

Concurrent Programming with Harmony



Robbert van Renesse

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

Contents

1	On Concurrent Programming	9
2	Hello World!	13
3	The Problems with Concurrent Programming	19
4	The Harmony Virtual Machine	26
5	Critical Sections	33
6	Harmony Methods and Pointers	43
7	Specifying a Lock	48
8	Lock Implementations	51
9	Concurrent Data Structures	61
10	Testing: Checking Behaviors	70
11	Debugging	78
12	Conditional Waiting	84
12.1	Reader/Writer Locks	84
12.2	Bounded Buffer	87
13	Condition Variables	91
14	Starvation	97
15	Deadlock	101
16	Actors and Message Passing	107
17	Barrier Synchronization	110
18	Advanced Barrier Synchronization	117

19 Example: A Concurrent File Service	120
20 Interrupts	132
21 Non-Blocking Synchronization	139
22 Alternating Bit Protocol	142
23 Leader Election	146
24 Transactions and Two Phase Commit	149
25 Chain Replication	154
26 Working with Actions	160
27 Replicated Atomic Read/Write Register	164
28 Distributed Consensus	169
29 Paxos	176
30 Needham-Schroeder Authentication Protocol	180
Bibliography	183
A Harmony Language Reference	186
A.1 Value Types and Operators	186
A.2 Statements	194
A.3 Harmony is not object-oriented	200
A.4 Constants, Global and Local Variables	201
A.5 Operator Precedence	202
A.6 Tuples, Lists, and Pattern Matching	202
A.7 Dynamic Allocation	203
A.8 Comments	205
A.9 Type Checking	205
B Modules	207
B.1 The <code>action</code> module	207
B.2 The <code>alloc</code> module	208
B.3 The <code>bags</code> module	208
B.4 The <code>hoare</code> module	208
B.5 The <code>lists</code> module	208
B.6 The <code>sets</code> module	209
B.7 The <code>synch</code> module	209
B.8 The <code>thread</code> module	210

C The Harmony Virtual Machine	211
C.1 Machine Instructions	212
C.2 Addresses and Method Calls	214
C.3 Contexts and Threads	214
C.4 Formal Specification	215
D How Harmony Works	216
D.1 Compiler	216
D.2 Model Checker	217
D.3 Automata Conversion	218
D.4 Model Checker Output Analysis	218
E Simplified Grammar	220
F Directly checking linearizability	223
G Manual Pages	227
H Peterson’s Algorithm	230
I Split Binary Semaphores	235
Acknowledgments	242
Index	243
Glossary	245

List of Figures

2.1	[code/hello1.hny]	Hello World!	13
2.2	[code/hello3.hny]	Harmony program with two possible outputs	14
2.3	[code/hello4.hny]	Harmony program with an infinite number of possible outputs	14
2.4		Demonstrating Harmony methods and threads	15
2.5	[code/hello7.hny]	Various interleavings of threads	16
2.6	[code/hello8.hny]	Making groups of operations atomic reduces interleaving	16
2.7	[code/triangle.hny]	Computing triangle numbers	17
2.8		Running the code in Figure 2.7	17
2.9		Running the code in Figure 2.7 for N = 100	18
3.1		The problem with non-determinism	20
3.2		The problem with non-atomicity	21
3.3	[code/prog5.hny]	What actually happens in Figure 3.2(b)	22
3.4	[python/inc.py]	Python implementation of Figure 3.2(b)	24
3.5	[python/incmany.py]	Using Python to increment N times	25
4.1		The first part of the HVM bytecode corresponding to Figure 3.2(b)	28
4.2		The HTML output of running Harmony on Figure 3.2(b)	30
4.3	[code/prog4enter.hny]	Incorrect attempt at fixing the code of Figure 3.2(b)	32
5.1	[code/csbarebones.hny]	Modeling a critical section	33
5.2	[code/cs.hny]	Harmony model of a critical section	34
5.3	[code/naiveLock.hny]	Naïve implementation of a shared lock and the markdown output of running Harmony	36
5.4	[code/naiveFlags.hny]	Naïve use of flags to solve mutual exclusion	37
5.5	[code/naiveTurn.hny]	Naïve use of turn variable to solve mutual exclusion	38
5.6	[code/Peterson.hny]	Peterson's Algorithm	39
5.7	[code/cstonebit.hny]	Mutual exclusion using a flag per thread	42
6.1	[code/PetersonMethod.hny]	Peterson's Algorithm accessed through methods	44
6.2	[code/hanoi.hny]	Nondeterministic Towers of Hanoi	45
6.3	[code/hanoi2.hny]	Recursive Towers of Hanoi	45
6.4	[code/clock.hny]	Harmony program that finds page replacement anomalies	46
7.1	[code/lock.hny]	Specification of a lock	49
7.2	[code/lock_demo.hny]	Using a lock to implement a critical section	49

7.3	[code/prog4lock.hny] Figure 3.2(b) fixed with a lock	50
8.1	[code/spinlock.hny] Mutual Exclusion using a “spinlock” based on atomic swap	52
8.2	[code/lock_tas.hny] Implementation of the lock specification in Figure 7.1 using a spinlock based on test-and-set	54
8.3	[code/lock_ticket.hny] Implementation of the lock specification in Figure 7.1 using a ticket lock	55
8.4	[code/lock_test1.hny] A test program for locks (based on Figure 5.2)	56
8.5	[modules/lock_susp.hny] Lock implementation using suspension	57
8.6	[code/xy.hny] Incomplete code for Exercise 8.2 with desired invariant $x + y = 100$	59
8.7	[code/atm.hny] Withdrawing money from an ATM	60
9.1	A sequential and a concurrent specification of a queue	62
9.2	[code/queue_test1.hny] Using a concurrent queue	62
9.3	[code/queue_lock.hny] An implementation of a concurrent queue data structure and a depiction of a queue with three elements	64
9.4	[code/queue_MS.hny] A queue with separate locks for enqueueing and dequeuing items and a depiction of a queue with two elements	65
9.5	[code/setobj.hny] Specification of a concurrent set object	66
9.6	[code/setobj_test1.hny] Test code for set objects	67
9.7	[code/setobj_linkedlist.hny] Implementation of a set of values using a linked list with fine-grained locking	68
10.1	[code/queue_test_seq.hny] Sequential queue test	71
10.2	[code/queue_btest2.hny] Concurrent queue test. The behavior DFA is for $N_PUT = N_GET = 1$.	73
10.3	[python/queue_btest2.py] Python implementation of Figure 10.2	74
10.4	[code/queue_broken1.hny] A correct sequential but not a correct concurrent queue implementation	75
11.1	[code/queue_broken2.hny] A buggy queue implementation	79
11.2	Running Figure 10.2 against Figure 11.1	80
11.3	HTML output of Figure 11.2 but for $N_PUT=2$ and $N_GET=1$	81
11.4	[code/queue_fix.hny] Queue implementation with hand-over-hand locking	83
12.1	[code/rwlock.hny] Specification of reader/writer locks	85
12.2	[code/rwlock_test1.hny] Test code for reader/writer locks	86
12.3	[code/rwlock_cheat.hny] “Cheating” reader/writer lock	87
12.4	[code/rwlock_btest.hny] A behavioral test of reader/writer locks	88
12.5	[code/rwlock_busy.hny] Busy waiting reader/writer lock	89
12.6	[code/boundedbuffer.hny] Bounded buffer specification	90
13.1	[code/rwlock_cv.hny] Reader/Writer Lock using Mesa-style condition variables	92
13.2	[modules/synch.hny] Implementation of condition variables in the <code>synch</code> module	94
13.3	[code/gpu.hny] A thread-unsafe GPU allocator	95
13.4	[code/qsorth.hny] Iterative <code>qsorth()</code> implementation	96
13.5	[code/qsorthtest.hny] Test program for Figure 13.4	96

14.1	[code/rwlock_cv_fair.hny]	Reader/Writer Lock implementation addressing fairness (part 1)	98
14.2	[code/rwlock_cv_fair.hny]	Reader/Writer Lock implementation addressing fairness (part 2)	99
15.1	[code/Diners.hny]	Dining Philosophers	102
15.2	[code/DinersCV.hny]	Dining Philosophers that grab both forks at the same time	103
15.3	[code/DinersAvoid.hny]	Dining Philosophers solutions that avoids getting into a deadlock by allowing at most $N-1$ philosophers to start eating at a time	105
15.4	[code/bank.hny]	Bank accounts	106
16.1		Depiction of three actors. The producer does not receive messages.	107
16.2	[code/counter.hny]	An illustration of the actor approach	108
17.1	[code/barrier_demo.hny]	Parallel dot product using barrier implementation	111
17.2	[code/barrier_test.hny]	Testing the barrier synchronization interface	112
17.3	[code/barrier_once.hny]	A single-use barrier specification	112
17.4	[code/barrier_broken.hny]	An incorrect barrier specification	113
17.5	[code/barrier_double.hny]	Barrier specification using the double turnstile approach	113
17.6	[code/barrier.hny]	Barrier specification	114
17.7	[code/barrier_cv.hny]	Barrier implementation	115
17.8	[code/bsort.hny]	Parallel bubble sort	116
18.1	[code/rollercoaster.hny]	Modeling a roller coaster	118
19.1	[code/file.hny]	Specification of the file system	121
19.2	[code/file_btest.hny]	Test program for a concurrent file system	122
19.3	[code/disk.hny]	Specification of a disk	123
19.4		The file system data structure: (a) disk layout (1 superblock, n blocks, m inode blocks, 4 inodes per block); (b) inode for a file with 3 data blocks	124
19.5	[code/file_inode.hny]	File system implementation preamble	125
19.6	[code/file_inode.hny]	File system interface implementation	126
19.7	[code/file_inode.hny]	File server and worker threads	127
19.8	[code/file_inode.hny]	File system initialization	127
19.9	[code/file_inode.hny]	File system utility functions	128
19.10	[code/file_inode.hny]	Handling of file read requests	129
19.11	[code/file_inode.hny]	Handling of file write and delete requests	130
20.1	[code/trap.hny]	How to use trap	133
20.2	[code/trap2.hny]	A race condition with interrupts	133
20.3	[code/trap3.hny]	Locks do not work with interrupts	134
20.4	[code/trap4.hny]	Disabling and enabling interrupts	135
20.5	[code/trap5.hny]	Example of an interrupt-safe method	136
20.6	[code/trap6.hny]	Code that is both interrupt-safe and thread-safe	137
21.1	[code/hw.hny]	Non-blocking queue	140

22.1	[code/abp.hny]	Alternating Bit Protocol	143
22.2	[code/abptest.hny]	Test code for alternating bit protocol	144
23.1	[code/leader.hny]	A leader election protocol on a ring	147
24.1	[code/2pc.hny]	Two Phase Commit protocol: code for banks	150
24.2	[code/2pc.hny]	Two Phase Commit protocol: code for transaction coordinators	151
25.1	[code/rsm.hny]	Replicated State Machine	155
25.2		The DFA generated by Figure 25.1 when <code>NOPS=2</code> and <code>NREPLICAS=2</code>	156
25.3	[code/chain.hny]	Chain Replication (part 1)	157
25.4	[code/chain.hny]	Chain Replication (part 2)	158
26.1	[code/chainaction.hny]	Chain Replication specification using actions (part 1)	161
26.2	[code/chainaction.hny]	Chain Replication specification using actions (part 2)	162
27.1	[code/register.hny]	An atomic read/write register	165
27.2	[code/abctest.hny]	Behavioral test for atomic read/write registers and the output for the case that <code>NREADERS = NWRITERS = 1</code>	166
27.3	[code/abd.hny]	An implementation of a replicated atomic read/write register	167
28.1	[code/consensus.hny]	Distributed consensus code and behavior DFA	170
28.2	[code/bosco.hny]	A crash-tolerant consensus protocol	172
28.3		The behavior DFA for Figure 28.2	173
28.4	[code/bosco2.hny]	Reducing the state space	175
29.1	[code/paxos.hny]	A version of the Paxos protocol, Part 1	177
29.2	[code/paxos.hny]	A version of the Paxos protocol, Part 2	178
30.1	[code/needhamschroeder.hny]	Needham-Schroeder protocol and an attack	181
A.1		Using save and go to implement fork()	193
A.2	[code/stacktest.hny]	Testing a stack implementation.	203
A.3	[code/stack1.hny]	Stack implemented using a dynamically updated list.	204
A.4	[code/stack2.hny]	Stack implemented using static lists.	204
A.5	[code/stack3.hny]	Stack implemented using a recursive tuple data structure.	204
A.6	[code/stack4.hny]	Stack implemented using a linked list.	205
F.1	[code/queuelin.hny]	Queue implementation with linearization points	224
F.2	[code/qtestconc.hny]	Concurrent queue test	225
H.1		Venn diagram classifying all states and a trace	231
I.1	[code/rwlock_sbs.hny]	Reader/Writer Lock using Split Binary Semaphores	236
I.2	[modules/hoare.hny]	Implementation of Hoare monitors	238
I.3	[code/boundedbuffer_hoare.hny]	Bounded Buffer implemented using a Hoare monitor	239

Chapter 1

On Concurrent Programming

Programming with concurrency is hard. 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 usually *deterministic*: If you run the program twice with the same input, the same output will be produced. Bugs are reproducible and thus easy to track down, for example by instrumenting the program. They are called *Bohrbugs*. However, 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. We call these *Heisenbugs*—overhead caused by instrumentation leads to timing changes that can make such bugs less likely to cause havoc.
- 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 the incorrect execution is hard to reproduce.

This book is intended to help people with understanding and developing concurrent code, which includes programs for distributed systems. In particular, the book comes with a tool called Harmony that helps with *testing* concurrent code. The approach is based on *model checking* [CES86]: 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 and will try to minimize the 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 the model checker may 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, 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 will be triggered and the model checker will produce a scenario that simplifies finding 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.) The model checker would not find a bug in this adversarial program.

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 and specification language with a model checker. Harmony was designed to teach the basics of concurrent and distributed 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 the Harmony programming language are 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], Gerard Holzmann’s work on Promela and SPIN [Hol11], and University of Washington’s DSLabs system [MWA⁺19]. Some of the examples in this book are derived from those sources. Har-

mony is designed to have a lower learning curve than those systems, but may not be as powerful for specific problems. 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.
- [Chapter 3](#) illustrates the problem of concurrent programming through a simple example in which two threads are concurrently incrementing a shared variable.
- [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. The chapter also introduces *Peterson’s Algorithm*, an elegant (although not very efficient or practical) solution to implementing a critical section.
- [Chapter 6](#) gives some more details on the Harmony language needed for the rest of the book.
- [Chapter 7](#) talks about how Harmony can be used as a specification language. The chapter introduces how to specify atomic constructs such as locks.
- [Chapter 8](#) looks at various ways in which a lock specification can be implemented.
- [Chapter 9](#) gives an introduction to building concurrent data structures.
- [Chapter 10](#) discusses approaches to testing concurrent code in Harmony.
- [Chapter 11](#) instead goes into how to find a bug in concurrent code using the Harmony output.
- [Chapter 12](#) talks about threads having to wait for certain conditions. As examples of conditional waiting, the chapter presents the reader/writer lock problem and the bounded buffer problem.
- [Chapter 13](#) presents *condition variables*, a synchronization construct to simplify efficient conditional waiting.
- [Chapter 14](#) talks about *starvation*: The problem that in some synchronization approaches threads may not be able to get access to a resource they need.
- [Chapter 15](#) describes *deadlock* where a set of threads are indefinitely waiting for one another to release a resource.
- [Chapter 16](#) presents the *actor model* and *message passing* as an approach to synchronization.
- [Chapter 17](#) describes *barrier synchronization*, useful in high-performance computing applications such as parallel simulations.
- [Chapter 18](#) delves deeper into barrier synchronization for situations when there are more threads than the barrier requires.

- [Chapter 19](#) presents a concurrent file system as a larger example of a concurrent program.
- [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 23](#) presents a protocol for electing a leader on a ring of processors, where each processor is uniquely identified and only knows its successor on the ring.
- [Chapter 24](#) describes atomic database transactions and the two-phase commit protocol used to implement them.
- [Chapter 25](#) describes *state machine replication* and the *chain replication* protocol to support replication.
- [Chapter 26](#) describes an alternative way to write concurrent and distributed specifications in Harmony, using chain replication as an example.
- [Chapter 27](#) presents a protocol for a fault-tolerant replicated object that supports only read and write operations.
- [Chapter 28](#) demonstrates a fault-tolerant distributed consensus algorithm (a.k.a. protocol) expressed in Harmony.
- [Chapter 29](#) shows how one can specify and check the well-known Paxos consensus protocol.
- [Chapter 30](#) demonstrates using Harmony to find a (known) bug in the original Needham-Schroeder authentication protocol.

If you already know about concurrent and distributed programming and are just interested in a “speed course” on Harmony, I would recommend reading [Chapter 2](#), [Chapter 4](#), [Chapter 6](#), [Chapter 7](#), and [Chapter 9](#). The appendices contain various details about Harmony itself, including an appendix on convenient Harmony modules ([Appendix B](#)), and an appendix that explains how Harmony works ([Appendix D](#)).

Chapter 2

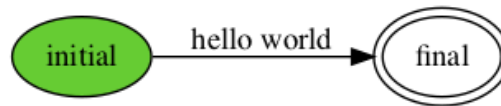
Hello World!

The first programming book that I read cover to cover was *The C Programming Language* (first edition) by Brian W. Kernighan and Dennis M. Ritchie, which was around 1980. I did not know at the time that 10 years later Dennis, the designer of the C programming language, would be my boss at AT&T Bell Labs in Murray Hill, NJ, while Brian would be my colleague in the same lab. The first C program in the book printed the string “hello, world”. Since then, most programming tutorials for pretty much any programming language start with that example.

Harmony, too, has a Hello World program. Figure 2.1 shows the program and the corresponding output. After installation (see <https://harmony.cs.cornell.edu>), you can run Harmony as follows from the command line:

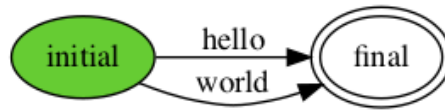
```
$ harmony code/hello1.hny
```

Try it out (here \$ represents a shell prompt). For this to work, make sure **harmony** is in your command shell’s search path. The code for examples in this book can be found in the **code** folder under the name listed in the caption of the example. If you need to, you can download the sources separately from <https://harmony.cs.cornell.edu/sources.zip>. In this case, the file **code/hello1.hny** contains the code in Figure 2.1. The output is a *Deterministic State Machine* (DFA). The green circle represents the initial state and the double circle represents the final state. There is one *transition*, labeled with the string “hello world”. The DFA describes (or *recognizes*) all possible outputs that the program can generate. In this case, there is only one.



```
1  print "hello world"
```

Figure 2.1: [\[code/hello1.hny\]](#) Hello World!

