[response] = True
   await all(done)
   assert done == ([True,] * NCLIENTS)

alice_queue = synch.Queue()
spawn client(?alice_queue)
bob_queue = synch.Queue()
spawn client(?bob_queue)
charlie_queue = synch.Queue()
spawn client(?charlie_queue)
```

Figure 18.2: [code/counter.hny] An illustration of the actor approach.

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

For those familiar with counting semaphores: note that a `Queue` behaves much like a zero-initialized counting semaphore. `put` is much like `V`, except that it is accompanied by data. `get` is much like `P`, except that it also returns data. Thus, synchronized queues can be considered a generalization of counting semaphores.

Figure 18.2 illustrates the actor approach. There are three client threads that each want to be assigned a unique identifier from the set $\{0, 1, 2\}$. Normally one would use a shared 0-initialized counter and a lock. Each client would acquire the lock, get the value of the counter and increment it, and release the lock. Instead, in the actor approach the counter is managed by a separate server thread. Each client sends a request to the server, consisting in this case of simply the queue to which the server must send the response. The server maintains a local, zero-initialized counter variable. Upon receiving a request, it returns a response with the value of the counter and increments the counter. No lock is required. The code is tested using the *done* array.

This illustration is an example of the *client/server* model. Here a single actor implements some service, and clients send request messages and receive response messages. The model is particularly popular in distributed systems, where each actor runs on a separate machine and the queues are message channels. For example, the server can be a web server, and its clients are web browsers.

Exercises

18.1 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 19

Barrier Synchronization

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

Blocking queues work well for implementing barrier synchronization. Figure 19.1 shows an example. There is a queue for each of the `N` threads. Before thread i enters a round, it first sends a message to every other thread and then waits until it receives a message from every other thread. In this simple case, each message contains `None`, but in practice useful information may be exchanged between the threads.

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

More generally, barrier synchronization can be abstracted as follows. We want to create a `Barrier`(n) object, with operations `enter()` and `exit()`. It is helpful to use a roller coaster car with n seats as a metaphor:

- the car cannot contain more than n people;
- the car won't take off until n people are in the car;
- no new people can enter the car until all n people have left it.

Notice there are two different waiting conditions:

1. waiting for the car to empty out;
2. waiting for the car to fill up.

```
1   from synch import Queue, put, get
2
3   const NTHREADS = 3
4   const NROUNDS = 4
5
6   sequential round
7   round = [0,] * NTHREADS
8
9   q = [Queue(),] * NTHREADS
10
11  def thread(self):
12      for r in {1..NROUNDS}:
13          for i in {0..NTHREADS–1} where i != self:
14              put(?q[i], None)
15          for i in {0..NTHREADS–1} where i != self:
16              get(?q[self])
17          round[self] += 1
18          assert (max(round) – min(round)) <= 1
19
20  for i in {0..NTHREADS–1}:
21      spawn thread(i)
```

Figure 19.1: [code/qbarrier.hny] Barrier synchronization with queues.

```
from synch import *

def Barrier(limit):
    result = {
        .limit: limit, .stage: 0, .mutex: Lock(),
        .empty: Condition(), .full: Condition()
    }

def enter(b):
    acquire(?b→mutex)
    while b→stage >= b→limit:     # wait for car to empty out
        wait(?b→empty, ?b→mutex)
    b→stage += 1
    if b→stage < b→limit:         # wait for car to fill up
        while b→stage < b→limit:
            wait(?b→full, ?b→mutex)
    else:
        notifyAll(?b→full)        # car is full and ready to go
    release(?b→mutex)

def exit(b):
    acquire(?b→mutex)
    assert b→limit <= b→stage < (2 * b→limit)
    b→stage += 1
    if b→stage == (2 * b→limit):  # everybody left
        b→stage = 0
        notifyAll(?b→empty)       # let next group in
    release(?b→mutex)
```

Figure 19.2: [code/barrier.hny] Generalized barrier synchronization.