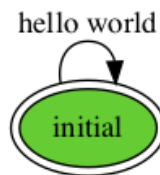


```

1  print choose { "hello", "world" }

```

Figure 2.2: [\[code/hello3.hny\]](#) Harmony program with two possible outputs



```

1  while choose { False, True }:
2    print "hello world"

```

Figure 2.3: [\[code/hello4.hny\]](#) Harmony program with an infinite number of possible outputs

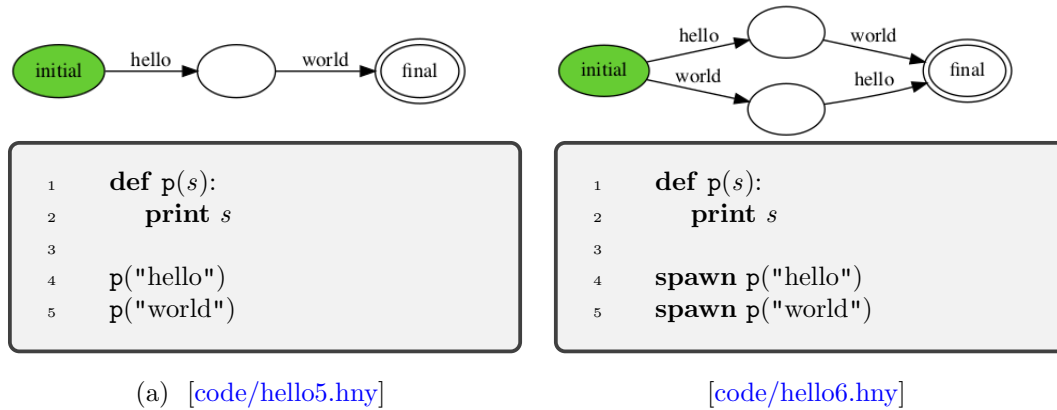


Figure 2.4: Demonstrating Harmony methods and threads

But programs can usually have more than one execution and produce multiple different outputs as a result. This is usually as a result of different inputs, but Harmony programs do not have inputs. Instead, [Figure 2.2](#) demonstrates *nondeterministic choice* in Harmony programs. In this case, the program chooses to print either “hello” or “world”. The corresponding DFA captures both possibilities. You can think of the **choose** operator as enumerating all possible inputs to the program.

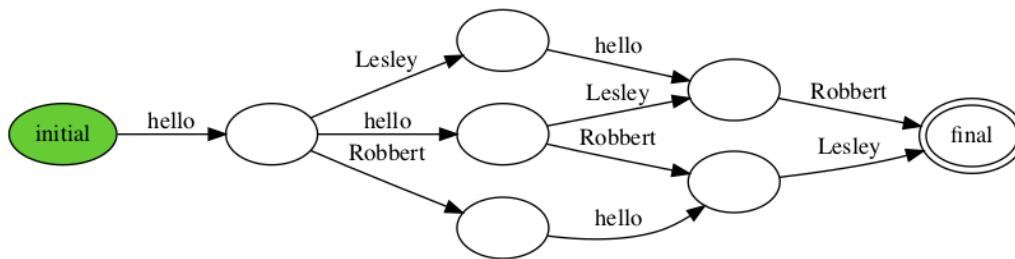
[Figure 2.3](#) shows a program that has an infinite number of possible outputs by using a loop with a non-deterministic stopping condition. Harmony usually requires that any program must be able to terminate, so the loop is conditioned on a nondeterministic choice between **False** and **True**. The possible outputs consist of zero or more copies of the string “hello world”. Note that this single state DFA (where the initial state and the final state happen to be the same) captures an infinite number of possible executions of the program.

[Figure 2.4](#) demonstrates *methods* and *threads* in Harmony. In [Figure 2.4\(a\)](#), the code simply prints the strings “hello” and “world”, in that order. Notice that this leads to an intermediate state after “hello” is printed but before “world” is. However, there is still only one execution possible. [Figure 2.4\(b\)](#) shows two threads, one printing “hello” and one printing “world”. Because the threads run concurrently, the program can either output “hello world” or “world hello”. Printing in Harmony is atomic, so “hweolrlld” is not a possible output.

[Figure 2.5](#) shows two threads, one printing the strings “hello” and “Robbert”, while the other prints “hello” and “Lesley”. Now there are four possible outputs depending on how the two threads are interleaved, including “hello hello Lesley Robbert”. This is probably not what the programmer wanted. [Figure 2.6](#) shows another important feature of Harmony: *atomic blocks*. The program is similar to [Figure 2.5](#), but the programmer specified that the two print statements in a thread should be executed as an atomic unit. As a result, there are only two interleavings possible.

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

- Harmony does not support floating point;

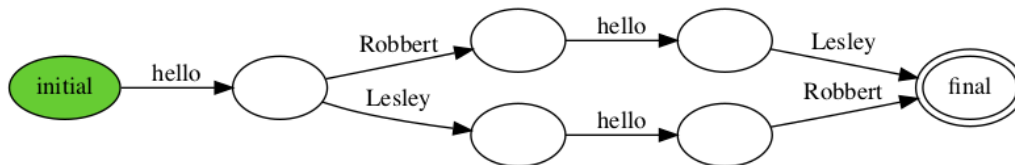


```

1  def hello(name):
2      print "hello"
3      print name
4
5  spawn hello("Lesley")
6  spawn hello("Robbert")

```

Figure 2.5: [\[code/hello7.hny\]](#) Various interleavings of threads



```

1  def hello(name):
2      atomically:
3          print "hello"
4          print name
5
6  spawn hello("Lesley")
7  spawn hello("Robbert")

```

Figure 2.6: [\[code/hello8.hny\]](#) Making groups of operations atomic reduces interleaving

```

1  const N = 10
2
3  def triangle(n) returns result:  # 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 2.7: [[code/triangle.hny](#)] Computing triangle numbers

```

$ harmony --noweb code/triangle.hny

```

- Phase 1: compile Harmony program to bytecode
- Phase 2: run the model checker (nworkers = 6)
 - 13 states (time 0.00s, mem=0.000GB)
 - 12/12 computations/edges
 - 2 rounds, 98 values
- Phase 3: Scan states
- Phase 4: Further analysis
 - **No issues found**
- Phase 4a: convert to DFA
- Phase 5: write results to code/triangle.hco
- Phase 6: loading code/triangle.hco

open file://.../code/triangle.htm for detailed information

Figure 2.8: Running the code in [Figure 2.7](#)

- Python is object-oriented, supporting classes with methods and inheritance; Harmony has objects but does not support classes. Harmony supports pointers, allowing construction of complicated data structures.
- In Python, lists, dictionaries, and sets are (mutable) objects. In Harmony, they are (immutable) values.

There are also less important differences that you will discover as you get more familiar with programming in Harmony.

[Figure 2.7](#) shows another example of a Harmony program. The example is a sequential program and has a method `triangle` that takes an integer number as argument. The method declares a variable called `result` that eventually contains the result of the method (there is no `return` statement in Harmony). The method also has a bound variable called `n` containing the value of the argument. The `{ x..y }` notation represents a set containing the numbers from `x` to `y` (inclusive). (Harmony

```
$ harmony -c N=100 -noweb code/triangle.hny
```

- Phase 1: compile Harmony program to bytecode
- Phase 2: run the model checker (nworkers = 6)
 - 103 states (time 0.00s, mem=0.000GB)
 - 102/102 computations/edges
 - 2 rounds, 728 values
- Phase 3: Scan states
- Phase 4: Further analysis
 - **No issues found**
- Phase 4a: convert to DFA
- Phase 5: write results to code/triangle.hco
- Phase 6: loading code/triangle.hco

open file://.../code/triangle.htm for detailed information

Figure 2.9: Running the code in [Figure 2.7](#) for $N = 100$

does not have a **range** operator like Python.) 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 ([Figure 2.8](#)) will try all possible executions, which includes all possible values for x . The results from running the Harmony model checker are output in **Markdown** format. Markdown is a lightweight markup language for creating formatted text. Usually Harmony also pops up a web browser window to show the DFA and detailed results—the **--noweb** flag tells Harmony not to do that.

The **assert** statement checks that the output is correct. If the program is correct, Harmony reports 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 specified value, but those can be overridden on the command line using the **-c** option. [Figure 2.9](#) shows how to test the code for $N = 100$.

Exercises

2.1 Write a Harmony program that uses **choose** instead of **spawn** to create the same output DFA as [Figure 2.4\(b\)](#).

2.2 Add the line **print**(x , **triangle**(x)) to the end of the program and create an output png file. Before you look at it, what do you think the output should look like?

2.3 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](#).)

2.4 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 Problems with Concurrent Programming

Concurrent programming, a.k.a. multithreaded programming, involves multiple threads running in parallel while sharing variables. As mentioned before, there are two problems with concurrent programming: non-determinism and non-atomicity.

We shall first consider non-determinism. [Figure 3.1](#) shows two programs. Program (a) is sequential. It sets x to **True**, asserts that $x = \mathbf{True}$ and finally sets x 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. This problem is an example of non-determinism: Methods `f()` and `g()` can run in either order. In one order, the assertion fails, while in the other it does not. But since Harmony checks all possible executions, it will always find the problematic one.

[Figure 3.1\(c\)](#) shows the output of running [Figure 3.1\(b\)](#) through Harmony. Underneath the line, there is a summary of what happened in one of its executions. First, the initialization thread (T0) runs and sets the global variable x to **True**. Then, the thread running `g()` runs to completion, setting x to **False**. Finally, the thread running `f()` runs, and the assertion fails.

[Figure 3.2](#) presents Harmony programs that illustrates non-atomicity. Again, the program on the left is sequential and the program on the right is concurrent. The Harmony **finally** statement is similar to an **assert** statement, except that the predicate is checked after all threads have terminated. Method `f()` increments variable x , which is initialized to 0. This time non-determinism does not play a role. Since `f()` is invoked twice, one might expect that $x = 2$ at the end of the execution of both programs. However, running the concurrent program through Harmony shows that this is not necessarily the case.

- Can you figure out why? Consider that **load** and **store** instructions of the underlying virtual machine architecture are atomic (indivisible).

```

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

```

(a) [[code/prog1.hny](#)] Sequential

```

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

```

(b) [[code/prog2.hny](#)] Concurrent

Summary: something went wrong in an execution

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
 - Line 1: Initialize x to True
 - **Thread terminated**
- Schedule thread T2: g()
 - Line 4: Set x to False (was True)
 - **Thread terminated**
- Schedule thread T1: f()
 - Line 3: Harmony assertion failed

(c) The output of running the concurrent code

Figure 3.1: The problem with non-determinism

```

1  x = 0
2
3  def f(): x += 1
4
5  f()
6  f()
7
8  finally x == 2

```

(a) [[code/prog3.hny](#)] Sequential

```

1  x = 0
2
3  def f(): x += 1
4
5  spawn f()
6  spawn f()
7
8  finally x == 2

```

(b) [[code/prog4.hny](#)] Concurrent

Summary: something went wrong in an execution

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
 - Line 1: Initialize x to 0
 - **Thread terminated**
- Schedule thread T1: **f()**
 - Preempted in **f()** about to store 1 into x in line 3
- Schedule thread T2: **f()**
 - Line 3: Set x to 1 (was 0)
 - **Thread terminated**
- Schedule thread T1: **f()**
 - Line 3: Set x to 1 (unchanged)
 - **Thread terminated**
- Schedule thread T3: **finally()**
 - Line 8: Harmony assertion failed

(c) The output of running the concurrent code

Figure 3.2: The problem with non-atomicity

```

1  x = 0
2
3  def f():
4      var register = x;
5      register += 1
6      x = register
7
8  spawn f()
9  spawn f()
10
11 finally x == 2

```

Figure 3.3: [\[code/prog5.hny\]](#) What actually happens in [Figure 3.2\(b\)](#)

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, the machine 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. [Figure 3.3](#) illustrates this. (The **var** statement declares a new local variable called *register*.) Conceptually, the machine

1. loads the value of x from its memory location into a register;
2. adds 1 to the register;
3. stores the new value into the memory location of x .

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 all possible interleavings of threads. Harmony will try all interleavings and report an error even if only one of those interleavings fails.

If the threads run one at a time to completion, then x will be incremented twice and ends up being 2. However, the following is also a possible interleaving of the two threads, which we will call T1 and T2 respectively:

1. T1 loads the value of x , which is 0;
2. T2 loads the value of x , which is still 0;
3. T2 adds 1 to the value that it loaded (0), and stores 1 into x ;
4. T1 adds 1 to the value that it loaded (0), and stores 1 into x ;

The result in this particular interleaving is that x ends up being 1. This is known as a *race condition*. When running a program with Harmony, it will report violations of assertions. Harmony also provides an example of an interleaving, like the one above, in which an assertion fails. [Figure 3.2\(c\)](#) shows the output of running [Figure 3.2\(b\)](#) through Harmony.

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:

```
finally (x == 1) or (x == 2)
```

This would fix the issue,¹ but more likely it is the program that must be fixed, not the specification.

The exercises below have you try the same thing (having threads concurrently increment an integer variable) 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. Deterministically triggering bugs allows you to find bugs in your concurrent programs much more easily than using a conventional programming environment.

Exercises

3.1 Harmony programs can usually be easily translated into Python by hand using minor modifications. For example, [Figure 3.4](#) is a Python version of [Figure 3.2](#).

1. Run [Figure 3.4](#) using Python. Does the assertion fail?
2. Using a script, run [Figure 3.4](#) 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 inc.py
done
```

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

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

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

3.2 [Figure 3.5](#) is a version of [Figure 3.4](#) that has each thread increment x N times. Also, the code slows down incrementing a bit by using a method. Run [Figure 3.5](#) 10 times (using Python). Report how many times the assertion fails and what the value of x was for each of the failed runs. Also experiment with other values of N . How large does N need to be for assertions to fail? (Try powers of 10 for N .)

¹Actually, Harmony still complains, this time about a *data race*, about which you will learn in [Chapter 4](#).


```
1  import threading
2
3  x = 0
4
5  def f():
6      global x
7      x += 1
8
9  t1 = threading.Thread(target=f)
10 t2 = threading.Thread(target=f)
11 t1.start()
12 t2.start()
13 t1.join()
14 t2.join()
15 assert x == 2
```

Figure 3.4: [\[python/inc.py\]](#) Python implementation of [Figure 3.2\(b\)](#)

3.3 Can you think of a fix to [Figure 3.2\(b\)](#)? 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 the **atomically** keyword or a lock, yet.)

```
1  import threading
2
3  N = 1000
4  x = 0
5
6  def inc(v):
7      return v + 1
8
9  def f():
10     global x
11     for i in range(N):
12         x = inc(x)
13
14  t1 = threading.Thread(target=f)
15  t2 = threading.Thread(target=f)
16  t1.start()
17  t2.start()
18  t1.join()
19  t2.join()
20  assert x == 2*N
```

Figure 3.5: [\[python/incmany.py\]](#) Using Python to increment 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). The Harmony compiler, **harmony**, places the bytecode for file *x.hny* in file *x.hvm*. The model checker (called *Charm*) executes the code in *x.hvm* and places its output in a file called *x.hco*. From the *x.hco* file, **harmony** creates a detailed human-readable output file in *x.hvb* and an interactive HTML file called *x.htm*. The *x.htm* file is automatically opened in your default web browser unless you specify the `--noweb` flag to **harmony**.

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 single or double quotes),
- sets of Harmony values,
- lists of Harmony values,
- dictionaries that map Harmony values to other Harmony values.

Strings that start with a letter or an underscore and only contain letters, digits, and underscores can be written without quotes by preceding them with a dot. So, *.example* is the same string as "example".

In Harmony (unlike Python), lists and tuples are the same type. As in Python, you can create a singleton tuple (or list) by including a comma. For example, $(1,)$ is a tuple consisting just of the number 1. Importantly, $(1) = 1 \neq (1,)$. Because, square brackets and parentheses work the same

in Harmony, `[a, b, c]` (which looks like a Python list) is the same Harmony value as `(a, b, c)` (which looks like a Python tuple). So, if $x = [\mathbf{False}, \mathbf{True}]$, then $x[0] = \mathbf{False}$ and $x[1] = \mathbf{True}$, 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`.

A dictionary maps keys to values. Unlike Python, which requires that keys must be hashable, any Harmony value can be a key, including another dictionary. Dictionaries are written as $\{k_0 : v_0, k_1 : v_1, \dots\}$. The empty dictionary is written as `{:}`. 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 $((d\ a)\ b)\ \dots$. 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 $d[.count]$ or $d["count"]$ (although any of those will work). Thus a dictionary can be made to look much like a Python object.

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.

[Section A.1](#) provides details on all the types of values that Harmony supports.

Harmony Bytecode

A Harmony program is translated into HVM bytecode. While the HVM is designed for efficient model checking, its architecture is nonetheless reminiscent 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 dictionary value.
- **Threads:** Any thread can spawn an unbounded number of other threads and threads may terminate. Each thread has a program counter (PC) that indexes into the code, a stack of Harmony values, and a private *register* that contains a Harmony dictionary value.

The register of a thread contains the local variables of the method that the thread is currently executing. The register is saved and restored by method invocations. The state of a thread is called a *context* (a.k.a. *continuation*): It contains the values of the thread's program counter, stack, and

```

0 Frame __init__()
1 Push 0
2 Store x
3 Jump 10
4 Frame f()
5 Load x
6 Push 1
7 2-ary +
8 Store x
9 Return

```

Figure 4.1: The first part of the HVM bytecode corresponding to [Figure 3.2\(b\)](#)

register. The HVM state 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 at the same time.

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 strings (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 strings.

Because directories are Harmony values themselves, and Harmony values include dictionaries and lists that themselves contain other Harmony values, 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 the example program shown in [Figure 3.2](#), the memory is a dictionary with a single entry: `.x`, and its address is the singleton sequence `[.x]`. As we will see later, an address is itself a Harmony value.

The program in [Figure 3.2\(b\)](#) is compiled into the HVM bytecode that begins as shown in [Figure 4.1](#). You can obtain a listing of this code by invoking `harmony` with the `-a` flag:

```
harmony -a code/prog4.hny
```

Each thread in the HVM is predominantly a *stack machine*, but it also has a register that contains the values of the local variables of the method that the thread is currently executing. Like shared memory, the register contains a dictionary so it can represent the values of multiple named variables. All individual instructions are atomically executed. The Harmony memory model is *sequentially consistent*: all accesses are one at a time and consistent with the program order.

The program of [Figure 3.2\(b\)](#) consists of a top-level method known as `__init__`. This code sets `x` to 0 and then spawns `f` twice. Execution of the bytecode begins at instruction 0 and continues until it reaches the last instruction in the bytecode, before any threads can run. After that, the spawned threads start executing.

Each defined method (e.g., `f`) as well as `__init__` starts with a `Frame` instruction and ends with a `Return` instruction. The `Frame` instruction lists the name of the method and the names of its arguments.

The definition of `f`, which appears in the middle of `__init__` at bytecode address 4, and which will be invoked by spawning, is bypassed by the `Jump 10` instruction at bytecode address 3 of the top-level code itself. The rest of the code for `__init__`, which continues at bytecode address 10, is not shown.

As shown in [Figure 4.1](#), the bytecode for method `f` begins with the `Frame` instruction at bytecode address 4 and continues with the instructions generated from `x += 1` (line 3 of [Figure 3.2\(b\)](#)), as follows:

- The `Load` instruction pushes the value of the `x` variable onto the stack.
- The `Push` instruction pushes the constant 1 onto the stack of the thread.
- `2-ary +` pops two values from the stack (the value of `x` and 1), adds them, and pushes the result back onto the stack.
- The `Store` instruction pops a Harmony value (the sum of the value of the `x` variable and 1) and stores it in the `x` variable.

You can think of Harmony as trying every possible interleaving of threads executing instructions. 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 were tried. This can include a failing assertion, behavior violations, 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 bad execution that it found (preferring short traces over longer ones).
- **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 a corresponding trace, but also the list of threads at that state, along with their program counters.
- **Behavior Violation:** The program can terminate in a state not allowed by the behavioral specification ([Chapter 10](#)).
- **Active Busy Waiting:** There are states in which some thread cannot make progress without the help of another thread, but does not block ([Chapter 12](#)).
- **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 ([Chapter 8](#)).

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 12](#)) 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 CPU cycles unnecessarily. A *data race* indicates that the execution depends on the semantics of the underlying memory operations and is therefore undesirable. Harmony also provides a warning if the set of generated behaviors does not contain all correct behaviors, which can happen if you provide Harmony with a description of all correct behaviors (see [Chapter 10](#)).