```
1   import barrier
2
3   const NROUNDS = 3
4   const NTHREADS = 3
5
6   barr = barrier.Barrier(NTHREADS)
7
8   sequential round
9   round = [None,] * NTHREADS
10
11  def thread(self):
12      for r in {0..NROUNDS–1}:
13          barrier.enter(?barr)
14          round[self] = r
15          assert { x for x in round where x != None } == { r }
16          round[self] = None
17          barrier.exit(?barr)
18
19  for i in {0..NTHREADS–1}:
20      spawn thread(i)
```

Figure 19.3: [code/barriertest.hny] Test program for Figure 19.2.

But this poses a complication. Suppose, for example, that there are two seats in the car, and there is one person in the car. Does that mean that the car is not yet full, or not yet empty? We have to distinguish those situations. To this end, it is useful to think of the car as going through $2 \cdot n$ stages:

Stage 0: the car is empty;

Stage $1...n-1$: the car is filling up;

Stage n: the car is full;

Stage $n+1...2n-1$: the car is emptying out.

Figure 19.2 shows code that implements barrier synchronization using these stages. Method `enter`() has two `wait` loops, one for each waiting condition. This is sometimes called *double turnstile*. Each loop uses a different condition variable. The first loop waits until the car has emptied out, while the second waits for the car to fill up. Figure 19.3 is a test program. The threads check that all threads within the barrier are in the same round.

Exercises

19.1 Implement barrier synchronization for `N` threads with just three binary semaphores. Busy waiting is not allowed. Can you implement barrier synchronization with two binary semaphores? (As always, the Little Book of Semaphores [Dow09] is a good resource for solving synchronization problems with semaphores. Look for the *double turnstile* solution.)

19.2 Imagine a pool hall with `N` tables. A table is *full* from the time there are two players until both players have left. When someone arrives, they can join a table that is not full, preferably one that has a player ready to start playing. Implement a simulation of such a pool hall.

Chapter 20

Interrupts

Threads 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 thread 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. Thus you can think of **trap** as setting a timer. 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*() thread 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 3.2, except that only a single thread 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 threads. If you run the code through Harmony, you will find that the code may not terminate. The issue is that a thread 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 thread 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 thread to `True` using the **setintlevel** interface. Figure 20.4 illustrates how this is done. Note that it is not necessary to change the

```
sequential done

count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    await done
    assert count == 1

spawn main()
```

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

```
sequential done

count = 0
done = False

def handler():
    count += 1
    done = True

def main():
    trap handler()
    count += 1
    await done
    assert count == 2

spawn main()
```

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

```
from synch import Lock, acquire, release

sequential done

countlock = Lock()
count = 0
done = False

def handler():
    acquire(?countlock)
    count += 1
    release(?countlock)
    done = True

def main():
    trap handler()
    acquire(?countlock)
    count += 1
    release(?countlock)
    await done
    assert count == 2

spawn main()
```

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

```
1   sequential done
2
3   count = 0
4   done = False
5
6   def handler():
7       count += 1
8       done = True
9
10  def main():
11      trap handler()
12      setintlevel(True)
13      count += 1
14      setintlevel(False)
15      await done
16      assert count == 2
17
18  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 thread 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 threads getting blocked indefinitely.

(Another option is to use synchronization techniques that do not use locks. See Chapter 21 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. It is ok to invoke non-blocking synchronization calls such as `notify`, but calls such as `acquire` and

```
1   sequential done
2
3   count = 0
4   done = False
5
6   def increment():
7       let prior = setintlevel(True):
8           count += 1
9           setintlevel(prior)
10
11  def handler():
12      increment()
13      done = True
14
15  def main():
16      trap handler()
17      increment()
18      await done
19      assert count == 2
20
21  spawn main()
```

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

```
from synch import Lock, acquire, release

sequential done

count = 0
countlock = Lock()
done = [ False, False ]

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await all(done)
    assert count == 4, count

spawn thread(0)
spawn thread(1)
```

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

`wait` should 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 `put` method you implemented in Exercise 16.1 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. Implement a new method `i_put`(*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

Non-Blocking Synchronization

So far, we have concentrated on critical sections to synchronize multiple threads. Certainly, preventing multiple threads 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 thread is slow for some reason while being in the critical section, the other threads 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 be too slow. In this chapter, we will have a look at how one can develop concurrent code in which threads do not have to wait for other threads (or interrupt handlers) to complete their ongoing operations.