Issue: Safety violation				Shared Variables
Turn	Thread	Instructions Executed	PC	<i>x</i>
1	T0: __init__()	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div> terminated	28	0
2	T1: f()	<div><div></div><div></div><div></div><div></div></div> about to store 1 in variable x	8	0
3	T2: f()	<div><div></div><div></div><div></div><div></div><div></div></div> terminated	9	1
4	T1: f()	<div><div></div><div></div></div> terminated	9	1
5	T3: finally()	<div><div></div><div></div><div></div><div></div></div> Harmony assertion failed	25	1

__main__:8 finally x == 2

Threads	
ID	Status
T0	terminated
T1	terminated
T2	terminated
T3	failed atomic read-only

Stack Trace	Stack Top
__init__()	
f()	
f()	
finally()	

Figure 4.2: The HTML output of running Harmony on Figure 3.2(b)

If there is a failure, Harmony selects an execution that exhibits the failure and provides various descriptions. One is the markdown output. Harmony also generates a detailed and self-explanatory text output file (see `code/prog4.hvb`) and an interactive HTML file that allows the user to explore more details of the failed execution. Open the suggested HTML file and you should see something like Figure 4.2.

At the top, the HTML output reports the issue in red. Underneath there is a table that shows the five *turns* in the execution. A turn is a finite sequence of executed instructions, all performed by the same thread. The threads take turns until all threads have terminated or until an error occurs. Each row in the table displays a sequence of square blocks, one block for each instruction executed in the corresponding turn.

There is also information on what the thread is about to do at the end of its turn. The table lists, for each turn, the value of the program counter of the thread, the values of its shared variables, and the values that the thread has printed (none in this case).

In this case, at the end of turn 1, initialization (executed by thread T0) has completed, and the value of variable *x* is 0. At the end of turn 2, thread T1 is about to store the value 1 in *x*, but at that point T1 is preempted by thread T2. Thread T2 does store 1 into *x* in turn 3, and in turn 4, T1 resumes and also stores 1 into *x*. T3 is a *pseudo-thread* that runs when all threads have finished and evaluates the **finally** clauses.

The line of Harmony source code that is being executed is shown underneath the table.

The bottom left shows the bytecode of the program. Hovering the mouse over a machine instruction provides a brief explanation of what the instruction does.

The bottom right contains a table with the state of each thread. 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 table also displays the stack of each thread (although in the displayed example all threads have terminated.) The stack of each thread is subdivided into two parts: The *stack trace* and the *stack top*. A stack trace is a list of pending method invocations: Each shows the method's name and arguments, as well as its local variables. In this case, the *f* method does not invoke any other methods, and so the list has length 1. The stack top shows the values that the currently executing method has pushed onto the stack.

When first loaded, the HTML file shows the final state of the program's execution. But you can navigate to any intermediate point in various ways. For example, you can go to any point in the execution by clicking one of the blocks. When you do, the current turn and thread will be highlighted in yellow, while the bytecode instruction that is about to be executed is highlighted in red. [Figure 4.2](#) shows the final state, with the last turn and corresponding thread highlighted in yellow.

There are also various handy keyboard shortcuts to navigate through the execution quickly:

<i>Right arrow</i> :	go to the next bytecode instruction
<i>Left arrow</i> :	go back to the previous bytecode instruction
<i>Down arrow</i> :	go to the end of the current method
<i>Up arrow</i> :	go back to the beginning of the current method
<i>Enter (a.k.a. Return)</i> :	go to the next line of Harmony source code
<i>0</i> :	go to the initial state

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


```

1  shared = 0
2  entered = [ False, False ]
3
4  def f(self):
5      entered[self] = True
6      await not entered[1 - self]
7      shared += 1
8      entered[self] = False
9
10 spawn f(0)
11 spawn f(1)
12
13 finally shared == 2

```

Figure 4.3: [\[code/prog4enter.hny\]](#) Incorrect attempt at fixing the code of Figure 3.2(b)

Exercises

4.1 Figure 4.3 shows an attempt at trying to fix the code of Figure 3.2(b). 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 swapped Lines 5 and 6 of Figure 4.3? 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 before arriving at a working solution. In the case of [Figure 3.2](#), 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. The code that must be isolated is called a *critical section* (a.k.a. critical region) [[Dij62](#), [Dij65b](#), [Dij65a](#)]: A region of code where only one thread is allowed to execute at a time and that must be executed until completion. In the case of [Figure 3.2](#), the critical section is Line 5.

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 (it is a list of bits), but—as we have seen—incrementing a variable 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 property of the data structure to hold at the beginning and at the end of the critical section, but in the middle the property may be temporarily invalid—this is not a problem as critical sections guarantee that no other thread will be able to see the intermediate state of the

```
1  def thread():
2      while True:
3          # Critical section is here
4          pass
5
6  spawn thread()
7  spawn thread()
```

Figure 5.1: [code/csbarebones.hny] Modeling a critical section

```

1  # number of threads in the critical section
2  in_cs = 0
3  invariant in_cs in { 0, 1 }
4
5  def thread():
6      while choose { False, True }:
7          # Enter critical section
8          atomically in_cs += 1
9
10         # Critical section is here
11         pass
12
13         # Exit critical section
14         atomically in_cs -= 1
15
16     spawn thread()
17     spawn thread()

```

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

data structure. An implementation of a data structure that can be safely accessed by multiple threads and is free of race conditions is called *thread-safe*.

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. We need to ensure is that there can never be two or more threads in the critical section. This property is called *mutual exclusion*. 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 executing in the critical section 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 without entering the critical section ever. (Recall that Harmony will try every possible execution, and so it will evaluate both choices.) As it turns out, there is an advantage to doing it this way: We also test if a thread can enter when there is no other thread trying to enter the critical section. As we will see below, this is not always obvious.

This code specifies that at most one thread can be executing in the critical section by using a counter *in_cs* that a thread atomically increments when entering the critical section and atomically decrements when it leaves. The code specifies the invariant that *in_cs* must be either 0 or 1. You can think of this as the type of *in_cs*.

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 an attempt at mutual exclusion based on a naïve (and, as it turns out, incorrect) 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 fails. Figure 5.3 also shows the Harmony output. `thread(1)` finds that the lock is available, but just before storing **True** in *lockTaken*, `thread(0)` gets to run. Because *lockTaken* is still **False**, `thread(0)` also believes it can acquire the lock, stores **True** in *lockTaken*, and moves on to the critical section. Finally, `thread(1)` moves on, also storing **True** into *lockTaken* and also moving into the critical section. 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 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 5.4 presents a solution based on each thread having a flag indicating that the thread 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 ($1 - \text{self}$) 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 $flags[i] = \text{True}$. 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 , $t_1 > t_0$, the thread finds that $flags[1] = \text{False}$. Because of the invariant, $flags[1] = \text{False}$ 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 $flags[1] = \text{False}$ at time t_1 . To reach the critical section between t_1 and

```

1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  lockTaken = False
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          await not lockTaken
10         lockTaken = True
11
12         atomically in_cs += 1
13         # Critical section
14         atomically in_cs -= 1
15
16         # Leave critical section
17         lockTaken = False
18
19  spawn thread(0)
20  spawn thread(1)

```

Summary: something went wrong in an execution

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
 - Line 1: Initialize *in_cs* to 0
 - Line 4: Initialize *lockTaken* to **False**
 - **Thread terminated**
- Schedule thread T3: thread(1)
 - Line 7: Choose **True**
 - Preempted in thread(1) about to store **True** into *lockTaken* in line 10
- Schedule thread T2: thread(0)
 - Line 7: Choose **True**
 - Line 10: Set *lockTaken* to **True** (was **False**)
 - Line 12: Set *in_cs* to 1 (was 0)
 - Preempted in thread(0) about to execute atomic section in line 14
- Schedule thread T3: thread(1)
 - Line 10: Set *lockTaken* to **True** (unchanged)
 - Line 12: Set *in_cs* to 2 (was 1)
 - Preempted in thread(1) about to execute atomic section in line 14
- Schedule thread T1: invariant()
 - Line 2: Harmony assertion failed

Figure 5.3: [\[code/naiveLock.hny\]](#) Naïve implementation of a shared lock and the markdown output of running Harmony

```

1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  flags = [ False, False ]
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          flags[self] = True
10         await not flags[1 - self]
11
12         atomically in_cs += 1
13         # Critical section
14         atomically in_cs -= 1
15
16         # Leave critical section
17         flags[self] = False
18
19  spawn thread(0)
20  spawn thread(1)

```

Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
 - Line 1: Initialize *in_cs* to 0
 - Line 4: Initialize *flags* to [**False**, **False**]
 - **Thread terminated**
- Schedule thread T1: thread(0)
 - Line 7: Choose **True**
 - Line 9: Set *flags*[0] to **True** (was **False**)
 - Preempted in thread(0) about to load variable *flags*[1] in line 10
- Schedule thread T2: thread(1)
 - Line 7: Choose **True**
 - Line 9: Set *flags*[1] to **True** (was **False**)
 - Preempted in thread(1) about to load variable *flags*[0] in line 10

Final state (all threads have terminated or are blocked):

- Threads:
 - T1: (blocked) thread(0)
 - * about to load variable *flags*[1] in line 10
 - T2: (blocked) thread(1)
 - * about to load variable *flags*[0] in line 10

Figure 5.4: [\[code/naiveFlags.hny\]](#) Naïve use of flags to solve mutual exclusion

```

1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  turn = 0
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          turn = 1 - self
10         await turn == self
11
12         atomically in_cs += 1
13         # Critical section
14         atomically in_cs -= 1
15
16         # Leave critical section
17
18     spawn thread(0)
19     spawn thread(1)

```

Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
 - Line 1: Initialize *in_cs* to 0
 - Line 4: Initialize *turn* to 0
 - **Thread terminated**
- Schedule thread T2: thread(1)
 - Line 7: Choose False
 - **Thread terminated**
- Schedule thread T1: thread(0)
 - Line 7: Choose True
 - Line 9: Set *turn* to 1 (was 0)
 - Preempted in thread(0) about to load variable *turn* in line 10

Final state (all threads have terminated or are blocked):

- Threads:
 - T1: (blocked) thread(0)
 - * about to load variable *turn* in line 10
 - T2: (terminated) thread(1)

Figure 5.5: [\[code/naiveTurn.hny\]](#) Naïve use of turn variable to solve mutual exclusion

```

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

```

Figure 5.6: [\[code/Peterson.hny\]](#) Peterson's Algorithm

t_2 , thread 1 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, 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 15](#).) 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.

You may now start to think that a solution to the mutual exclusion problem without special hardware support is impossible. However, in 1981, Gary L. Peterson came up with a beautiful solution, now known as “Peterson’s Algorithm” [Pet81]. The algorithm is an amalgam of the (incorrect) algorithms in Figure 5.4 and Figure 5.5, and is presented in Figure 5.6. (The first line specifies that the *flags* and *turn* variables are assumed to satisfy *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. The thread 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 ($flag[1 - self] = \text{False}$) or has given way ($turn = self$).

Running the algorithm with Harmony shows that it satisfies both mutual exclusion and progress. Why does it work? To answer that we need to prove the correctness of Peterson’s algorithm. Unfortunately, proving the correctness of Peterson’s algorithm is not trivial. You can find an informal proof in Appendix H.

Peterson’s algorithm implements a critical section, but it is not efficient, especially if generalized to more than two 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 still requires two operations. And even with a 32-bit data bus, the problem could arise if the variable is not aligned on a 4-byte word boundary.
- 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 5.4 shows, Peterson’s algorithm breaks down if such seemingly independent 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 in between?

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 atomically (so concurrently invoked instructions are sequentialized), and that this order should be consistent with the order of instructions invoked on each thread. The default memory models of C and Java are not sequentially consistent. The **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.

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.

5.3 [Figure 5.7](#) presents another solution to the mutual exclusion problem. The solution is similar to the one in [Figure 5.4](#), 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*? Does the algorithm *guarantee* that at least one thread can enter the critical section?

5.4 Consider 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. Then run the program in [Figure 5.6](#) after reversing the two assignments and describe in English what happens.

5.5 Bonus question: Can you generalize Peterson’s algorithm to more than two threads?

5.6 Bonus question: Implement [Dekker’s Algorithm](#), [Dijkstra’s Algorithm](#) [[Dij65b](#)], [Eisenstein and McGuire’s Algorithm](#), [Szymański’s Algorithm](#), or [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  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  sequential flags
5  flags = [ False, False ]
6
7  def thread(self):
8      while choose({ False, True }):
9          # Enter critical section
10         flags[self] = True
11         while flags[1 - self]:
12             flags[self] = False
13             flags[self] = True
14
15         atomically in_cs += 1
16         # Critical section
17         atomically in_cs -= 1
18
19         # Leave critical section
20         flags[self] = False
21
22  spawn thread(0)
23  spawn thread(1)

```

Figure 5.7: [[code/csonebit.hny](https://code.csonebit.hny)] Mutual exclusion using a flag per thread

Chapter 6

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 and lists (see [Chapter 4](#)). Dictionaries, lists, and methods all map Harmony values to Harmony values, and their syntax is indistinguishable. If `f` is a method, list, or dictionary, and `x` is some Harmony value, then `f x`, `f(x)`, and `f[x]` are all the same expression in Harmony.

Harmony does not have a **return** statement. Using the **returns** clause of **def**, a result variable can be declared, for example: **def** `f()` **returns** *something*. The result of the method should be assigned to variable *something*. Harmony also does not 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 is 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 an object reference 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` evaluates to 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` (written `p->field`), which greatly improves readability. However, if `p` points to a list, you have to write `(!p)[i]` to index into the list. (Unlike C, Harmony does not support pointer arithmetic.)

```

1  def Peterson_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 Peterson_exit(pm, pid):
7      pm→flags[pid] = False
8
9  def Peterson_mutex() returns result:
10     result = { .turn: choose({0, 1}), .flags: [ False, False ] }
11
12     ##### The code above can go into its own Harmony module #####
13
14     in_cs = 0
15     invariant in_cs in { 0, 1 }
16
17     sequential mutex
18     mutex = Peterson_mutex()
19
20     def thread(self):
21         while choose({ False, True }):
22             Peterson_enter(?mutex, self)
23
24             atomically in_cs += 1
25             # Critical section
26             atomically in_cs -= 1
27
28             Peterson_exit(?mutex, self)
29
30     spawn thread(0)
31     spawn thread(1)

```

Figure 6.1: [\[code/PetersonMethod.hny\]](#) Peterson’s Algorithm accessed through methods

```

1  current = [ [1, 2, 3], [], [] ]
2
3  while current[2] != [1, 2, 3]:
4      let moves = { (s, d) for s in {0..2} for d in {0..2}
5                  where current[s] != []
6                  where (current[d] == [] or (current[s][0] < current[d][0])) }
7      let (src, dst) = choose moves:
8      print str(src) + " -> " + str(dst)
9      current[dst] = [current[src][0],] + current[dst]
10     del current[src][0]

```

Figure 6.2: [\[code/hanoi.hny\]](#) Nondeterministic Towers of Hanoi

```

1  def towers(n, src, dst, other):
2      if n > 0:
3          towers(n - 1, src, other, dst)
4          print str(src) + " -> " + str(dst)
5          towers(n - 1, other, dst, src)
6
7  towers(3, 1, 2, 3)

```

Figure 6.3: [\[code/hanoi2.hny\]](#) Recursive Towers of Hanoi

Figure 6.1 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. `Peterson_mutex` is a method that returns a “mutex,” which, in this case, is a dictionary that contains Peterson’s Algorithm’s shared memory state: a turn variable and two flags. Both methods `Peterson_enter` and `Peterson_exit` take two arguments: a pointer to a mutex and the thread identifier (0 or 1). $pm \rightarrow turn$ is the value of the *.turn* key in the dictionary that *pm* points to.

You can put the first three methods in its own Harmony source file and include the file using the Harmony **import** statement. This would make the code usable by multiple applications.

Finally, methods can have local variables.

Method variables are either mutable (writable) or immutable (read-only). The arguments to a method and the bound variable (or variables) within a **for** statement are immutable; the result variable is mutable. Using the **var** statement, new mutable local variables can be declared. For example, **var** *x* = 3 declares a new mutable local variable *x*. The **let** statement allows declaring new immutable local variables. For example: **let** *x* = 3: *y* += *x* adds 3 to variable *y*.

As an example of using **let**, Figure 6.2 solves the *Towers of Hanoi* problem. If you are not familiar with this problem: There are three towers with disks of varying sizes. In the initial configuration, the first tower has three disks (of sizes 1, 2, and 3), with the largest disk at the

```

1  const FIFO = False
2
3  def CLOCK(n) returns result:
4      result = { .entries: [None,] * n, .recent: {}, .hand: 0, .misses: 0 }
5
6  def ref(clock, x):
7      if x not in clock→entries:
8          while clock→entries[clock→hand] in clock→recent:
9              clock→recent -= {clock→entries[clock→hand]}
10             clock→hand = (clock→hand + 1) % len(clock→entries)
11             clock→entries[clock→hand] = x
12             clock→hand = (clock→hand + 1) % len(clock→entries)
13             clock→misses += 1
14      if not FIFO:
15          clock→recent |= {x}
16
17  clock3, clock4, refs = CLOCK(3), CLOCK(4), []
18
19  const VALUES = { 1..5 }
20
21  var last = {}
22  for i in {1..100}:
23      let x = i if i < 5 else choose(VALUES - last):
24          refs = refs + [x,]
25          ref(?clock3, x); ref(?clock4, x)
26          assert(clock4.misses <= clock3.misses)
27          last = {x}

```

Figure 6.4: [\[code/clock.hny\]](#) Harmony program that finds page replacement anomalies

bottom, while the other two towers are empty. You are allowed to move a top disk from one tower to another, but you are not allowed to stack a larger disk on a smaller one. The objective is to move the disks from the first tower to the third one. The program tries valid moves non-deterministically until it finds a solution. Since Harmony tries all possible executions, the output of this program shows all possible solutions to the problem.

The program also contains an example of set comprehension (Lines 4 to 6). This is similar to set comprehension in Python, except that Harmony uses the keyword **where** instead of **if** (simplifying parsing).

Figure 6.3 shows the well-known recursive solution for Towers of Hanoi. It is deterministic and therefore outputs only a single solution.

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, Bélády 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 6.4 will find exactly the same scenario if you define `FIFO` to be **True**. Moreover, if you define `FIFO` to be **False**, Harmony will find a scenario for the CLOCK replacement policy [Cor69].

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. (The set 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 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.

Exercises

6.1 (This is just for fun or exercise as it is not a concurrent programming problem.) Implement a Harmony program that finds solutions to the “cabbage, goat, and wolf” problem. In this problem, a person accompanied by these three items has to cross a stream in a small boat, but can only take one item at a time. So, the person has to cross back and forth several times, leaving two items on one or the other shore by themselves. Unfortunately, if left to themselves, the goat would eat the cabbage and the wolf would eat the goat. What crossings does the person need to make in order not to lose any items?

Chapter 7

Specifying a Lock

So far, we have used Harmony to *implement* various algorithms. But Harmony can also be used to *specify* what an algorithm is supposed to do. For example, [Figure 7.1](#) specifies the intended behavior of a lock. In this case, a lock is a boolean, initially **False**, with two operations, **acquire()** and **release()**. The **acquire()** operation waits until the lock is **False** and then sets it to **True** in an atomic operation. The **release()** operation sets the lock back to **False**. The code is similar to [Figure 5.3](#), except that waiting for the lock to become available and taking it is executed as an atomic operation.

The code in [Figure 7.1](#) is also in Harmony’s **synch** module. [Figure 7.2](#) shows how locks may be used to implement a critical section. [Figure 7.3](#) gives an example of how locks may be used to fix the program of [Figure 3.2\(b\)](#).

Note that the code of [Figure 7.1](#) is executable in Harmony. However, the **atomically** keyword is not available in general programming languages and should not be used for implementation. Peterson’s algorithm is an implementation of a lock, although only for two processes. In the following chapters, we will look at more general ways of implementing locks using atomic constructions that are usually available in the underlying hardware.

In Harmony, any statement can be preceded by the **atomically** keyword. It means that the statement as a whole is to be executed atomically. (Harmony can accomplish this by controlling how threads are scheduled.) The **atomically** keyword can be used to specify the behavior of methods such as **acquire** and **release**. But an actual executable program should not use the **atomically** keyword because—on a normal machine—it cannot be directly compiled into machine code. If we want to make the program executable on hardware, we have to show how **Lock**, **acquire**, and **release** are implemented, not just how they are specified. [Chapter 8](#) presents such implementations.

The code in [Figure 7.1](#) also uses the Harmony **when** statement. A **when** statement waits until a time in which condition holds (not necessarily the first time) and then executes the statement block. The “**await condition**” statement is the same as “**when condition: pass**”. Combined with the **atomically** keyword, the entire statement is executed atomically at a time that the condition holds.

It is important to appreciate the difference between an *atomic section* (the statements executed within an atomic block of statements) and a *critical section* (protected by a lock of some sort). The former ensures that while the statements in the block are executing no other thread can execute.