As an example, we will revisit the producer/consumer problem. The code in Figure 21.1 is based on code developed by Herlihy and Wing [HW87]. The code is a "proof of existence" for non-blocking synchronization; it is not necessarily practical. There are two variables. *items* is an unbounded array with each entry initialized to `None`. *back* is an index into the array and points to the next slot where a new value is inserted. The code uses two atomic operations:

- `inc`(p): atomically increments `!`p and returns the old value;
- `exch`(p): sets `!`p to `None` 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 threads attempt to extract an item from the same slot, at most one will succeed.

There is no critical section. If one thread is executing instructions very slowly, this does not negatively impact the other threads, 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` atomic operations are not universally available on existing processors.

```
1   const MAX_ITEMS = 3
2
3   sequential back, items
4   back = 0
5   items = [None,] * MAX_ITEMS
6
7   def inc(pcnt):
8     atomic:
9       result = !pcnt
10      !pcnt += 1
11
12  def exch(pv):
13    atomic:
14      result = !pv
15      !pv = None
16
17  def produce(item):
18    items[inc(?back)] = item
19
20  def consume():
21    result = None
22    while result == None:
23      let range, i = back, 0:
24        while (i < range) and (result == None):
25          result = exch(?items[i])
26          i += 1
27
28  for i in {1..MAX_ITEMS}:
29    spawn produce(i)
30  for i in {1..choose({0..MAX_ITEMS})}:
31    spawn consume()
```

Figure 21.1: [code/hw.hny] Non-blocking queue.

However, in the literature you can find examples of practical non-blocking (aka *wait-free*) synchronization algorithms.

Exercises

21.1 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 17.2, 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.

Chapter 22

Alternating Bit Protocol

A *distributed system* is a concurrent system in which a collection of threads 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 threads 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 threads, 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

```
sequential s_chan, r_chan

s_chan = r_chan = ()
s_seq = r_seq = 0

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 = { .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, { .ack: m.seq }, reliable)
               if m.seq == r_seq:
                  result = m.payload
                  blocked = False
```

Figure 22.1: [code/abp.hny] Alternating Bit Protocol.

```
import abp

const NMSGS = 5

def sender():
   for i in {1..NMSGS}:
      abp.app_send(i)

def receiver():
   for i in {1..NMSGS}:
      let payload = abp.app_recv(i == NMSGS):
         assert payload == i

spawn sender()
spawn receiver()
```

Figure 22.2: [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 22.2 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

22.1 Chapter 18 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.

22.2 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.

Bibliography

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems (doctoral dissertation)*. MIT Press, Cambridge, MA, USA, 1986.

[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.

[BNS69] László A. Bélády, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, June 1969.

[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.

[Cor69] Fernando J. Corbató. A paging experiment with the Multics system. In *In Honor of Philip M. Morse*, pages 217–228, 1969.

[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.

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

[Hol11] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[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.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[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

Value Types

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	`{ .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 E) are ordered first by name, then by program counter, then by the remaining content.

Generic operators on Harmony values include:

e `==` e	equivalence
e `!=` e	inequivalence
e `<` e, e `<=` e, e `>` e, e `>=` e	comparison

A.1 Boolean

The Boolean type has only two possible values: `False` and `True`. Unlike in Python, in Harmony booleans are distinct from integers, and in particular `False` < 0. In statements and expressions where booleans are expected, it is not possible to substibute values of other types.

Operations on booleans include:

e `and` e `and ...`	conjuction
e `or` e `or ...`	disjunction
e `=>` e, e `not =>` e	implication
`not` e	negation
v `if` e `else` v'	v or v' depending on e
`any` s, `all` s	disjunction / conjunction for set or list s

A.2 Integer

The integer type supports any whole number, as well as $-$`inf` and `inf`. In the Python-based model checker, integers are infinite precision. In the C-based model checker, integers are implemented by two's complement 61-bit words, and $-$`inf` is represented by the minimal integer and `inf` is represented by the maximal integer.

Operations on integers include:

`-`e	negation
`abs(`e`)`	absolute value
e `+` e `+ ...`	sum
e `-` e	difference
e `*` e `*` e `...`	product
e `/` e`,` e `//` e	integer division
e `%` e`,` e `mod` e	integer division remainder
e `**` e	power
`~`e	binary inversion
e `&` e `& ...`	binary and
e `|` e `| ...`	binary or
e `^` e `^ ...`	binary exclusive or
e `<<` e	binary shift left
e `>>` e	binary shift right
$\{e..e'\}$	set of integers from e to e' inclusive

A.3 Atom

Atoms are essentially names and consist of one or more unicode characters. If they do not contain special characters and do not start with a digit, then an atom can be represented by a "." followed by the characters. For example, `.hello` is a representation of the atom "hello". A special character can be represented by `.0xXX`, where `XX` is the hexidecimal unicode for the character. Atoms should not be confused with strings. There are no special operations on atoms. (In the future, operators may be introduced that convert between strings and atoms.)

A.4 Set

In Harmony you can create a set of any collection of Harmony values. Its syntax is { $v_0, v_1, ...$ }. Python users: note that in Harmony the empty set is denoted as {}.

The `set` module (Section C.5) contains various convenient routines that operate on sets. Native operations on sets include:

`len` s	cardinality
s `-` s	set difference
s `&` s `& ...`	intersection
s `|` s `| ...`	union
s `^` s `^ ...`	inclusion/exclusion (elements in odd number of sets)
`choose` s	select an element (Harmony will try all)
`min` s	minimum element
`max` s	maximum element
`any` s	`True` if any value is `True`
`all` s	`True` if all values are `True`

Harmony also supports *set comprehension*. In its simplest form, { `f(`v`)` `for` v `in` s } returns a set that is constructed by applying `f` to all elements in s (where s is a set or a list). This is known

as *mapping*. But set comprehension is much more powerful and can include joining multiple sets (using nested for loops) and filtering (using the `where` keyword).

For example: { x `+` y `for` x `in` s `for` y `in` s `such that` $(x * y) == 4$ } returns a set that is constructed by summing pairs of elements from s that, when multiplied, have the value 4.

A.5 Dictionary

A dictionary maps a set of values (known as *keys*) to another set of values. The generic syntax of a dictionary is $\{k_0 : v_0, k_1 : v_1, ...\}$. Different from Python, the empty dictionary is either () or [] (because {} is the empty set in Harmony). If there are duplicate keys in the list, then only the one with the maximum value survives. Therefore the order of the keys in the dictionary does not matter.

Dictionaries support comprehension. The basic form is: { `f(`k`):g(`k`) for` k `in` s }.

There are various special cases of dictionaries, including lists, tuples, strings, and bags (multi-sets) that are individually described below.

Operations on dictionaries include the following:

d k	indexing
`len` d	the number of keys in d
`keys` d	the set of keys in d
v `[not] in keys` d	check if v is a *key* in d
v `[not] in` d	check if v is a *value* in d
`min` d	the minimum value in d
`max` d	the maximum value in d
`any` d	`True` if any value is `True`
`all` d	`True` if all values are `True`
d `&` d `& ...`	dictionary intersection
d `|` d `| ...`	dictionary union

Because in Harmony brackets are used for parsing purposes only, you can write $d[k]$ (or $d(k)$) instead of d k. However, if k is an atom, like `.id`, then you might prefer the notation k`.id`.

Python users beware: the Harmony v `in` d operator checks if there is some key k such that $d[k] = v$. In Python, the same syntax checks if v is a key in d. The difference exists because in Harmony a list is a special case of a dictionary.

Dictionary intersection and dictionary union are defined so that they work well with bags. With disjoint dictionaries, intersection and union work as expected. If there is a key in the intersection, then dictionary intersection retains the minimum value while dictionary union retains the maximum value. Unlike Python, Harmony dictionary intersection and union are commutative and associative.

A.6 List or Tuple

In Harmony, there is no distinction between a list or a tuple. Both are a special case of dictionary. In particular, a list of n values is represented by a dictionary that maps the integers $0, ..., n-1$ to those values. Hence, if t is a list or tuple, then the notation $t[0]$ returns the first element of the list.

You can denote a list by a sequence of values, each value terminated by a comma. As per usual, you can use brackets or parentheses at your discretion. For Python users, the important thing to note is that a singleton list in Harmony must contain a comma. For example [1,] is a list containing the value 1, while [1] is simply the value 1.