```

1  def Lock() returns result:
2      result = False
3
4  def acquire(lk):
5      atomically when not !lk:
6          !lk = True
7
8  def release(lk):
9      assert !lk
10     atomically !lk = False

```

Figure 7.1: [\[code/lock.hny\]](#) Specification of a lock

```

1  from synch import Lock, acquire, release
2
3  const NTHREADS = 5
4
5  thelock = Lock()
6
7  def thread():
8      acquire(?thelock)
9      pass           # critical section is here
10     release(?thelock)
11
12  for i in {1..NTHREADS}:
13      spawn thread()

```

Figure 7.2: [\[code/lock_demo.hny\]](#) Using a lock to implement a critical section

```

1  from synch import Lock, acquire, release
2
3  shared = 0
4  thelock = Lock()
5
6  def f():
7      acquire(?thelock)
8      shared += 1
9      release(?thelock)
10
11 spawn f()
12 spawn f()
13
14 finally shared == 2

```

Figure 7.3: [\[code/prog4lock.hny\]](#) Figure 3.2(b) fixed with a lock

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.

Atomic statements are not intended to replace locks or other synchronization primitives. When implementing synchronization solutions you should not directly use atomic statements but use the synchronization primitives that are available to you. But if you want to *specify* a synchronization primitive, then use **atomically** by all means. You can also use atomic statements in your test code. In fact, **assert** statements are executed atomically and specify that some condition must hold in every execution of the program. Note two important differences with other programming languages: In most programming languages, an **assert** statement only checks the current execution, and the condition is not evaluated atomically.

Chapter 8

Lock Implementations

Locks are 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. (An *inconsistent* state of a data structure is either an invalid state or a state that does not reflect a possible value in the execution.)

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 there is a state in which multiple threads are trying to access the same variable, at least one of those accesses updates the variable, and at least one of those accesses is non-atomic. 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.

Because atomic operations cannot overlap, data races can be avoided by making operations atomic. In Harmony, this can be done in two ways. First, using the **sequential** statement, you can specify that concurrent load and store operations to the specified variables are considered atomic. Second, you can make accesses atomic using the **atomically** keyword. However, these are Harmony specification constructs. In a practical implementation, a programmer will often use one or more locks to prevent overlapping access to shared variables.

To implement locks, 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. We can use the **atomically** keyword to specify 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  const N = 5
2
3  in_cs = 0
4  invariant in_cs in { 0, 1 }
5
6  shared = False
7  private = [ True, ] * N
8  invariant [x for x in [shared,] + private where not x] == [False,]
9
10 def swap(s, p):
11     atomically !p, !s = !s, !p
12
13 def thread(self):
14     while choose({ False, True }):
15         # Enter critical section
16         while private[self]:
17             swap(?shared, ?private[self])
18
19         atomically in_cs += 1
20         assert not private[self]
21         atomically in_cs -= 1
22
23         # Leave critical section
24         swap(?shared, ?private[self])
25
26 for i in {0..N-1}:
27     spawn thread(i)

```

Figure 8.1: [\[code/spinlock.hny\]](#) Mutual Exclusion using a “spinlock” based on atomic swap

```

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

```

To support implementing locks, some CPUs have an atomic “swap” operation. Method `swap` in Figure 8.1 shows its specification. Here s points to a shared boolean variable (the lock) and p to a private boolean variable, belonging to some thread. The operation exchanges the value of the shared variable and the private variable.

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 $N + 1$ boolean variables. Variable *shared* (the lock) 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 exactly one of these variables is **False** (expressed in Harmony in Line 8). Another invariant, $\mathcal{I}_2(i)$, is that if thread i is in the critical section, then *private*[i] = **False** (expressed in Line 20). 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.

To see that invariant \mathcal{I}_1 is maintained, we can use induction. Note that $!p = \mathbf{True}$ upon entry of `swap` (because of the condition on the **while** loop that the `swap` method is invoked in). There are two cases:

1. $!s$ is **False** upon entry to `swap`. Then upon exit $!p = \mathbf{False}$ and $!s = \mathbf{True}$, maintaining the invariant.
2. $!s$ is **True** upon entry to `swap`. 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. $\mathcal{I}_1(i)$ is specified in Line 8, while $\mathcal{I}_2(i)$ is expressed in Line 20 using an **assert** statement in the critical section. The expression in Line 8 counts the number of **False** values in the list consisting of the shared variable and the private variables and checks that the result is 1.

The lock implementation in Figure 8.1 uses 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 need to keep explicit track of whether it has a lock or not, because it is 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. But in practice this is very expensive: An atomic swap operation requires two memory loads and two memory stores.

A more common solution to implementing a lock uses an atomic *test-and-set* instruction provided by most CPUs. Figure 8.2 shows the specification of `test_and_set` and the implementation of a spinlock based on it. Method `test_and_set` involves only a single variable: The method sets the

```

1  def test_and_set(s) returns oldvalue:
2      atomically:
3          oldvalue = !s
4          !s = True
5
6  def atomic_store(p, v):
7      atomically !p = v
8
9  def Lock() returns initvalue:
10     initvalue = False
11
12  def acquire(lk):
13     while test_and_set(lk):
14         pass
15
16  def release(lk):
17     atomic_store(lk, False)

```

Figure 8.2: [\[code/lock_tas.hny\]](#) Implementation of the lock specification in Figure 7.1 using a spinlock based on test-and-set

variable to **True** and returns the old value of the variable. The variable represents the lock: If free, the value is **False**; if taken, the value is **True**. Method `Lock()` returns the initial value of a lock, which is **False**. Method `acquire(lk)` invokes `test_and_set(lk)` until it returns **False**, meaning that the lock variable pointed to by `lk` was available (but now taken). Method `release(lk)` releases the lock by setting `lk` back to **False**. The lock is cleared in an atomic statement to prevent a data race. (CPUs typically provide an atomic store operation.)

Besides this solution being more efficient than Figure 8.1, the solution is general for any number of threads.

You can test the spinlock with the program in Figure 8.4 using the command `harmony -m synch=lock_tas code/lock_test1.hny`. The `-m` flag tells Harmony to use the `lock_tas.hny` file for the `synch` module rather than the standard `synch` module (which contains only a specification of the lock methods). The test program has a collection of threads repeatedly enter a critical section and testing that there is at most one thread in the critical section at any time.

The spinlock implementation suffers potentially from *starvation*: An unlucky thread may never be able to get the lock while other threads successfully acquire the lock one after another. It could even happen with just two threads: One thread might successfully acquire the lock repeatedly in a loop, while another thread is never lucky enough to acquire the lock in between.

A *ticket lock* (Figure 8.3) is an implementation of a lock that prevents starvation using an atomic *fetch-and-increment* operator. It is inspired by European bakeries. A European bakery often has a clearly displayed counter (usually just two digits) and a ticket dispenser. Tickets are numbered 0 through 99 and repeat over and over again (in the case of a two digit counter). When a customer walks into the bakery, they draw a number from the dispenser and wait until their number comes

```

1  const MAX_THREADS = 8
2
3  def fetch_and_increment(p) returns oldvalue:
4      atomically:
5          oldvalue = !p
6          !p = (!p + 1) % MAX_THREADS
7
8  def atomic_load(p) returns value:
9      atomically value = !p
10
11 def Lock():
12     result = { .counter: 0, .dispenser: 0 }
13
14 def acquire(lk):
15     let my_ticket = fetch_and_increment(?lk→dispenser):
16         while atomic_load(?lk→counter) != my_ticket:
17             pass
18
19 def release(lk):
20     fetch_and_increment(?lk→counter)

```

Figure 8.3: [[code/lock_ticket.hny](#)] Implementation of the lock specification in Figure 7.1 using a ticket lock


```

1  import lock
2
3  const N = 5
4
5  in_cs = 0
6  invariant in_cs in { 0, 1 }
7
8  thelock = lock.Lock()
9
10 def thread():
11     while choose({ False, True }):
12         lock.acquire(thelock)
13
14         atomically in_cs += 1
15         # Critical section
16         atomically in_cs -= 1
17
18         lock.release(thelock)
19
20 for i in {1..N}:
21     spawn thread()

```

Figure 8.4: [\[code/lock_test1.hny\]](#) A test program for locks (based on [Figure 5.2](#))

```

1  def Lock() returns result:
2      result = { .acquired: False, .suspended: [] }
3
4  def acquire(lk):
5      atomically:
6          if lk→acquired:
7              stop ?lk→suspended[len lk→suspended]
8              assert lk→acquired
9          else:
10             lk→acquired = True
11
12 def release(lk):
13     atomically:
14         assert lk→acquired
15         if lk→suspended == []:
16             lk→acquired = False
17         else:
18             go (lk→suspended[0]) ()
19             del lk→suspended[0]

```

Figure 8.5: [\[modules/lock_susp.hny\]](#) Lock implementation using suspension

up. Every time a customer has been helped, the counter is incremented. (Note that this only works if there can be no more than 100 customers in the bakery at a time.)

Figure 8.3 uses two variables for a ticket lock, *counter* and *dispenser*. When a thread tries to acquire the lock, it first fetches the current value of the dispenser variable (i.e., the ticket) and increments the variable modulo `MAX_THREADS`, all in one atomic operation. In practice, `MAX_THREADS` would be a number like 2^{32} or 2^{64} , but since the Harmony model checker checks every possible state, limiting `MAX_THREADS` to a small number helps to keep the time and memory needed to model check a Harmony program within reason. Moreover, it is easier to check that it fails when you run it with more than `MAX_THREADS` threads. After obtaining the ticket, the thread waits until its ticket value equals the counter. Note that loading the counter must also be done atomically in order to avoid a data race. To release the lock, it suffices to atomically increment the counter. You can test the implementation using the command `harmony -m synch=lock_ticket code/lock_test1.hny`. To see it fail, try `harmony -c N=10 -m synch=lock_ticket code/lock_test1.hny`.

We now turn to a radically different way of implementing locks, one that is commonly provided by operating systems to user processes. 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 lock held by another thread is a good example of a thread being blocked. The only way forward is if the other thread releases the lock.

In most operating systems, threads are virtual (as opposed to “raw CPU cores”) and can be suspended until some condition changes. For example, threads are blocked while they wait for a

disk block they are trying to read or while they wait for a network message to arrive. Similarly, 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 register (containing the current method's variables). In Harmony, a context is a Harmony value and can be saved in a variable just like any other value. The syntax of the expression that a thread executes to suspend itself is as follows:

```
stop s
```

Here *s* is a pointer to some variable. The expression 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 !s r
```

Here *r* is any Harmony value. The statement causes a thread with the context contained in *!s* to be added to the state that has just executed the **stop** *s* expression. The **stop** expression that this thread last executed resumes by returning the value *r*.

Figure 8.5 shows the lock interface using suspension. It is implemented as follows:

- A lock maintains both a boolean indicating whether the lock is currently acquired and a list of contexts of threads that want to acquire the lock.
- **acquire()** acquires the lock 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 block—this is the only exception to such an atomic statement block running to completion. While the thread is running no other threads can run but, after the thread suspends itself, other threads can run.
- **release()** checks to see if any threads are waiting to acquire the lock. If so, it selects the first context on the list and resumes it.

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 (FIFO) is a common choice in lock implementations as it is easy and prevents starvation.

You will find that using the *implementation* of a lock instead of the *specification* of a lock (in the **synch** module) often leads to the model checker searching a significantly larger state space. Thus it makes sense to model check larger programs in a modular fashion: Model check one module implementation at a time, using specifications for the other modules.

Exercises

8.1 Run the program in Figure 7.3 using (i) **synch** and (ii) **synchS**. Report how many states were explored by Harmony for each module.

8.2 Figure 8.6 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.

```

1  x, y = 0, 100
2
3  def setX(a):
4      x = a
5      y = 100 - a
6
7  def getXY() returns xy:
8      xy = [x, y]
9
10 def checker():
11     let xy = getXY():
12         assert (xy[0] + xy[1]) == 100, xy
13
14 spawn checker()
15 spawn setX(50)

```

Figure 8.6: [\[code/xy.hny\]](#) Incomplete code for [Exercise 8.2](#) with desired invariant $x + y = 100$

8.3 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 lock was available (and now acquired) or **False** if the lock was already acquired. Hint: You do not have to change the existing code.

8.4 People who use an ATM often first check their balance and then withdraw a certain amount of money not exceeding their balance. A negative balance is not allowed. [Figure 8.7](#) 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 8.7](#). Note, you should leave the customer code the same. You are only allowed to change the `atm_` methods, and you cannot use the **atomically** keyword.

```

1  from synch import Lock, acquire, release
2
3  const N_ACCOUNTS = 2
4  const N_CUSTOMERS = 2
5  const N_ATMS = 2
6  const MAX_BALANCE = 1
7
8  accounts = [ { .lock: Lock(), .balance: choose({0..MAX_BALANCE})}
9               for i in {1..N_ACCOUNTS} ]
10
11 invariant min(accounts[acct].balance for acct in {0..N_ACCOUNTS-1}) >= 0
12
13 def atm_check_balance(acct) returns balance: # return the balance on acct
14     acquire(?accounts[acct].lock)
15     balance = accounts[acct].balance
16     release(?accounts[acct].lock)
17
18 def atm_withdraw(acct, amount) returns success: # withdraw amount from acct
19     assert amount >= 0
20     acquire(?accounts[acct].lock)
21     accounts[acct].balance -= amount
22     release(?accounts[acct].lock)
23     success = True
24
25 def customer(atm, acct, amount):
26     assert amount >= 0
27     let bal = atm_check_balance(acct):
28         if amount <= bal:
29             atm_withdraw(acct, amount)
30
31 for i in {1..N_ATMS}:
32     spawn customer(i, choose({0..N_ACCOUNTS-1}),
33                   choose({0..MAX_BALANCE}))

```

Figure 8.7: [\[code/atm.hny\]](#) Withdrawing money from an ATM

Chapter 9

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. Concurrent queues turn out to be very important in many concurrent and distributed programs, particularly in so-called *producer/consumer* scenarios where one or more threads are producing items and one or more other threads consume them.

The `queue` module can be defined as follows:

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

Figure 9.1(a) shows a sequential specification for such a queue in Harmony. Sequential specifications are useful to write down. They tend to be easy to understand as you only need to understand each operation individually without having to worry about interactions between operations, because we are assuming that operations are executed one at a time. Also, when writing down a specification, you do not have to worry about efficiency. In this particular case, we capture the state of a queue by a list. As a result, in the absence of concurrency, the sequential specification of the queue is also a credible, if inefficient queue *implementation*.

Unfortunately, this sequential specification does not work well with threads concurrently accessing this queue. This is because operations can *overlap* in execution, and so one operation can witness the intermediate state of another operation. As a compiler would be free to implement the sequential specification in any way it wants to, it gets to decide what data structures to use to store the list, and what machine instructions to use to implement the operations. For a sequential specification that is not a problem because we only need to understand the state in between operations, but in a concurrent setting Figure 9.1(a) is underspecified; it does not say what happens when two or more operations overlap.

Figure 9.1(b) shows the corresponding concurrent specification. It is almost identical to the sequential specification, but it states that the actual execution of an operation of each method has to happen atomically sometime between invoking the method and the method completing. The specification cannot be used as an implementation for a queue, as processors generally do not have atomic operations on lists, but it will work well as a specification. See Figure 9.2 for a simple demonstration program that uses a concurrent queue; the specification can be used with any

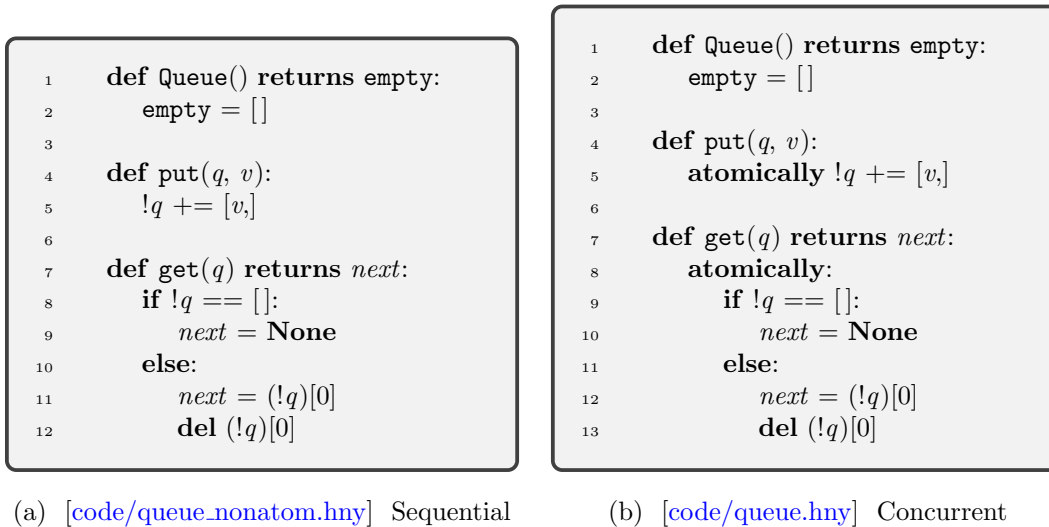


Figure 9.1: A sequential and a concurrent specification of a queue

```

1  import queue
2
3  def sender(q, v):
4      queue.put(q, v)
5
6  def receiver(q):
7      let v = queue.get(q):
8          assert v in { None, 1, 2 }
9
10     demoq = queue.Queue()
11     spawn sender(?demoq, 1)
12     spawn sender(?demoq, 2)
13     spawn receiver(?demoq)
14     spawn receiver(?demoq)

```

Figure 9.2: [[code/queue_test1.hny](#)] Using a concurrent queue

concurrent program that uses a queue. But this concurrent specification is too much for compilers to handle; we have to write down how to implement it.

While in the concurrent specification operations cannot overlap execution, the same need not be true for an implementation. Indeed, we will show two implementations of a concurrent queue. The first uses critical sections, ensuring that operations cannot overlap. The second allows a **get** and a **put** operation on a non-empty queue to execute simultaneously. Both are correct implementations of the concurrent queue specification.

In both implementations, the queue data structure is maintained as a linked list. They use the **alloc** module for dynamic allocation of nodes in the list, which provides methods **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 implementation in Figure 9.3 uses a lock-based critical section to operate on the data structure. The implementation 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() returns the initial value for a queue object consisting of a **None** head and tail pointer and a lock. The **put(*q*, *v*)** and **get(*q*)** methods both take a pointer *q* to the queue object. It has to be a pointer rather than the value of the queue object because both methods must be able to modify the queue object. Before a method accesses the value of the head or tail of the queue, it first obtains the lock. When the method is done, it releases the lock. Doing so prevents concurrently executing methods from witnessing an intermediate, inconsistent state of the linked list.

Importantly, note Lines 7 and 8 in Figure 9.2. 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** and **finally** statements) must be predicates over the state and thus are not allowed to change the state.

The implementation in Figure 9.3 uses a single lock to protect the queue data structure. A thread acquires the lock before any access and releases it afterwards. This is possibly the simplest way to make a data structure concurrent. However, it can cause significant contention over the lock between a producer and a consumer, reducing performance by serializing operations.

Figure 9.4 shows another concurrent queue implementation that addresses this problem [MS96]. It is based on the insight that a **put** method operates on the tail of a queue, while a **get** method operates on the head, and so in theory they should be able to execute concurrently. Although the algorithm is well-known, what is not often realized (because it is not stated explicitly in the paper) is that it requires sequentially consistent memory. The algorithm must be coded carefully to work correctly with modern programming languages and computer hardware that generally lack sequentially consistent memory.

The implementation uses separate locks for the head and the tail, allowing a **put** and a **get** operation to proceed concurrently. The implementation uses a dummy node (a.k.a. sentinel node) at the head of the linked list. Except initially, the dummy node is the last node that was dequeued. Using dummy nodes can simplify implementations because they can reduce the number of exceptional conditions. In this case, note that neither the *head* nor *tail* pointer are ever **None**, reducing the need for **if** statements in the code.