The `list` module (Section C.4) contains various convenient routines that operate on lists or tuples. Native operations on lists or tuples include the following:

t k	indexing
t `+` *t* `+ ...`	concatenation
t `*` *n*	*n* copies of *t* concatenated
v `[not] in` *t*	check if *v* is a *value* in *t*
`len` *t*	the length of *t*
`min` *t*	the minimum value in *t*
`max` *t*	the maximum value in *t*
`any` *t*	`True` if any value is `True`
`all` *t*	`True` if all values are `True`
t `&` *t* `& ...`	pairwise list minimum
t `|` *t* `| ...`	pairwise list maximum

Lists and tuples support comprehension as well. In their most basic form: `f(`*v*`) for` *v* `in` *s*. For example, to check if any element in a list *t* is even, you can write:

`any(`$(x\%2) == 0$ `for` *x* `in` *t*`)`.

A.7 String

In Harmony, a string is a list of single-character atoms. The Python string notations mostly work in Harmony as well. For example, `'abc'` is a three-character string consisting of the atoms `.a`, `.b`, and `.c`. `"abc"` is the same three-character string. Various special characters (including quotes, newlines, etc.) can be *escaped* using a backslash. Multi-line strings can be terminated by triple quotes or triple double-quotes.

Native operations on strings include the following:

s k	indexing
s `+` *s* `+ ...`	concatenation
s `*` *n*	*n* copies of *s* concatenated
c `[not] in` *s*	check if *c* (a one-character atom) is in *s*
`len` *s*	the length of *s*

A.8 Bag or Multiset

A bag is represented by a dictionary that maps each element to its multiplicity, for example: { 10:2, 12:1 }. The `bag` module (Section C.2) contains various convenient routines that operate on lists or tuples. Native operations on bags include the following:

v `[not] in keys` *b*	check if *v* is in *b*
t `&` *t* `& ...`	bag intersection
t `|` *t* `| ...`	bag union

A.9 Program Counter

A program counter is an integer that can be used to index into Harmony bytecode. When you define a method, a lambda function, or a label, you are creating a constant of the program counter type.

Operations on program counters include the following:

m a	invoke method with program counter m and argument a
`atLabel` l	return a bag of (method, argument) pairs of threads executing at label l

You can create lambda functions similarly to Python. For example: `lambda`(x, y)`:`$x + y$.

A.10 Address

In Harmony, each shared variable has an address, which is essentially a list of Harmony values. For example, the address of variable `d.v[3]` consists of the list `.d`, `.v`, and 3. The only way to construct an address in Harmony is using the `?` operator. `?d.v[3]` returns the address of variable `d.v[3]`. An address can be dereferenced to return the value of the variable. If a is an address, then !a returns the value. Like C, Harmony supports the shorthand a`->v` for the expression `(!`a`).v`.

A.11 Context

A context value captures the state of a thread. A context is itself composed over various Harmony values. Some of these values can be extracted. In particular, if c is a context, then:

`c.name`	returns the name of the thread's main method
`c.entry`	returns the program counter of the thread's main method
`c.arg`	returns the argument value of the thread's main method
`c.this`	returns the thread-local state
`c.mode`	returns the *mode* of the context

The possible modes of a context are as follows:

`normal`	context of a executing thread
`stopped`	context of a stopped thread
`terminated`	context of a terminated thread
`failed`	context of a failed thread

A thread can call `get_context()` to retrieve its current context. Also, the function `contexts()` returns a bag of all contexts (including terminated ones). Both these functions should only be called in `atomic` mode and used for checking invariants.

Appendix B

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`
`assert` *b* `[,` *e*`]`	*b* is a boolean. Optionally report value of expression *e*
`atomic:` `S`	`S` a list of statements
`await` *b*	*b* is a boolean
`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
`del` *lv*	delete
`for` *a* `in` *e*`:` `S`	*a* is a bounded variable, *e* is a set, `S` a list of statements
`from m import ...`	`m` identifies a module
`go` *c e*	*c* is a context, *e* is an expression
`if` *b*`:` `S else:` `S`	*b* is a boolean, `S` a list of statements
`import m, ...`	`m` identifies a module
`let` *a* `=` *e* `[let ...]:` `S`	*a* is a bounded variable, *e* is an expression, `S` a list of statements
`pass`	do nothing
`sequential v, ...`	`v` has sequential consistency
`spawn m` *e* `[,` *t*`]`	`m` is a method, *e* is an expression, *t* is the initial thread-local state
`trap m` *e*	`m` is a method and *e* is an expression
`while` *b*`:` `S`	*b* is a boolean, `S` a list of statements

Appendix C

List of Modules

C.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.

C.2 The `bag` module

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

`empty()`	returns an empty bag
`fromSet(`s`)`	create a bag from set s
`fromList(`t`)`	convert list t into a bag
`count(`b`, `e`)`	count how many times e occurs in bag b
`bchoose(`b`)`	like `choose(s)`, but applied to a bag
`add(`pb`, `e`)`	add one copy of e to bag `!`pb
`remove(`pb`, `e`)`	remove one copy of e from bag `!`pb

C.3 The `hoare` module

The `hoare` module implements support for Hoare-style monitors and condition variables.

`Monitor()`	return a monitor mutex
`enter(`m`)`	enter a monitor. m points to a monitor mutex
`exit(`m`)`	exit a monitor.
`Condition()`	return a condition variable
`wait(`c`, `m`)`	wait on condition variable pointed to by c in monitor pointed to by m
`signal(`c`, `m`)`	signal a condition variable

C.4 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
`qsort(`t`)`	sort list t using quicksort

C.5 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.

C.6 The `synch` module

The `synch` module provides the following methods:

`tas`(*lk*)	test-and-set on `!`*lk*
`cas`(*ptr*, *old*, *new*)	compare-and-swap on `!`*ptr*
`BinSem`(*v*)	return a binary semaphore initialized to *v*
`Lock`()	return a binary semaphore initialized to `False`
`acquire`(*bs*)	acquire binary semaphore `!`*bs*
`release`(*bs*)	release binary semaphore `!`*bs*
`Condition`()	return a condition variable
`wait`(*c*, *lk*)	wait on condition variable `!`*c* and lock *lk*
`notify`(*c*)	notify a thread waiting on condition variable `!`*c*
`notifyAll`(*c*)	notify all threads waiting on condition variable `!`*c*
`Semaphore`(*cnt*)	return a counting semaphore initialized to *cnt*
`P`(*sema*)	procure `!`*sema*
`V`(*sema*)	vacate `!`*sema*
`Queue`()	return a synchronized queue object
`get`(*q*)	return next element of *q*, blocking if empty
`put`(*q*, *item*)	add *item* to *a*

Appendix D

List of Machine Instructions

Address	compute address from two components
Apply	pop m and i and apply i to m, pushing a value
Assert, Assert2	pop b and check that it is `True`. Assert2 also pops value to print
AtomicInc/Dec	increment/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	cut a set into its smallest element and the remainder
Del [v]	delete shared variable v
DelVar [v]	delete thread variable v
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
IncVar v	increment thread variable v
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 thread 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
ReadonlyInc/Dec	increment/decrement the read-only counter of this context
Return	pop return address, push `result`, and restore program counter
Sequential	pop an address of a variable that has sequential consistency
SetIntLevel	pop e, set interrupt level to e, and push old interrupt level
Spawn	pop initial thread-local state, 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 thread 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 thread variables, both shared and thread 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 thread register (*i.e.*, the values of the thread variables) onto the stack. It initializes the `result` variable to the empty dictionary. The `Return` instruction restores the thread 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 thread variables by the given names.
- Every `Stop` instruction must immediately be followed by a `Continue` instruction.

Appendix E

Contexts and Threads

A context captures the state of a thread. Each time the thread 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 threads with the same state at the same time. A context consists of the following:

name	the name of the main method that the thread is executing
argument	the argument given to the main method
program counter	an integer value pointing into the code
frame pointer	an integer value pointing into the stack
atomic	if non-zero, the thread is in atomic mode
readonly	if non-zero, the thread is in read-only mode
stack	a list of Harmony values
method variables	a dictionary mapping atoms (names of method variables) to values
thread-local variables	a dictionary mapping atoms (names of thread-local variables) to values
stopped	a boolean indicating if the context is stopped
failure	if not None, string that describes how the thread failed

Details:

- 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 `thread`), and the return address (in case of `normal`).
- A thread terminates when it reaches the `Return` instruction of the top-level method (when the stack frame is of type `thread`) or when it hits an exception. Exceptions include divide by zero, reading a non-existent key in a dictionary, accessing a non-existent variable, as well as when an assertion fails;
- The execution of a thread in *atomic mode* does not get interleaved with that of other threads.
- The execution of a thread in *read-only mode* is not allowed to update shared variables of spawn threads.