```

1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def Queue() returns empty:
5      empty = { .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) returns next:
18      acquire(?q→lock)
19      let node = q→head:
20         if node == None:
21             next = None
22         else:
23             next = node→value
24             q→head = node→next
25             if q→head == None:
26                 q→tail = None
27             free(node)
28      release(?q→lock)

```

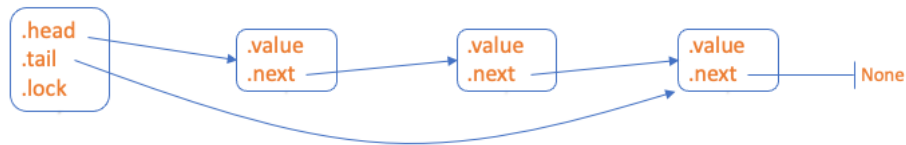


Figure 9.3: [\[code/queue_lock.hny\]](#) An implementation of a concurrent queue data structure and a depiction of a queue with three elements

```

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

```

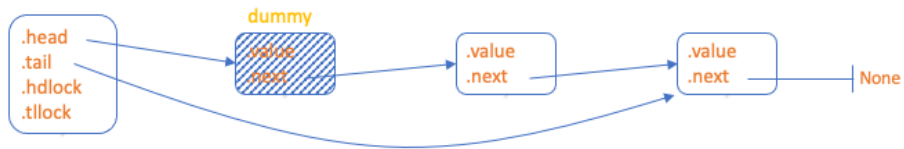


Figure 9.4: [\[code/queue_MS.hny\]](#) A queue with separate locks for enqueueing and dequeuing items and a depiction of a queue with two elements

```

1  from alloc import malloc
2
3  def SetObject() returns object:
4      object = malloc({})
5
6  def insert(s, v):
7      atomically !s |= {v}
8
9  def remove(s, v):
10     atomically !s -= {v}
11
12 def contains(s, v) returns present:
13     atomically present = v in !s

```

Figure 9.5: [\[code/setobj.hny\]](#) Specification of a concurrent set object

The problem with the original specification of the algorithm 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 modern environments. To get around this problem, the implementation in Figure 9.4 uses `atomic_load` and `atomic_store` from the `synch` module.

A queue has the nice property that only the head or the tail can be accessed. However, in many data structures it is necessary to “walk” the entire 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 9.5 gives the specification of a concurrent set object. `SetObject()` returns a pointer to a variable that contains an empty set, rather than returning an empty set *value*. As such, it is more like an object in an object-oriented language than like a value in its own right. Values can be added to the set object using `insert()` or deleted using `remove()`. Method `contains()` checks if a particular value is in the list. Figure 9.6 contains a simple (although not very thorough) test program to demonstrate the use of set objects.

Figure 9.7 implements a concurrent set object using an ordered linked list without duplicates. The list has two dummy “book-end” nodes with values `(-1, None)` and `(1, None)`. A value *v* is stored as `(0, v)`—note that for any value *v*, `(-1, None) < (0, v) < (1, None)` (because of lexicographical ordering of tuples). An invariant of the algorithm is that at any point in time the list is “valid,” starting with a `(-1, None)` node and ending with an `(1, None)` node.

Each node has a lock, a value, and *next*, a pointer to the next node (which is `None` for the `(1, None)` node to mark the end of the list). The `_find(lst, v)` helper method first finds and locks two consecutive nodes *before* and *after* such that *before*→*data.value* < (0, *v*) ≤ *after*→*data.value*. The method does so by performing something called *hand-over-hand locking*. It first locks the first node, which is the `(-1, None)` node. Then, iteratively, it obtains a lock on the next node and

```

1  from setobj import *
2
3  myset = SetObject()
4
5  def thread1():
6      insert(myset, 1)
7      let x = contains(myset, 1):
8          assert x
9
10 def thread2(v):
11     insert(myset, v)
12     remove(myset, v)
13
14 spawn thread1()
15 spawn thread2(0)
16 spawn thread2(2)

```

Figure 9.6: [\[code/setobj_test1.hny\]](#) Test code for set objects

release the lock on the last one, and so on, similar to climbing a rope hand-over-hand. Using `_find`, the `insert`, `remove`, and `contains` methods are fairly straightforward to implement.

Exercises

9.1 Add a method `head(q)` that returns the value at the head of the queue, or **None** if the queue is empty.

9.2 Add a method `contains(q, v)` to [Figure 9.1\(b\)](#) that checks to see if v is in queue q .

9.3 Add a method `length(q)` to [Figure 9.3](#) 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`.

9.4 Write a method `check(q)` that checks the integrity of the queue in [Figure 9.3](#). In particular, it should check the following integrity properties:

- If the list is empty, $q \rightarrow tail$ should be **None**. Otherwise, the last element in the linked list starting from $q \rightarrow head$ should equal $q \rightarrow head$. Moreover, $q \rightarrow tail \rightarrow next$ should be **None**;
- The length field that you added in [Exercise 9.3](#) 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()
```

```

1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def _node(v, n) returns node: # allocate and initialize a new list node
5      node = malloc({ .lock: Lock(), .value: v, .next: n })
6
7  def _find(lst, v) returns pair:
8      var before = lst
9      acquire(?before→lock)
10     var after = before→next
11     acquire(?after→lock)
12     while after→value < (0, v):
13         release(?before→lock)
14         before = after
15         after = before→next
16         acquire(?after→lock)
17     pair = (before, after)
18
19  def SetObject() returns object:
20     object = _node((-1, None), _node((1, None), None))
21
22  def insert(lst, v):
23     let before, after = _find(lst, v):
24         if after→value != (0, v):
25             before→next = _node((0, 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 == (0, v):
32             before→next = after→next
33             free(after)
34         else:
35             release(?after→lock)
36             release(?before→lock)
37
38  def contains(lst, v) returns present:
39     let before, after = _find(lst, v):
40         present = after→value == (0, v)
41         release(?after→lock)
42         release(?before→lock)

```

Figure 9.7: [code/setobj.linkedlist.hny] Implementation of a set of values using a linked list with fine-grained locking

- 9.5** Add a method `insert(q, v)` to [Figure 9.3](#) that inserts *v* at the head of queue *q*.
- 9.6** Add a method `remove(q, v)` to [Figure 9.3](#) that removes all occurrences of *v*, if any, from queue *q*.
- 9.7** The test program in [Figure 9.2](#) is not a thorough test program. Design and implement a test program for [Figure 9.2](#). Make sure you *test* the test program by trying it out against some buggy queue implementations. (You will learn more about testing concurrent programs in [Chapter 10](#).)
- 9.8** Add methods to the data structure in [Figure 9.7](#) 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. (All these should ignore the two end nodes.)
- 9.9** 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.
- 9.10** Create a binary tree that uses, instead of a single lock per tree, a lock for each node in the tree.

Chapter 10

Testing: Checking Behaviors

Testing is a way to increase confidence in the correctness of an implementation. [Figure 9.2](#) 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 9.3](#) 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.

Checking the behavior of a concurrent queue can be tricky. For example, suppose that, in some execution of [Figure 9.2](#), thread 1 invokes `put(?demoq, 1)` and then thread 2 invokes `put(?demoq, 2)` and then thread 3 invokes `get(?demoq)`. What value should thread 3 obtain? The answer is: It depends, or in other words, not enough information is provided. All of 1, 2, and **None** are possible outcomes. Certainly, if `put(?demoq, 1)` completes before `put(?demoq, 2)` is invoked and `put(?demoq, 2)` completes before `get(?demoq)` is invoked, then 1 is the only correct result. However, if `put(?demoq, 1)` is invoked first, and then `put(?demoq, 2)` is invoked before `put(?demoq, 1)` has completed, then those two operations are executing concurrently, and it might come to pass that 2 is enqueued before 1. In that case, if `get(?demoq)` is invoked after both those operations complete, both 1 and 2 are possible results, but **None** cannot be. However, if `get(?demoq)` is invoked before the other two operations complete, then **None** is a valid outcome as well, even if the other two operations were invoked before `get(?demoq)` was invoked.

The issue is that operations take time, and therefore operations can overlap. While locks reduce the amount to which operations can overlap, they do not completely eliminate it, and obtaining a lock takes time as well. Unlike critical sections, it turns out that checking the correct behavior of a queue is difficult to do with assertions alone because it is difficult to characterize the set of correct behaviors of a concurrent queue.

Behaviors say something about how we got to a state. The same state can be reached by multiple behaviors, and the behaviors are often an integral part of whether a program is correct or not. Just because a state satisfies some invariant—however important—does not mean that the state is valid given the sequence of operations. For example, a state in which the queue is empty is certainly a valid state in its own right, but if the last operation to get there was an enqueue operation, there must be a bug in the program. It can therefore be important to capture the behaviors. We could store behaviors in the state itself by adding what is known as a *history variable* that keeps track of all the operations. While this can be useful for correctness proofs, for model checking this approach

```

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

```

Figure 10.1: [\[code/queue_test_seq.hny\]](#) Sequential queue test

presents a problem: Introducing this additional state can lead to state explosion or even turn a finite model (a model with a finite number of states) into an infinite one.

Fortunately, if we have a specification of a queue, we can generate the set of correct behaviors of a queue. For simplicity, we will first check if the queue implementation in [Figure 9.3](#) meets the *sequential* queue specification in [Figure 9.1\(a\)](#). 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 *behaviors* or are *behaviorally equivalent*. This is called *differential testing*.

[Figure 10.1](#) presents a differential test program for sequences of up to `NOPS` queue operations. It maintains two queues:

- *specq*: the queue of the specification;
- *implq*: the queue of the implementation.

For each operation, the code first chooses whether to perform a `get` or `put` operation. In the case of a `put` operation, the code also chooses 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.

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. So, run a test program like this with a variety of queue implementations that have known

bugs in them and make sure that the test program finds them. Conversely, a test program may be broken in that it finds bugs that do not exist. In my experience, it is often harder to implement the test program than the algorithm that the test program tests.

As with any other test program, [Figure 10.1](#) 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 9.3](#) 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.

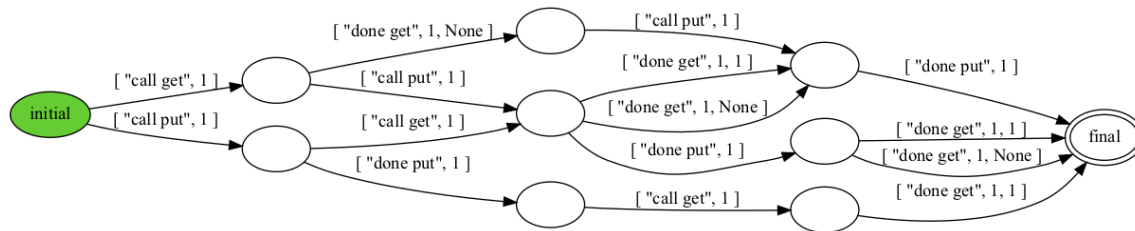
The test program also finds some common mistakes in using locks, such as forgetting to release a lock when the queue is empty, but it is not designed to find concurrency bugs in general. If you remove all `acquire()` and `release()` calls from [Figure 9.3](#), the test program will not (and should not) find any errors, but it would be an incorrect implementation of a concurrent queue.

We will now see how to test whether the queue implementation meets the *concurrent* queue specification or not. [Figure 9.1\(b\)](#) shows the concurrent queue specification. It is similar to the sequential specification in [Figure 9.1\(a\)](#) but makes all operations (except instantiation itself) atomic. This means that the effect of an operation appears to happen instantaneously sometime between invoking the operation and the operation completing, and in particular the effects cannot appear to overlap, while the operations themselves can. Testing the implementation of a concurrent queue specification is trickier than testing the implementation of a sequential one because there are many more scenarios to check.

We would like a way that—similar to the sequential test—systematically compares behaviors of the concurrent queue implementation with behaviors of the concurrent queue specification. But we cannot do this by composing the specification and the implementation and simply run the same test operations on both as we did before—concurrency make the operations non-deterministic and thus the specification and implementation of a single execution might produce different results, even if both are correct. Instead, we will create a test program that tries various concurrent combinations of queue operations, and run it twice: once against the specification of the concurrent queue and once against the implementation. In the second phase, we check if the behaviors obtained from running the implementation are also behaviors obtained from the specification.

[Figure 10.2](#) shows the test program. It starts `N_PUT` threads doing a `put` operation and `N_GET` threads doing a `get` operation. In case of a `put` operation, the thread enqueues its own name (which is provided as an argument to the thread). In order to capture the behaviors, each thread prints what operation it is about to perform, and afterwards it prints that the operation has completed (including the return value if any). The figure also shows the *behavior DFA* that captures the generated behaviors in case `N_PUT = N_GET = 1`. You can see that if `get()` completes before `put(1)` is invoked, then `get()` must return **None**. Vice versa, if `put(1)` completes before `get()` is invoked, then `get()` must return 1. Otherwise `get()` can return either **None** or 1 depending on how the two operations are serialized.

This is probably much like you would do if you were trying to find a bug in a program. [Figure 10.3](#) shows a Python implementation of the same test program. You can run it a bunch of times and manually check the output. There are, however, two problems with this approach. First, it is often



```

1  import queue
2
3  const N_PUT = 2
4  const N_GET = 2
5  q = queue.Queue()
6
7  def put_test(self):
8      print("call put", self)
9      queue.put(?q, self)
10     print("done put", self)
11
12  def get_test(self):
13      print("call get", self)
14      let v = queue.get(?q):
15          print("done get", self, v)
16
17  for i in {1..N_PUT}:
18      spawn put_test(i)
19  for i in {1..N_GET}:
20      spawn get_test(i)

```

Figure 10.2: [\[code/queue_btest2.hny\]](#) Concurrent queue test. The behavior DFA is for $N_PUT = N_GET = 1$.

```

1  import queue, threading, random
2
3  N_PUT = 2
4  N_GET = 2
5  q = queue.Queue()
6
7  def put_test(self):
8      print("call put", self)
9      q.put(self)
10     print("done put", self)
11
12  def get_test(self):
13     print("call get", self)
14     try:
15         v = q.get(block=False)
16         print("done get", self, v)
17     except queue.Empty:
18         print("done get empty", self)
19
20  for i in range(N_PUT):
21     threading.Thread(target=put_test, args=(i,)).start()
22  for i in range(N_GET):
23     threading.Thread(target=get_test, args=(i,)).start()

```

Figure 10.3: [[python/queue_btest2.py](#)] Python implementation of [Figure 10.2](#)

```

1  from synch import Lock, acquire, release
2
3  def Queue() returns empty:
4      empty = { .elements: [], .lock: Lock() }
5
6  def put(q, v):
7      acquire(?q→lock)
8      q→elements += [v,]
9      release(?q→lock)
10
11 def get(q) returns next:
12     acquire(?q→lock)
13     if q→elements == []:
14         next = None
15     else:
16         next = q→elements[0]
17     release(?q→lock)
18     acquire(?q→lock)
19     if q→elements != []:
20         del q→elements[0]
21     release(?q→lock)

```

Figure 10.4: [<code/queue-broken1.hny>] A correct sequential but not a correct concurrent queue implementation

difficult to check if the behaviors you find are correct ones, and it is easy to overlook problems in the output. Second, the test program may not check all possible behaviors.

Using Harmony, these problems can be avoided. One approach is to compare the two behavior DFAs by manual inspection:

```
$ harmony -c N_PUT=1 -c N_GET=1 code/queue_btest2.hny
$ harmony -c N_PUT=1 -c N_GET=1 -m queue=queue_lock code/queue_btest2.hny
```

You can try this for various `N_PUT` and `N_GET`, slowly increasing their values, although it gets increasingly harder to check by hand that the generated DFAs are the same. For example, run the test program against [Figure 10.4](#), which is an incorrect implementation of the concurrent queue specification in that it cannot handle concurrent `get` operations correctly. Try, for example:

```
$ harmony -c N_GET=2 -o x.png code/queue_btest2.hny
$ harmony -c N_GET=2 -o y.png -m queue=queue_broken1 code/queue_btest2.hny
```

and compare the finite state automata in `x.png` and `y.png`. If you look carefully, you will find that the two behaviors are not the same, although the differences are hard to spot. Harmony does not necessarily detect any problems itself this way. This is because the test program only outputs the behaviors—it does not check if they are correct.

Harmony does have a way to check the behaviors of one program against the behaviors of another. In particular, we want to check if the behaviors of the an implementation matches the behaviors of the specification. The following shows, for example, how to check the `queue_lock.hny` implementation on the command line:

```
$ harmony -o queue.hfa code/queue_btest2.hny
$ harmony -B queue.hfa -m queue=queue_lock code/queue_btest2.hny
```

The first command runs the `code/queue_btest2.hny` program (with the default 4 threads) and writes a representation of the output DFA in the file `queue.hfa`. The second command runs the same test program, but using the queue implementation in the file `code/queue_lock.hny`. Moreover, it reads the DFA in `queue.hfa` to check if every behavior of the second run of the test program is also a behavior of the first run.

Now try the same using the `queue_broken1.hny` implementation:

```
$ harmony -o queue.hfa code/queue_btest2.hny
$ harmony -B queue.hfa -m queue=queue_broken1 code/queue_btest2.hny
```

This way, Harmony will find a “behavior violation” that is hard to spot manually.

Exercises

10.1 [Figure 7.1](#) shows a specification of a lock. Write a program that checks the behaviors of lock implementations such as [Figure 8.2](#) and [Figure 8.3](#). That is, it should not rely on assertions such as in [Figure 5.2](#).

10.2 Write a Harmony program that checks if [Figure 9.7](#) satisfies the specification of [Figure 9.5](#) *sequentially*.

10.3 Write a Harmony program that checks if [Figure 9.7](#) satisfies the specification of [Figure 9.5](#) *concurrently*.