- The register of a thread always contains a dictionary, mapping atoms to arbitrary values. The atoms correspond to the variable names in a Harmony program.

Appendix F

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 runnable contexts
stopbag	a bag of stopped contexts
choosing	if not `None`, indicates a context that is choosing

There is initially a single context with name `__init__/()` and program counter 0. It starts executing in atomic mode until it finishes executing the last `Return` instruction. Other threads, created through `spawn` statements, do not start executing until then.

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

A *stride* can involve multiple steps. The following instructions start a new stride: `Load`, `Store`, `AtomicInc`, and `Continue`. The HVM interleaves stides, not steps. Like steps, each stride involves a single context. Unlike a step, a stride can leave the state unchanged (because its steps lead back to where the stride started).

Executing a Harmony program results in a graph where the nodes are Harmony states and the edges are strides. 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 strides by the same thread are called a *turn*. Each state maintains the shortest path to it from the initial state in terms of turns. The diameter of the graph is the length of the longest path found in terms of turns.

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 threads. 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 G

Harmony Language Details

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

G.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 = y = [ 1, 2 ]
2    x[0] = 3
3    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 2, 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:

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

That would allow you to simulate object methods.

There are at least two reasons why Harmony is not object-oriented. First, object-orientation often adds layers of indirection that would make it harder to model check and also to interpret the results. Consider, for example, a lock. In Python, a lock is an object. A lock variable would contain a reference to a lock object. In Harmony, a lock variable contains the value of the lock itself. Thus, the following statement means something quite different in Python and Harmony:

```
x = y = Lock();
```

In Python, this creates two variables x and y referring to the same lock. In Harmony, the two variables will be two different locks. If you want two variables referring to the same lock in Harmony, you would write:

```
x = y = malloc(Lock());
```

The second reason for Harmony not being object-oriented is that many concurrency solutions in the literature are expressed in C or some other low-level language that does not support object-orientation, but instead use `malloc` and `free`.

G.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 threads. 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 threads are spawned.

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

G.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 $a\ b\ c\ ....$ 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 have the next highest precedence, and the remaining operators have the lowest precedence. For example, $-2 + 3$ evaluates to 1, not -5.

Associative operators ($+$, $*$, $|$, &, $\hat{}$, **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`. A sequence of comparison operations, such as $a < b < c$, is evaluated left to right but stops as soon as one of the comparisons fails.

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`. Similarly, **not** => is non-implication.

G.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. For example, the statement x `= 1,;` assigns a singleton tuple (or list) to x.

Because tuples and lists are dictionaries, the `del` statement is different from Python. For example, if x = [.a, .b, .c], then `del` x[1] results in x = { 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 C.4).

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.

```
1   from stack import Stack, push, pop
2
3   teststack = Stack()
4   push(?teststack, 1)
5   push(?teststack, 2)
6   v = pop(?teststack)
7   assert v == 2
8   push(?teststack, 3)
9   v = pop(?teststack)
10  assert v == 3
11  v = pop(?teststack)
12  assert v == 1
```

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

G.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 C.4) provides methods `values`(), `items`(), `enumerate`(), and `reversed`() for other types of iteration supported by Python.

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

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

Harmony also supports set, list, and dictionary comprehensions. Comprehensions are similar to Python, except that filtering uses the keyword `where` instead of `if`.

G.6 Dynamic Allocation

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

```
def Stack():
    result = [ ]

def push(st, v):
    (!st)[len(!st)] = v

def pop(st):
    let n = len(!st) – 1:
        result = (!st)[n]
        del (!st)[n]
```

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

```
import list

def Stack():
    result = [ ]

def push(st, v):
    !st += [v,]

def pop(st):
    let n = len(!st) – 1:
        result = (!st)[n]
        !st = list.subseq(!st, 0, n)
```

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

```
def Stack():
    result = ()

def push(st, v):
    (!st) = (v, !st)

def pop(st):
    let (top, rest) = !st:
        result = top
        !st = rest
```

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

```
1   from alloc import malloc, free
2
3   def Stack():
4       result = None
5
6   def push(st, v):
7       !st = malloc({ .value: v, .rest: !st })
8
9   def pop(st):
10      let node = !st:
11          result = node→value
12          !st = node→rest
13          free(node)
```

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

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

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

Figure G.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 G.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`.

G.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 thank 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.

The following people contributed by making comments on or finding bugs in early drafts of the book: Alex Chang, Anneke van Renesse, Brendon Nguyen, Hartek Sabharwal, Heather Zheng, Jack Rehmann, Jacob Brugh, Liam Arzola, Lorenzo Alvisi, Maria Martucci, Phillip O'Reggio, Saleh Hassen, Sunwook Kim, Terryn Jung, Trishita Tiwari, William Ma, Xiangyu Zhang, Yidan Wang, Zhuoyu Xu, and Zoltan Csaki.

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

Glossary

actor model is a concurrency model where there are no shared variables, only threads with private variables that communicate through message passing. 103

atomic instruction a machine instruction that may involve multiple memory load and/or store operations and is executed atomically. 44

atomicity describes that a certain machine instruction or sequence of machine instructions is executed indivisibly by a thread and cannot be interleaved with machine instructions of another thread. 5

barrier synchronization is when a set of threads execute in rounds, waiting for one another to complete each round. 106

blocked thread is a thread that cannot change the state or terminate or can only do so after another thread changes the state first. For example, a thread that is waiting for a lock to become available. 49

busy waiting (aka spin-waiting) is when a thread waits in a loop for some application-defined condition instead of blocking. 77

concurrent execution (aka parallel execution) is when there are multiple threads executing and their machine instructions are interleaved in an unpredictable manner. 5

condition variable a variable that keeps track of which threads are waiting for a specific application-level condition. The variable can be waited on as well as signaled or notified. 88

conditional critical section is a critical section with, besides mutual exclusion, additional conditions on when a thread is allowed to enter the critical section. 80

context (aka continuation) describes the state of a running thread, including its program counter, the values of its variables (stored in its register), and the contents of its stack. 16

critical section (aka critical region) is a set of instructions that only one thread is allowed to execute at a time. The instructions are, however, not executed atomically, as other threads can continue to execute and access shared variables. 23

data race is when there are two or more threads concurrently accessing a shared variable, at least one of which is an update to the variable. 46

deadlock is when there are two or more threads waiting indefinitely for one another to release a resource. 97

determinism is when the outcome of an execution is uniquely determined by the initial state. 5

fairness is when each thread eventually can access each resource it needs to access with high probability. 85

invariant is a binary predicate over states that must hold for every reachable state of a thread. 6

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. 59

lock an object that can be owned by at most one thread at a time. Useful for implementing mutual exclusion. 25

machine instruction is an atomic operation on the Harmony virtual machine, executed by a thread. 11

model checking is a formal verification method that explores all possible executions of a program, which must have a finite number of states. 5

monitor is a programming language paradigm that supports mutual exclusion as well as waiting for resources to become available. 88

mutual exclusion is the property that two threads never enter the same critical section. 23

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 threads are not making progress. 118

producer/consumer problem is a synchronization problem whereby one or more producing threads submit items and one or more consuming threads 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. 79

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 threads being able to enter the critical section if they want to. 85

race condition describes when multiple threads access shared state concurrently, leading to undesirable outcomes. 12

reader/writer lock is a lock on a resource that can be held by multiple threads if they all only read the resource. 77

sequential consistency is a consistency model in which shared memory accesses are executed in an order consistent with the program order. 30

sequential execution is when there is just one thread executing, as opposed to concurrent execution. 5

shared variable is a variable that is stored in the memory of the Harmony virtual machine and shared between multiple threads, as opposed to a thread variable. 5

spinlock is an implementation of a lock whereby a thread loops until the lock is available, at which point the thread atomically obtains the lock. 42

stack machine is a model of computing where the state of a thread is kept on a stack. Harmony uses a combination of a stack machine and a register-based machine. 18

starvation is when a thread cannot make progress because it is continuously losing a competition with other threads to get access to a resource. 85

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. 30

step is the execution of a machine instruction by a thread, updating its state.. 30

thread is code in execution. We do not make the distinction between threads and threads. A thread has a current context and updates its context every time it executes a machine instruction. 10

thread safety is when the implementation of a data structure allows concurrent access with well-defined semantics. 23

thread variable is a variable that is private to a single thread and stored in its register. 30

trace is a sequence of steps, starting from an initial state. An infinite trace is also called a *behavior*. 30