Chapter 11

Debugging

So, you wrote a Harmony program and Harmony reports a problem. 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 11.1 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 the test programs in the last chapter. Harmony does not detect any issues with the sequential test in Figure 10.1. (Run this using the `-m` flag like this: `harmony -m queueconc=queue_broken2 code/queue_test_seq.hny`) However, when we run the new queue code through the test in Figure 10.2, Harmony reports a safety violation (even without specifying a behavior). Figure 11.2 shows the command line to reproduce this and the first few lines of markdown output.

Before we go look at the details of what went wrong, we want to make sure that we generate the simplest scenario. So, first we want to explore what the smallest `N_PUT` and `N_GET` parameters are. With some experimentation, we find that `N_PUT = 2` and `N_GET = 1` generates a problem (`harmony -m queue=queue_broken2 -c N_PUT=2 -c N_GET=1 code/queue_btest2.hny`). Figure 11.3 shows the HTML output.

There is quite a bit of information in the HTML output. 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 `put_test(1)` (thread T1) was trying to dereference the address `?alloc$pool[0]["lock"]`.

The `alloc` module maintains a shared array `pool` that it uses for dynamic allocation. Apparently T1 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 under "Shared Variables", we see that in fact thread T2 allocated `pool[0]` in turn 2 (during `put_test(2)`), but T3 freed it in turn 4 (during `get_test(1)`).

```

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

```

Figure 11.1: [\[code/queue_broken2.hny\]](#) A buggy queue implementation

Summary: something went wrong in an execution

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
 - Line alloc/1: Initialize alloc\$pool to {:}
 - Line alloc/2: Initialize alloc\$next to 0
 - Line 5: Initialize q to { "lock": False, "next": None, "value": None }
 - **Thread terminated**
- Schedule thread T1: put_test(1)
 - Line 8: Print ["call put", 1]
 - Preempted in put_test(1) – put(?q, 1) – malloc({ "lock": False, "next": None, "value": 1 }) about to execute atomic section in line alloc/7
- Schedule thread T4: get_test(2)
 - Line 13: Print ["call get", 2]
 - Preempted in get_test(2) – get(?q) – acquire(?q["lock"]) about to execute atomic section in line synch/35
- Schedule thread T1: put_test(1) – put(?q, 1) – malloc({ "lock": False, "next": None, "value": 1 })
 - Line alloc/8: Initialize alloc\$pool[0] to { "lock": False, "next": None, "value": 1 }
 - Line alloc/10: Set alloc\$next to 1 (was 0)
 - Line synch/36: Set q["lock"] to True (was False)
 - Line queue/14: Set q["next"] to ?alloc\$pool[0] (was None)
 - Line synch/41: Set q["lock"] to False (was True)
 - Preempted in put_test(1) about to print ["done put", 1] in line 10
- Schedule thread T2: put_test(2)
 - Line 8: Print ["call put", 2]
 - Line alloc/8: Initialize alloc\$pool[1] to { "lock": False, "next": None, "value": 2 }
 - Line alloc/10: Set alloc\$next to 2 (was 1)
 - Line synch/36: Set q["lock"] to True (was False)
 - Line synch/41: Set q["lock"] to False (was True)
 - Preempted in put_test(2) – put(?q, 2) – acquire(?alloc\$pool[0]["lock"]) about to execute atomic section in line synch/35

...

Figure 11.2: Running [Figure 10.2](#) against [Figure 11.1](#)

Issue: Safety violation				Shared Variables										Output
Turn	Thread	Instructions Executed	PC	alloc\$next	alloc\$pool						q			
					0			1			lock	next	value	
					lock	next	value	lock	next	value				
1	T0: __init__()	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>												

[synch:35](#) atomically when not !binsema:

Threads			
ID	Status	Stack Trace	Stack Top
T0	terminated	__init__()	
T1	failed atomic read-only	put_test(1) self: 1 node: ?alloc\$pool[1], nq: ?alloc\$pool[0] acquire(?alloc\$pool[0]["lock"])binsema: ?alloc\$pool[0]["lock"]	
T2	runnable	put_test(2)	["done put", 2]
T3	runnable	get_test(1) self: 1 get(?q) next: 2 release(?q["lock"])binsema: ?q["lock"]	

Figure 11.3: HTML output of Figure 11.2 but for N_PUT=2 and N_GET=1

Looking at the stack traces in the bottom table, we can see that T3 was in the process of executing `release(?q.lock)` within `get(?q)`. T1 is currently executing `acquire(?alloc$pool[0].lock)` within `put(?q, 1)`, but `alloc$pool[0]` does not exist.

So, how did we get there? In the top we can see that the order of events was the following:

1. initialization completed, with `q` being `{ .lock: False, .next: None, .value: None }`;
2. thread T2 (`put.test(2)`) ran and finished executing `put(2)` (and is about to print ["done put", 2]). We can see that `q.next` now points to `alloc$pool[0]`, which the thread must have allocated. The contents is `{ .lock: False, .next: None, .value: 2 }`, as expected;
3. thread T1 (`put.test(1)`) started running, calling `put(?q, 1)`. We can see that T1 got as far as allocating a node, but it has not yet added the node to the end of the queue. T1 then tries to acquire `alloc$pool[0].lock`;
4. thread T3 (`get.test(1)`) started running, calling `get(?q)`. We can also see that T3 freed `pool[0]`, and is now releasing `q.lock`;
5. thread T1 resumes and tries to access `alloc$pool[0]`, which no longer exists (because T3 just freed it).

Clearly there was a race in which T1 was trying to lock `pool[0].lock` (which contained the node with the value 1) while T3 was freeing that very same node, and T1 lost the race. More precisely, T1 was executing `put(?q, 1)`, when T3 preempted it with `get(?q)` and removed the node that T1 was trying to access. But why did the locks not prevent this?

It is time to start stepping through the code that has been executed before this happened. This is sometimes known as *reverse debugging*. In fact, Harmony allows you to step through an execution forwards and backwards. In this case, we first want to see what T2 is doing. You can click its first (top-left) orange box to time-travel to that part in the execution. The boxes are color-coded: Each method has its own color. Now by hitting the `<return>` key repeatedly, we can quickly skip through the code, line by line. T1 first calls `put(?q, 1)` and then allocates a new node initialized with a lock. Keep stepping until it executes `nq = q`. Select `<return>` once more and inspect the state of T1 in the lower-right corner. You can see that variable `nq` is initialized to `?q`. T1 then enters into the **while** loop and tries to acquire `nq→lock`. This succeeds, and next T1 executes `let n = nq→next`. Now `n = ?alloc$pool[0]`, which is not **None**. T1 then releases `nq→lock` (`nq` points to `q`). T1 then sets `nq` to `n`, which is still `alloc$pool[0]`. Finally, it calls `acquire(?nq→lock)`. But before it can complete that operation, T3 runs next.

T3 chooses "get" and then goes on to invoke `get(?q)`. This first successfully acquires `q→lock`. T3 then finds out that `q→next` points to `alloc$pool[0]`. T3 sets `node` to `alloc$pool[0]` as well and sets `q→next` to `node→next`. T3 sets the method result `next` to `node→value` (which is 2) and then frees `node`. This is where the problem is—T1 is about to acquire the lock in that same node.

To fix the code without changing the data structure, we can use hand-over-hand locking ([Chapter 9](#)). [Figure 11.4](#) shows an implementation that 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() returns empty:
5      empty = { .next: None, .value: None, .lock: Lock() }
6
7  def put(q, v):
8      var nq = q
9      let node = malloc({ .next: None, .value: v, .lock: Lock() }):
10         acquire(?nq→lock)
11         var n = nq→next
12         while n != None:
13             acquire(?n→lock)
14             release(?nq→lock)
15             nq = n
16             n = n→next
17         nq→next = node
18         release(?nq→lock)
19
20 def get(q) returns next:
21     acquire(?q→lock)
22     if q→next == None:
23         next = None
24     else:
25         let node = q→next:
26             acquire(?node→lock)
27             q→next = node→next
28             next = node→value
29             release(?node→lock)
30             free(node)
31     release(?q→lock)

```

Figure 11.4: [\[code/queue_fix.hny\]](#) Queue implementation with hand-over-hand locking

Chapter 12

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 waits until there is no other thread in the critical section.

A thread that is waiting for a condition without changing shared variables is considered *blocked*. In other words, a blocked thread can only change shared variables if some other thread changes them first. A spinlock to protect a critical section is a good example: A thread that is waiting for a spinlock does not change the state of the lock variable until it acquires the lock.

Sometimes it is useful for a thread to wait for additional conditions besides the critical section being unoccupied. For example, when dequeuing from an empty shared queue, the thread may want to block until the queue is non-empty. The alternative to blocking would be *active busy waiting* (or also just *busy waiting*) where the thread repeatedly tries to dequeue an item until it is successful.

A thread that is active 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. Doing so wastes CPU cycles and adds contention to queue access.

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*.

12.1 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].

```

1  def RWlock() returns lock:
2      lock = { .nreaders: 0, .nwriters: 0 }
3
4  def read_acquire(rw):
5      atomically when  $rw \rightarrow nwriters == 0$ :
6           $rw \rightarrow nreaders += 1$ 
7
8  def read_release(rw):
9      atomically  $rw \rightarrow nreaders -= 1$ 
10
11 def write_acquire(rw):
12     atomically when  $(rw \rightarrow nreaders == 0)$  and  $(rw \rightarrow nwriters == 0)$ :
13          $rw \rightarrow nwriters = 1$ 
14
15 def write_release(rw):
16     atomically  $rw \rightarrow nwriters = 0$ 

```

Figure 12.1: [\[code/rwlock.hny\]](#) Specification of reader/writer locks

A reader/writer lock is an object whose abstract state contains two integer counters (see [Figure 12.1](#)):

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

satisfying the following invariant: $(nreaders \geq 0 \wedge nwriters = 0) \vee (nreaders = 0 \wedge nwriters = 1)$.

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.

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 the lock for reading and the same for writing.

[Figure 12.2](#) shows how reader/writer locks operations may be tested. A problem with this test is that it does not find a problem with an implementation like the one in [Figure 12.3](#). This implementation implements a reader/writer lock as an ordinary lock, and thus lets only one thread in the critical section at a time. In some sense, the implementation is correct because it satisfies the invariant, but it is not a desirable implementation because it does not allow multiple readers to enter the critical section at the same time. For a case like this one, it is better to compare behaviors between the specification and the implementation.

```

1  import rwlock
2
3  nreaders = nwriters = 0
4  invariant ((nreaders >= 0) and (nwriters == 0)) or
5             ((nreaders == 0) and (nwriters == 1))
6
7  const NOPS = 4
8
9  rw = rwlock.RWlock()
10
11 def thread():
12     while choose({ False, True }):
13         if choose({ "read", "write" }) == "read":
14             rwlock.read_acquire(rw)
15             atomically nreaders += 1
16             atomically nreaders -= 1
17             rwlock.read_release(rw)
18         else:                                # write
19             rwlock.write_acquire(rw)
20             atomically nwriters += 1
21             atomically nwriters -= 1
22             rwlock.write_release(rw)
23
24 for i in {1..NOPS}:
25     spawn thread()

```

Figure 12.2: [\[code/rwlock_test1.hny\]](#) Test code for reader/writer locks

```

1  import synch
2
3  def RWlock() returns lock:
4      lock = synch.Lock()
5
6  def read_acquire(rw):
7      synch.acquire(rw)
8
9  def read_release(rw):
10     synch.release(rw)
11
12 def write_acquire(rw):
13     synch.acquire(rw)
14
15 def write_release(rw):
16     synch.release(rw)

```

Figure 12.3: [[code/rwlock_cheat.hny](https://code.rwlock_cheat.hny)] "Cheating" reader/writer lock

Figure 12.4 is the same test as Figure 12.2 but prints identifying information before and every lock operation. Now we can compare behaviors as follows:

```

$ harmony -o rw.hfa -cNOPS=2 code/rwlock_btest.hny
$ harmony -B rw.hfa -cNOPS=2 -m rwlock=rwlock_cheat code/rwlock_btest.hny

```

The second command will print a warning that there are behaviors in the specification that are not achieved by the implementation.

Figure 12.5 illustrates an implementation of a reader/writer lock that uses active busy waiting. The solution is careful only to access the variables while holding a lock and at the same time careful to release the lock regularly to allow other threads to change the state. Nonetheless, it is an undesirable solution, as it wastes CPU cycles. Harmony complains about this solution.

12.2 Bounded Buffer

A *bounded buffer* is a queue with the usual `put/get` interface, but implemented using a buffer of a certain maximum length. If the buffer is full, an enqueueer must wait; if the buffer is empty, a dequeuer must wait. Figure 12.6 specifies a bounded buffer. It is similar to the implementation in Figure 9.1(b) but adds checking for bounds. Coming up with a good implementation 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


```

1  import rlock
2
3  const NOPS = 3
4
5  rw = rlock.RWlock()
6
7  def thread(self):
8      while choose({ False, True }):
9          if choose({ "read", "write" }) == "read":
10             print(self, "enter ra")
11             rwlock.read_acquire(?rw)
12             print(self, "exit ra")
13
14             print(self, "enter rr")
15             rwlock.read_release(?rw)
16             print(self, "exit rr")
17         else: # write
18             print(self, "enter wa")
19             rwlock.write_acquire(?rw)
20             print(self, "exit wa")
21
22             print(self, "enter wr")
23             rwlock.write_release(?rw)
24             print(self, "enter wr")
25
26  for i in {1..NOPS}:
27      spawn thread(i)

```

Figure 12.4: [\[code/rwlock_btest.hny\]](#) A behavioral test of reader/writer locks

```

1  from synch import Lock, acquire, release
2
3  def RWlock() returns lock:
4      lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6  def read_acquire(rw):
7      acquire(?rw→lock)
8      while rw→nwriters > 0:
9          release(?rw→lock)
10         acquire(?rw→lock)
11         rw→nreaders += 1
12         release(?rw→lock)
13
14  def read_release(rw):
15      acquire(?rw→lock)
16      rw→nreaders -= 1
17      release(?rw→lock)
18
19  def write_acquire(rw):
20      acquire(?rw→lock)
21      while rw→nreaders > 0 or rw→nwriters > 0:
22          release(?rw→lock)
23          acquire(?rw→lock)
24      rw→nwriters = 1
25      release(?rw→lock)
26
27  def write_release(rw):
28      acquire(?rw→lock)
29      rw→nwriters = 0
30      release(?rw→lock)

```

Figure 12.5: [[code/rwlock.busy.hny](http://code.rwlock.busy.hny)] Busy waiting reader/writer lock

```

1  def BoundedBuffer(size) returns buffer:
2      buffer = { .buffer: [], .size: size }
3
4  def put(bb, v):
5      atomically when len(bb→buffer) < bb→size:
6          bb→buffer += [v,]
7
8  def get(bb) returns next:
9      atomically when bb→buffer != []:
10         next = bb→buffer[0]
11         del bb→buffer[0]

```

Figure 12.6: [\[code/boundedbuffer.hny\]](#) Bounded buffer specification

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.

Unlike an ordinary queue, where queues can grow arbitrarily, bounded buffers provide *flow control*: if the consumer runs faster than the producer (or producers), it will automatically block until there are new requests. Similarly, if the producers add requests at a rate that is higher than the consumers can deal with, the producers are blocked. While a buffer of size 1 already provides those properties, a larger buffer is able to deal with short load spikes without blocking producers.

Exercises

12.1 Optimize your solutions to [Exercise 9.2](#) to use reader/writer locks.

Chapter 13

Condition Variables

The last chapter introduced conditional waiting, but the implementations provided were unsatisfactory, either because they unnecessarily restricted behaviors or because they were inefficient. *Condition variables* are a synchronization primitive designed to allow for general and efficient solutions to conditional waiting problems.

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. The Mesa language also incorporated a version of condition variables. There are two main classes of condition variable semantics. [Appendix I](#) describes *Hoare condition variables*. However, most programming languages today support the semantics of Mesa’s condition variables. Below, we will use the semantics that Mesa provides.

A condition variable works in conjunction with a lock, often called *mutex* (for mutual exclusion) in this context. A thread p that holds the mutex can invoke the `wait` operation on a condition variable. This temporarily releases the mutex and places thread p on a queue associated with the condition variable so p no longer executes. Another thread can `notify` the condition variable. If the condition variable’s queue is non-empty, this operation removes a thread q from the queue (according to some scheduling policy) and makes q runnable again. Thread q first re-acquires the mutex before resuming from the `wait` operation it invoked. If, on the other hand, the condition variable’s queue is empty, the `notify` operation is a no-op.

Condition variables 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 `notify_all` (a.k.a. `broadcast`) notifies *all* threads that are waiting on a condition.

We demonstrate Mesa condition variables using an implementation of reader/writer locks. [Figure 13.1](#) shows the code, which, the reader should note, is similar to the busy waiting solution in [Figure 12.5](#). *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

```

1  from synch import *
2
3  def RWlock() returns lock:
4      lock = {
5          .nreaders: 0, .nwriters: 0, .mutex: Lock(),
6          .r_cond: Condition(), .w_cond: Condition()
7      }
8
9  def read_acquire(rw):
10     acquire(?rw→mutex)
11     while rw→nwriters > 0:
12         wait(?rw→r_cond, ?rw→mutex)
13     rw→nreaders += 1
14     release(?rw→mutex)
15
16  def read_release(rw):
17     acquire(?rw→mutex)
18     rw→nreaders -= 1
19     if rw→nreaders == 0:
20         notify(?rw→w_cond)
21     release(?rw→mutex)
22
23  def write_acquire(rw):
24     acquire(?rw→mutex)
25     while rw→nreaders > 0 or rw→nwriters > 0:
26         wait(?rw→w_cond, ?rw→mutex)
27     rw→nwriters = 1
28     release(?rw→mutex)
29
30  def write_release(rw):
31     acquire(?rw→mutex)
32     rw→nwriters = 0
33     notify_all(?rw→r_cond)
34     notify(?rw→w_cond)
35     release(?rw→mutex)

```

Figure 13.1: [\[code/rwlock_cv.hny\]](#) Reader/Writer Lock using Mesa-style condition variables

`wait` containing the negation of the condition that the thread is waiting for. Many implementations 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 condition variable has not been notified. As a rule of thumb, one should always be able to replace `wait` by `release` followed by `acquire`. This turns the solution into a busy-waiting one, inefficient but still correct.

In `read_release`, `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 `notify_all(?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. Awakening both readers and writers is ok because both execute `wait` within a `while` loop, re-checking the condition that they are waiting for. So, if both types 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 for each condition.)

When using condition variables, you have to be careful to invoke `notify` or `notify_all` in the right places. Much of the complexity of programming with condition variables is in figuring out when to invoke `notify` and when to invoke `notify_all`. As a rule of thumb: Be conservative—it is better to wake up too many threads than too few. In case of doubt, use `notify_all`. For example, in [Figure 13.1](#) it is ok, if inefficient, to replace the `notify` operations with `notify_all`, but it would be incorrect to replace the `notify_all` operation by `notify`. 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 `notify_all` 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](#)]. They have been adopted by all major programming languages. 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 `notify_all`. In addition, Java supports explicit allocations of locks and condition variables. In Python, locks and condition variables must be explicitly declared. The `with` statement simplifies acquiring and releasing 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 13.2](#) 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.

By the way, in Harmony a bag is usually represented by a dictionary that maps the elements of the bag to their multiplicities. For example, the value `{ .a: 2, .b: 3 }` represents a bag with two copies of `.a` and three copies of `.b`. The `bags` module ([Section B.3](#)) contains a variety of handy functions on bags.

Method `wait` adds the context of the thread—used as a unique identifier for the thread—to the bag, incrementing the number of threads in the bag with the same context. The Harmony `save` expression ([Section C.3](#)) returns a tuple containing a value (in this case `()`) and the context of the thread. `wait` then loops until that count is restored to the value that it had upon entry to `wait`. Method `notify` removes an arbitrary context from the bag, allowing one of the threads with that

```

1  def Condition() returns condition:
2      condition = bags.empty()
3
4  def wait(c, lk):
5      var cnt = 0
6      let _, ctx = save():
7          atomically:
8              cnt = bags.multiplicity(!c, ctx)
9              !c = bags.add(!c, ctx)
10             !lk = False
11             atomically when (not !lk) and (bags.multiplicity(!c, ctx) <= cnt):
12                 !lk = True
13
14  def notify(c):
15      atomically if !c != bags.empty():
16          !c = bags.remove(!c, bags.bchoose(!c))
17
18  def notify_all(c):
19      !c = bags.empty()

```

Figure 13.2: [\[modules/synch.hny\]](#) Implementation of condition variables in the `synch` module

context to resume and re-acquire the lock associated with the monitor. `notify_all` empties out the entire bag, allowing all threads in the bag to resume.

Exercises

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

13.2 Implement a “try lock” module using Mesa condition variables (see also [Exercise 8.3](#)). 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.

13.3 Implement a thread-safe *GPU allocator* by modifying [Figure 13.3](#). 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 GPU available. Method `gpuRelease(gpu)` releases the given GPU. It never needs to block.

```

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

```

Figure 13.3: [\[code/gpu.hny\]](#) A thread-unsafe GPU allocator

13.4 Bonus problem: [Figure 13.4](#) shows an iterative implementation of the quicksort algorithm, and [Figure 13.5](#) an accompanying test program. Shared variable *testqs.arr* keeps track of both the array to be sorted and a todo list containing 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.

13.5 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. 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.

13.6 Extend the solution to [Exercise 13.5](#) by implementing the requirement that at most *n* cars are allowed on the bridge. Add *n* as an argument to **OLBlock**.


```

1  def Qsort(arr) returns state:
2      state = { .arr: arr, .todo: { (0, len(arr) - 1) } }
3
4  def swap(p, q):          # swap !p and !q
5      !p, !q = !q, !p
6
7  def partition(qs, lo, hi) returns pivot:
8      pivot = lo
9      for i in {lo..hi - 1}:
10         if qs→arr[i] <= qs→arr[hi]:
11             swap(?qs→arr[pivot], ?qs→arr[i])
12             pivot += 1
13         swap(?qs→arr[pivot], ?qs→arr[hi])
14
15  def sortrange(qs, range):
16      let lo, hi = range let pivot = partition(qs, lo, hi):
17          if (pivot - 1) > lo:
18              qs→todo |= { (lo, pivot - 1) }
19          if (pivot + 1) < hi:
20              qs→todo |= { (pivot + 1, hi) }
21
22  def sort(qs) returns sorted_list:
23      while qs→todo != {}:
24          let range = choose(qs→todo):
25              qs→todo -= { range }
26              sortrange(qs, range)
27      sorted_list = qs→arr

```

Figure 13.4: [[code/qsort.hny](#)] Iterative qsort() implementation

```

1  import qsort, bags
2
3  const NITEMS = 4
4
5  a = [ choose({1..NITEMS}) for i in {1..choose({1..NITEMS})} ]
6  testqs = qsort.Qsort(a)
7  sa = qsort.sort(?testqs)
8  assert all(sa[i - 1] <= sa[i] for i in {1..len(sa)-1}) # sorted?
9  assert bags.fromList(a) == bags.fromList(sa) # is it a permutation?

```

Figure 13.5: [[code/qsorttest.hny](#)] Test program for Figure 13.4

Chapter 14

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 says that *some* thread eventually can enter, but it does not prevent a scenario such as the following in which some thread never is able to enter. 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.

Peterson’s Algorithm (Figure 5.6) 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. Ticket locks (Figure 8.3) are also free from starvation.

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 Figure 13.1. 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

```

3  from synch import *
4
5  def RWlock() returns lock:
6      lock = {
7          .nreaders: 0, .nwriters: 0, .mutex: Lock(),
8          .r_waiting: 0, .r_cleared: 0, .w_waiting: 0, .w_cleared: 0,
9          .r_cond: Condition(), .w_cond: Condition()
10     }
11
12  def read_acquire(rw):
13      acquire(?rw→mutex)
14      if (rw→nwriters == 0) and (rw→w_waiting == 0):
15          assert rw→r_waiting == 0
16          rw→nreaders += 1
17      else:
18          rw→r_waiting += 1
19          while rw→r_cleared == 0:
20              wait(?rw→r_cond, ?rw→mutex)
21          rw→r_cleared -= 1
22      assert rw→nreaders > 0
23      assert rw→nwriters == 0
24      release(?rw→mutex)
25
26  def read_release(rw):
27      acquire(?rw→mutex)
28      assert rw→nreaders > 0
29      assert rw→nwriters == 0
30      rw→nreaders -= 1
31      if (rw→nreaders == 0) and (rw→w_waiting > 0):
32          rw→w_cleared = rw→nwriters = 1
33          rw→w_waiting -= 1
34          notify(?rw→w_cond)
35      release(?rw→mutex)

```

Figure 14.1: [\[code/rwlock_cv_fair.hny\]](#) Reader/Writer Lock implementation addressing fairness (part 1)

```

39 def write_acquire(rw):
40     acquire(?rw→mutex)
41     if (rw→nreaders == 0) and (rw→nwriters == 0):
42         assert rw→r_waiting == rw→w_waiting == 0
43         rw→nwriters = 1
44     else:
45         rw→w_waiting += 1
46         while rw→w_cleared == 0:
47             wait(?rw→w_cond, ?rw→mutex)
48         rw→w_cleared = 0
49     assert rw→nreaders == 0
50     assert rw→nwriters == 1
51     release(?rw→mutex)
52
53 def write_release(rw):
54     acquire(?rw→mutex)
55     assert rw→nreaders == 0
56     assert rw→nwriters == 1
57     if rw→r_waiting > 0:
58         rw→nwriters = 0
59         rw→r_cleared = rw→nreaders = rw→r_waiting
60         rw→r_waiting = 0
61         notify_all(?rw→r_cond)
62     elif rw→w_waiting > 0:
63         rw→w_waiting -= 1
64         rw→w_cleared = 1
65         notify(?rw→w_cond)
66     else:
67         rw→nwriters = 0
68     release(?rw→mutex)

```

Figure 14.2: [\[code/rwlock_cv_fair.hny\]](#) Reader/Writer Lock implementation addressing fairness (part 2)

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.

[Figure 14.1](#) and [Figure 14.2](#) present a fair implementation of a read/writer lock. When there is contention between readers and writers, 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 scheduling the threads themselves.

Besides the number of readers and writers in the critical section, the implementation keeps track of the number of readers and writers that are waiting to enter the critical section, and the number of readers and writers that are cleared to enter the critical section.

Starting with `read_acquire` in [Figure 14.1](#), the thread first checks to see if there are no writers in the critical section *and* there are no writers waiting to enter the critical section (Line 14). If so, the thread can enter the critical section. Otherwise it waits until it is cleared to enter the critical section. `read_release` checks to see if the thread is the last one to leave the critical section *and* there are writers waiting to enter. If so, it clears one of the writers.

`write_acquire` in [Figure 14.2](#) first checks to see that there are no other threads in the critical section. If so, the thread enters the critical section. Otherwise, the thread waits until it is cleared to enter. `write_release` first checks to see if there are readers waiting to enter. If so, it clears all the readers. If not, it checks to see if there are writers waiting to enter. If so, it clears one of the writers.

Exercises

14.1 Write a fair solution to the one-lane bridge problem of [Exercise 13.5](#).

Chapter 15

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 philosophers 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—if they wish—leave the table when they are not using any forks.

[Figure 15.1](#) 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?

There are four conditions that must hold for deadlock to occur, sometimes known as the *Coffman Conditions* [[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 because many resources allow only one process to use it at a time (although, for some resources it is possible, as demonstrated in [Chapter 21](#)). Havender proposed the following techniques that avoid the remaining three conditions [[Hav68](#)]:

```

1  from synch import Lock, acquire, release
2
3  const N = 5
4
5  forks = [Lock(),] * N
6
7  def diner(which):
8      let left, right = (which, (which + 1) % N):
9          while choose({ False, True }):
10             acquire(?forks[left])
11             acquire(?forks[right])
12             # dine
13             release(?forks[left])
14             release(?forks[right])
15             # think
16
17  for i in {0..N-1}:
18      spawn diner(i)

```

Figure 15.1: [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, so we re-implement the Dining Philosophers using condition variables that allow one to wait for arbitrary application-specific conditions.

Figure 15.2 demonstrates how this might be done. The implementation uses a boolean for each fork indicating if the fork has been taken. It also uses a condition variable for each pair of forks, or, equivalently, for each philosopher. Each philosopher that is about to eat waits on its condition variable if either the left or right fork is already taken. When a philosopher is done eating, it notifies both of its neighbors.

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

```

1  from synch import *
2
3  const N = 5
4
5  mutex = Lock()
6  forks = [False,] * N
7  conds = [Condition(),] * N
8
9  def diner(which):
10     let left, right = (which, (which + 1) % N):
11         while choose({ False, True }):
12             acquire(?mutex)
13             while forks[left] or forks[right]:
14                 wait(?conds[which], ?mutex)
15             forks[left] = forks[right] = True
16             release(?mutex)
17             # dine
18             acquire(?mutex)
19             forks[left] = forks[right] = False
20             notify(?conds[(which - 1) % N])
21             notify(?conds[(which + 1) % N])
22             release(?mutex)
23             # think
24
25     for i in {0..N-1}:
26         spawn diner(i)

```

Figure 15.2: [\[code/DinersCV.hny\]](#) Dining Philosophers that grab both forks at the same time

the case of the philosophers, they could prevent deadlock by always picking up the lower numbered fork before the higher numbered fork, like so:

```
1  if left < right:
2      synch.acquire(?forks[left])
3      synch.acquire(?forks[right])
4  else:
5      synch.acquire(?forks[right])
6      synch.acquire(?forks[left])
```

or, equivalently, like so:

```
1  synch.acquire(?forks[min(left, right)])
2  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 15.3](#) demonstrates this concept. It uses a *counting semaphore* to restrict the number of diners at any time to four. A counting semaphore is a resource that can be procured a given number of times. It is supported by the `synch` module. The P or “procure” operation acquires a counting semaphore, blocking if too many threads have procured it already. The V or “vacate” operation releases 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 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 usually impractical.

Exercises

15.1 [Figure 15.4](#) 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?

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

15.3 Add an invariant that checks that the total of the balances never changes. Note that this predicate only holds if none of the locks are held. You can use the `held(lk)` method in the `synch` module to check if a lock is held or not.

```

1  from synch import *
2
3  const N = 5
4
5  forks = [Lock(),] * N
6  sema = Semaphore(N - 1)    # can be procured up to N-1 times
7
8  def diner(which):
9      let left, right = (which, (which + 1) % N):
10         while choose({ False, True }):
11             P(?sema)          # procure counting semaphore
12             acquire(?forks[left])
13             acquire(?forks[right])
14             # dine
15             release(?forks[left])
16             release(?forks[right])
17             V(?sema)          # vacate counting semaphore
18             # think
19
20  for i in {0..N-1}:
21      spawn diner(i)

```

Figure 15.3: [\[code/DinersAvoid.hny\]](#) Dining Philosophers solutions that avoids getting into a deadlock by allowing at most $N-1$ philosophers to start eating at a time

```

1  from synch import Lock, acquire, release
2
3  const MAX_BALANCE = 2
4  const N_ACCOUNTS = 2
5  const N_THREADS = 2
6
7  accounts = [ { .lock: Lock(), .balance: choose({0..MAX_BALANCE})}
8               for i in {1..N_ACCOUNTS} ]
9
10 def transfer(a1, a2, amount) returns success:
11     acquire(?accounts[a1].lock)
12     if amount <= accounts[a1].balance:
13         accounts[a1].balance -= amount
14         acquire(?accounts[a2].lock)
15         accounts[a2].balance += amount
16         release(?accounts[a2].lock)
17         success = True
18     else:
19         success = False
20     release(?accounts[a1].lock)
21
22 def thread():
23     let a1 = choose({0..N_ACCOUNTS-1})
24     let a2 = choose({0..N_ACCOUNTS-1} - { a1 }):
25         transfer(a1, a2, choose({1..MAX_BALANCE}))
26
27 for i in {1..N_THREADS}:
28     spawn thread()

```

Figure 15.4: [\[code/bank.hny\]](#) Bank accounts

Chapter 16

Actors and Message Passing

So far we have focused on using critical sections to achieve synchronization between threads accessing shared data structures. Some programming languages favor a different way using so-called *actors* [HBS73, Agh86]. Actors are threads that have only private memory and communicate through *message passing*. See Figure 16.1 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 as well.

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* (a.k.a. *blocking queues*) for messaging. Each thread would have one such queue for receiving messages. Dequeueing from an empty synchronized queue blocks the thread until another thread enqueues a message on the queue.

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

- `Queue()` returns an empty queue;

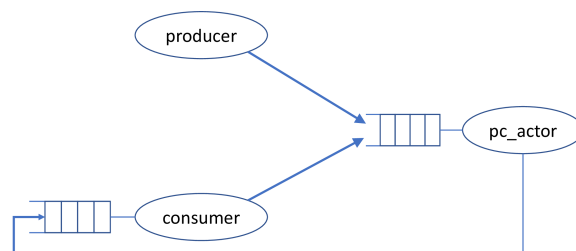


Figure 16.1: Depiction of three actors. The producer does not receive messages.

```

1  import synch
2
3  const NCLIENTS = 3
4
5  server_queue = synch.Queue()
6
7  def server():
8      var counter = 0
9      while True:
10         let q = synch.get(?server_queue): # await request
11         synch.put(q, counter)           # send response
12         counter += 1
13
14  def client(client_queue):
15      synch.put(?server_queue, client_queue)    # send request
16      let response = synch.get(client_queue):    # await response
17      print(response)
18
19  spawn eternal server()
20
21  alice_queue = synch.Queue()
22  spawn client(?alice_queue)
23  bob_queue = synch.Queue()
24  spawn client(?bob_queue)
25  charlie_queue = synch.Queue()
26  spawn client(?charlie_queue)

```

Figure 16.2: [\[code/counter.hny\]](#) An illustration of the actor approach

- `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 16.2 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. The server never terminates, so it is spawned with the keyword **eternal** to suppress non-terminating state warnings. 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.

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

16.1 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 17

Barrier Synchronization

Barrier synchronization is a technique that comes up in high-performance parallel computing. The Harmony model checker uses it internally to synchronize its worker threads. 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, when there is a barrier in concurrent code, all threads must complete the work before the barrier before they can start the work after the barrier. Therefore, threads that reach the barrier must wait until all threads reached the barrier.

Like locks, barriers can be re-used. Threads may execute in rounds, and between each round threads wait at the barrier until all threads have completed the previous round. 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.

The interface to a barrier is as follows:

- `b = Barrier(n)`: initialize a barrier `b` for a collection of `n` threads;
- `bwait(?b)`: wait until all threads have reached the barrier.

Figure 17.1 demonstrates the use of barriers in a parallel dot product computation. In this example, there are two vectors `vec1` and `vec2` that need to be multiplied. Because the vectors may be very large, they are split among `NWORKERS` threads, each computing a partial sum. After reaching the barrier, they know that each worker is done and print the sum. To see why the barrier is needed, you can comment out Line 19 and run it through harmony.

Before we show how one might specify and implement a barrier in Harmony, we present a test program for barriers in Figure 17.2. The work that a thread T_i does before the barrier is captured in the counter `before[i]`, while the work that a thread does after the barrier is captured in counter `after[i]`. If the barrier works correctly, then it must be the case that a worker cannot increment `after[i]` in a round until all workers have incremented `before[i]`. Therefore the following invariant must hold: $\max(\text{before}) \geq \min(\text{after})$.

How can one specify a barrier? A complication here is that the interface (similar to interfaces from mainstream programming languages) does not identify *which* thread invokes `bwait()`, nor how many times the thread has invoked `bwait()`. For example, one might be tempted to specify the

```

1  import barrier
2
3  const NWORKERS = 2
4
5  vec1 = [ 1, 2, 3, 4 ]
6  vec2 = [ 5, 6, 7, 8 ]
7  barr = barrier.Barrier(NWORKERS)
8  output = [ 0, ] * NWORKERS
9
10 def split(self, v) returns x:
11     x = (self * len(v)) / NWORKERS
12
13 def dotproduct(self, v1, v2):
14     assert len(v1) == len(v2)
15     var total = 0
16     for i in { split(self, v1) .. split(self + 1, v1) - 1 }:
17         total += v1[i] * v2[i]
18     output[self] = total
19     barrier.bwait(?barr)
20     print sum(output)
21
22 for i in { 0 .. NWORKERS - 1 }:
23     spawn dotproduct(i, vec1, vec2)

```

Figure 17.1: [\[code/barrier_demo.hny\]](#) Parallel dot product using barrier implementation


```

1  import barrier
2
3  const NTHREADS = 3
4  const NROUNDS = 4
5
6  barr = barrier.Barrier(NTHREADS)
7  before = after = [0,] * NTHREADS
8
9  invariant min(before) >= max(after)
10
11 def thread(self):
12     for _ in { 1 .. NROUNDS }:
13         before[self] += 1
14         barrier.bwait(?barr)
15         after[self] += 1
16
17 for i in { 0 .. NTHREADS - 1 }:
18     spawn thread(i)

```

Figure 17.2: [[code/barrier_test.hny](#)] Testing the barrier synchronization interface

```

1  def Barrier(required) returns barrier:
2      barrier = { .required: required, .n: 0 }
3
4  def bwait(b):
5      atomically b→n += 1
6      atomically await b→n == b→required

```

Figure 17.3: [[code/barrier_once.hny](#)] A single-use barrier specification

```

1  def Barrier(required) returns barrier:
2      barrier = { .required: required, .n: 0 }
3
4  def bwait(b):
5      atomically:
6           $b \rightarrow n \ += 1$ 
7          if  $b \rightarrow n == b \rightarrow required$ :
8               $b \rightarrow n = 0$ 
9      atomically await  $b \rightarrow n == 0$ 

```

Figure 17.4: [\[code/barrier_broken.hny\]](#) An incorrect barrier specification

```

1  def Barrier(required) returns barrier:
2      barrier = { .required: required, .n: [0, 0] }
3
4  def turnstile(b, i):
5      atomically:
6           $b \rightarrow n[i] \ += 1$ 
7          if  $b \rightarrow n[i] == b \rightarrow required$ :
8               $b \rightarrow n[1 - i] = 0$ 
9      atomically await  $b \rightarrow n[i] == b \rightarrow required$ 
10
11 def bwait(b):
12     turnstile(b, 0)
13     turnstile(b, 1)

```

Figure 17.5: [\[code/barrier_double.hny\]](#) Barrier specification using the double turnstile approach

barrier as in [Figure 17.3](#). It is implemented as a *turnstile* that lets one thread through into a waiting area until a certain number *required* is reached (which is set to the number of threads). At that time that threads in the waiting area are released. The number of threads that have passed the turnstile is maintained in *n*.

Clearly, we have to have a way to reset *n*, but which thread should be responsible for it? One option is to let the last thread that reaches the barrier do it. [Figure 17.4](#) illustrates this. Before you read on, can you see what the problem is?

The barrier implementation is correct for single use, and all threads wait until the last thread has reached the barrier. However, now suppose one of those threads goes on and gets back to `bwait()` for the next round. The thread will find that $n = 0$ and will increment n to 1. The thread then starts waiting until $n = required$. However, so will all other threads, because n is no longer 0. The crux of the problem is that the barrier state cannot distinguish between the threads that are waiting for the other threads to arrive and the threads that have been cleared to leave.

```

1  def Barrier(required) returns barrier:
2      barrier = { .required: required, .n: 0, .color: 0 }
3
4  def bwait(b):
5      var color = None
6      atomically:
7          color = b→color
8          b→n += 1
9          if b→n == b→required:
10             b→color ^= 1
11             b→n = 0
12      atomically await b→color != color

```

Figure 17.6: [\[code/barrier.hny\]](#) Barrier specification

This insight inspires the solution in [Figure 17.5](#), which is sometimes called the *double turnstile approach*. Each turnstile is as before: It counts the number of threads that have arrived at the turnstile. We shall call the turnstiles red (0) and green (1). The threads first go through the red turnstile and wait there for the last thread. The last thread to arrive *resets* the green turnstile’s counter to 0, knowing that there cannot be any threads waiting at the green turnstile. The threads can then move on to the green turnstile. Again they wait for one another, and the last thread to arrive there resets the red turnstile.

If we were to implement a barrier this way, threads would have to wait twice, which is a potential inefficiency. Fortunately, there is a trick we can use to avoid this. Instead of using a red and a green turnstile, we will use a single turnstile, but have the last thread that gets to the turnstile change its color.

[Figure 17.6](#) shows how this is done. As before a counter n counts how many threads still have to reach the barrier. The barriers also uses a variable *color* that is flipped between 0 and 1 after each use of the barrier. The last thread that reaches the barrier restores n to 0 and flips the color (using the exclusive or operation: \wedge). The other threads are waiting for the turnstile to change colors. Turning the specification of [Figure 17.6](#) into an implementation is now straightforward, as shown in [Figure 17.7](#) using condition variables.

[Figure 17.8](#) shows an implementation of a parallel sorting algorithm based on bubblesort. The threads (one for every two elements) go through three phases. In the first phase, the threads swap entries 0 and 1, 2 and 3, ... as needed. In the second phase, they swap entries 1 and 2, 3 and 4, ... as needed. Finally, they check if any elements were swapped. If so, they repeat the phases.

```

1  from synch import *
2
3  def Barrier(required) returns barrier:
4      barrier = {
5          .mutex: Lock(), .cond: Condition(),
6          .required: required, .n: 0, .color: 0
7      }
8
9  def bwait(b):
10     acquire(?b→mutex)
11     b→n += 1
12     if b→n == b→required:
13         b→color ^= 1
14         b→n = 0
15         notify_all(?b→cond)
16     else:
17         let color = b→color:
18             while b→color == color:
19                 wait(?b→cond, ?b→mutex)
20     release(?b→mutex)

```

Figure 17.7: [\[code/barrier.cv.hny\]](#) Barrier implementation

```

1  from barrier import *
2
3  const N = 5    # size of list to be sorted
4
5  thelist = [ choose { 1 .. N } for i in { 1 .. N } ]
6
7  finally all(thelist[i-1] <= thelist[i] for i in { 1 .. N - 1 })
8
9  const NTHREADS = N / 2
10 bar = Barrier(NTHREADS)
11 count = [0,] * NTHREADS    # to detect termination
12
13 def sorter(self, i):
14     var unsorted = True
15     while unsorted:
16         let old_count = count:
17             bwait(?bar)    # wait until all have assigned old_count
18
19             # Even phase
20             if thelist[i - 1] > thelist[i]:
21                 thelist[i - 1], thelist[i] = thelist[i], thelist[i - 1]
22                 count[self] += 1
23
24             bwait(?bar)    # wait until all have finished even phase
25
26             # Odd phase
27             if i < N - 1 and thelist[i] > thelist[i + 1]:
28                 thelist[i], thelist[i + 1] = thelist[i + 1], thelist[i]
29                 count[self] += 1
30
31             bwait(?bar)    # wait until all have finished odd phase
32
33             # Sorted iff nobody swapped anything
34             unsorted = count != old_count
35
36 for k in { 0 .. NTHREADS - 1}:
37     spawn sorter(k, 2*k + 1)

```

Figure 17.8: [[code/bsort.hny](#)] Parallel bubble sort

Chapter 18

Advanced Barrier Synchronization

In the previous chapter, we presented barrier synchronization for the case where the number of threads is exactly the number of threads that the barrier is waiting for. However, we may have a resource that can be used simultaneously by some maximum number of threads, but there are more threads than the maximum that would like to use the resource. Consider, for example, the following example taken from an exam:

Given is a roller coaster with a single car. Safety precautions require the following: Each ride requires that all seats are filled. That is, partially filled cars are not allowed to ride. After a ride, the car must completely empty out before new riders are allowed to enter the car. Write code to model this.

In such problems there is a shared resource (in this case the car) that is used repeatedly in what we shall call *rounds*. In each round, the resource usually has to be used to capacity. The users (the riders in this case) can be modeled by threads. It is often useful to think of these problems as having a section that threads must enter and exit, much like a critical section. In this case, entering the section means going for a ride, and exiting the section means being done with the ride.

Entering a section consists of two phases. A thread should first wait until any threads that already entered the section have exited. Then, it should wait until conditions are met, in this case until there are enough users (threads) to fill up the car. Not following this recipe might lead to the following problem: As the car is emptying out, new riders may try to enter the car because it is no longer full.

In the first phase, a thread needs to be able to detect that all threads that have entered previously have now left. In the second phase, a thread should check to see if it is the last thread to enter the second phase. If not, it should wait. If so, it should notify the ones that are waiting. When leaving the section, a thread should check if it is the last thread to leave. If so, it should notify the threads that are waiting in the first phase.

Figure 18.1 presents a solution. Method `RollerCoaster(nseats)` returns the initial state of a roller coaster with *nseats* seats to a car. Method `enter(b)` takes a pointer *b* to a roller coaster variable. Following the recipe sketched above, a thread that calls `enter(b)` blocks until

1. all previous riders have left the car, and
2. the car has filled up again.

```

1  from synch import *
2
3  def RollerCoaster(nseats): result = {
4      .mutex: Lock(), .nseats: nseats, .entered: 0, .left: nseats,
5      .empty: Condition(), .full: Condition()
6  }
7
8  def enter(b):
9      acquire(?b→mutex)
10     while b→entered == b→nseats: # wait for car to empty out
11         wait(?b→empty, ?b→mutex)
12     b→entered += 1
13     if b→entered != b→nseats: # wait for car to fill up
14         while b→entered < b→nseats:
15             wait(?b→full, ?b→mutex)
16     else: # car is ready to go
17         b→left = 0
18         notify_all(?b→full) # wake up others waiting in car
19     release(?b→mutex)
20
21 def exit(b):
22     acquire(?b→mutex)
23     b→left += 1
24     if b→left == b→nseats: # car is empty
25         b→entered = 0
26         notify_all(?b→empty) # wake up riders wanting to go
27     release(?b→mutex)

```

Figure 18.1: [[code/rollercoaster.hny](#)] Modeling a roller coaster

After “doing the ride,” the thread must call method `exit(b)`.

An important part of solving the problem is figuring out what state must be kept about the resource. The code keeps track of the number of threads that have entered the car (including the threads that have not yet entered the section and the threads that have left the section) and the number of threads that have left the section. The first number is only reset after that last thread has left the section. The second number is reset after the last thread has entered the car.

In this example exam question there was just a single instance of a resource, and all users were in some sense the same in that they are interchangeable. In the problem set below we include a problem in which there are multiple resources and a problem in which not all users are the same. Nonetheless, a similar approach can be used as illustrated above.

Exercises

18.1 Imagine a pool hall with a certain number of 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 this.

18.2 The following is a classical problem. There is a ferry that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won’t leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe. Implement a simulation of this.

Chapter 19

Example: A Concurrent File Service

This chapter presents a concurrent file service to illustrate many of the techniques we have discussed inside a single example. We will cover the specification of such a service as well as that of a disk, and show how the specification can be implemented on top of the disk. The file service implementation will use a collection of worker threads synchronizing using both ordinary locks and reader/writer locks. Clients of the file service implementation (threads themselves) use blocking synchronized queues to communicate with the workers. The example will also illustrate modular model checking, as the disk, the locks, and the queues are only specified.

In practice, there are many aspects to a file system. We will focus here on a low-level notion of a file, where the file abstraction is identified by a number (the so-called “inode number” or *ino*) and consists of a sequence of fixed-sized *blocks*. In our abstraction, each block holds an arbitrary Harmony value. If you want to remain more truthful to reality, you might only store lists of numbers of fixed length in a block, representing a block of bytes. A more complete file system would keep track of various additional information about each file, such as its size in bytes, its owner, its access rights, and when the file was last modified. Moreover, a system of folders (a.k.a. directories) built on top of the files would associate user-readable names to the files. Such abstractions can be modularly implemented on top.

Figure 19.1 shows the file system interface. Just like in Unix-like file systems, you have to specify the (maximum) number of files when you initialize the file system. `File(n)` returns a handle that must be passed to file operations, where *n* is the maximum number of files. For our example, we have only included three operations on files. `read(fs, ino, offset)` returns the block of file *ino* at the given offset, or **None** if nothing has been stored at that offset. `write(fs, ino, offset, data)` stores *data* at the given offset in file *ino*. If needed, the file is grown to include the given offset. “Holes” (unwritten blocks) are plugged with **None** values. `delete(fs, ino)` deletes a file.

Figure 19.2 shows how the file system may be tested and illustrates how the file system interface is used. As shown in Chapter 10, we can test a concurrent system by checking all interleavings of some selection of its operations. We can do this for both the specification and implementation of the file system and check that every behavior of the implementation is also a behavior of the specification.

```

1  from alloc import malloc
2
3  def File(n_files) returns fs:
4      fs = malloc([ [], ] * n_files)
5
6  def read(fs, ino, offset) returns result:
7      atomically result = (!fs)[ino][offset] if 0 <= offset < len (!fs)[ino] else None
8
9  def write(fs, ino, offset, data) returns result:
10     atomically:
11         let n = len (!fs)[ino]:
12             if 0 <= offset <= n:
13                 (!fs)[ino][offset] = data
14             else:
15                 (!fs)[ino] += ([ None, ] * (offset - n)) + [data,]
16                 result = "ok"
17
18  def delete(fs, ino) returns result:
19     atomically:
20         (!fs)[ino] = []
21         result = "ok"

```

Figure 19.1: [\[code/file.hny\]](#) Specification of the file system

```

1  from file import *
2
3  const N_FILES = 2
4  const MAX_FILE_SIZE = 2
5
6  const N_READ = 1
7  const N_WRITE = 1
8  const N_DELETE = 1
9
10 system = File(N_FILES)
11
12 def read_test(i):
13     let ino = choose { 0 .. N_FILES - 1 }
14     let offset = choose { 0 .. MAX_FILE_SIZE - 1 }:
15         print(i, "read", ino, offset)
16         let data = read(system, ino, offset):
17             print(i, "read done", ino, offset, data)
18
19 def write_test(i):
20     let ino = choose { 0 .. N_FILES - 1 }
21     let offset = choose { 0 .. MAX_FILE_SIZE - 1 }:
22         print(i, "write", ino, offset)
23         write(system, ino, offset, i)
24         print(i, "write done", ino, offset)
25
26 def delete_test(i):
27     let ino = choose { 0 .. N_FILES - 1 }:
28         print(i, "delete", ino)
29         delete(system, ino)
30         print(i, "delete done", ino)
31
32 for i in { 1 .. N_READ }:
33     spawn read_test(i)
34 for i in { 1 .. N_WRITE }:
35     spawn write_test(i)
36 for i in { 1 .. N_DELETE }:
37     spawn delete_test(i)

```

Figure 19.2: [\[code/file_btest.hny\]](#) Test program for a concurrent file system

```

1  from alloc import malloc
2
3  def new(n_blocks) returns disk:
4      disk = malloc([ None, ] * n_blocks)
5
6  def getsize(disk) returns size:
7      size = len !disk
8
9  def read(disk, bno) returns contents:
10     contents = (!disk)[bno]
11
12 def write(disk, bno, block):
13     (!disk)[bno] = block

```

Figure 19.3: [\[code/disk.hny\]](#) Specification of a disk

To store the file system, we will use a disk. Like a file, a disk is an array of blocks, albeit one of fixed length. [Figure 19.3](#) specifies a disk. The interface is similar to that of files, except that there are no inode numbers. Each block is identified by its offset or *block number*. For example, *disk_read(disk, bno)* retrieves the value of block *bno* on the given disk. Note that operations are not atomic. For example, two threads concurrently writing the same block can result in chaos. It is up to the file system implementation that this does not happen. Of course, more than one thread can read the same block at the same time. This is only a specification of a disk—an implementation may want to include a cache of blocks for good performance. Certain operations may also be re-ordered to further improve performance.

For the implementation, we will use a simplified Unix file system. In a Unix file system, the disk is subdivided into three parts ([Figure 19.4\(a\)](#)):

1. The *superblock*, at offset 0.
2. An array of fixed-sized *inodes*, stored in a range of *m* disk blocks starting at block 1. An inode number indexes into this array. Each inode maintains some information about a file, including the size and where to find the data.
3. The remaining blocks, starting at offset $1 + m$, which will either store data, metadata, or are free. Metadata blocks contain a list of block numbers.

The superblock specifies the number of inode blocks and the start of a linked list of free blocks.

In this simplified file system, each inode contains just three pieces of information about the file ([Figure 19.4\(b\)](#)): the size of the file in blocks, the block number of the first data block, and the block number of an *indirect block*—a metadata block that contains block numbers of additional data blocks. Any block number may be **None** to indicate a hole in the file (unused blocks). Note that a Unix file is essentially implemented as a tree of blocks.

Free blocks are maintained in a simple linked list. The superblock contains the block number of the first free block, which every block on the free list contains the block number of the next free

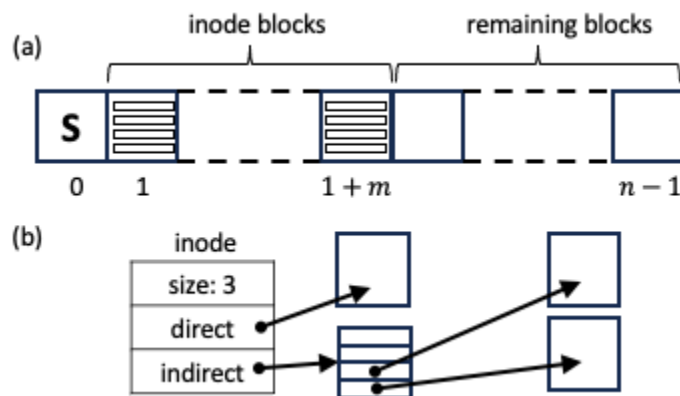


Figure 19.4: The file system data structure: (a) disk layout (1 superblock, n blocks, m inode blocks, 4 inodes per block); (b) inode for a file with 3 data blocks

block, or *None* for the end of the list. (In a more realistic Unix file system, each block on the free list would maintain pointers to additional free blocks.)

Note that the entire file system data structure is essentially a tree of blocks, with the superblock acting as the root of the tree. The superblock points to the free list and the inode blocks. The inode blocks point to the blocks that are allocated. An invariant of the data structure is that all blocks are in the tree and each block (except for the superblock) is pointed to exactly once. The invariant may not hold while the data structure is being updated. For example, temporarily a block may be both on the free list but also be part of an inode, or a block may not be referenced at all.

Figure 19.5 shows the modules that the file system implementation will use and some constants. The implementation uses the actor model (Chapter 16)—the `synch` module provides blocking multi-reader/multi-writer queues that the actors will use for messaging. The file server itself is implemented as a multithreaded actor. The threads synchronize using a plain lock for the free list and reader/writer locks for each of the inode blocks. `N_BLOCKS` specifies the size of the disk to be used in blocks. `INODES_PER_BLOCK` specifies how many inodes fit in an inode block. `INDIR_PER_BLOCK` specifies how many block numbers fit in a metadata block. Note that the maximum file size in this simplified file system is $1 + \text{INDIR_PER_BLOCK}$ blocks. In a more realistic Unix file system, indirect blocks can point to other indirect blocks, allowing for much larger files. `N_WORKERS` specifies the number of worker threads or actors.

Figure 19.6 shows the implementation of the file system interface, which is the same interface as the specification (Figure 19.1) but a different implementation. `File()`, instead of returning an object containing an array of files, returns an object containing a queue to communicate with the file system worker threads. The first argument is the maximum number of files. The number of inode blocks can be computed from the number of files by dividing by `INODES_PER_BLOCK` and rounding up. The function also initializes a disk object using `fs_init` and then allocates some shared state to be used by the worker threads. The shared state includes the following information:

- `.disk`: points to the disk object;
- `.req_q`: the shared queue on which requests from clients arrive;

```

12  from synch import *           # shared queue for file server and lock for superblock
13  from rlock import *          # read/write locks for inode blocks
14  from alloc import *          # malloc/free
15  from lists import subseq      # list slicing
16  import disk                   # reading and writing blocks
17
18  const N_BLOCKS = 10           # total number of disk blocks
19  const INODES_PER_BLOCK = 2    # number of inodes that fit in a block
20  const INDIR_PER_BLOCK = 4     # number of block pointers per block

```

Figure 19.5: [\[code/file_inode.hny\]](#) File system implementation preamble

- *.free_lock*: a lock on the free list;
- *.ib_locks*: an array of reader/writer locks, one for each inode block.

Finally, `File()` spawns the `fs_worker()` threads that will handle requests.

The remaining interfaces simply put a request on the request queue and wait for a response on another queue *res_q* that is allocated just for this purpose. Note that the request queue has concurrent producers (the clients) and concurrent consumers (the worker threads). The response queues are single use only and have a single producer (a worker thread) and a single consumer (the client).

Each worker thread (Figure 19.7) executes an infinite loop, obtaining client requests and handling them. Each request is for a particular inode. The worker first determines which inode block needs to be locked. Depending on the request, it obtains either a read lock or a write lock on the block. In practice, files are read much more frequently than written, so reader/writer locks can significantly improve the potential for concurrent access compared to regular locks. The requests themselves are handled in the *fs_query_request()* and *fs_update_request()* methods, which we will describe below.

Figure 19.8 shows how the disk is initialized with a fresh file system. The superblock is first initialized with the number of inode blocks and a linked list of free blocks. Next, the inode blocks are initialized, each with an empty file.

Figure 19.9 contains the code for allocating and freeing blocks. The methods first acquire the free list lock and then do simple linked list operations. Block allocation can be made much more efficient if each worker thread maintained a small cache of free blocks that it can allocate from without having to coordinate with the other workers.

Figure 19.10 shows the code for read-only operations on files, which is currently only reading a block from a file. The method first needs to read the block that contains the inode. Argument *ib* contains the inode block number, which is computed by dividing the inode number by `INODES_PER_BLOCK` and adding 1 (because the first inode block is block 1). To get the index of the inode in the block, you need to compute the remainder of that division. Handling of a **read** request depends on the offset. If the offset is 0, then the request tries to access the data that is in the direct block. Otherwise, it is necessary to read the indirect block first. In any block number is **None** along the way, the response should be **None**.

```

168 def File(n_files) returns req_q:
169     req_q = malloc(Queue())
170     let n_inode_blocks = (n_files + (INODES_PER_BLOCK - 1)) / INODES_PER_BLOCK
171     let n_workers = 2
172     let d = disk.new(N_BLOCKS):
173         # Initialize the file system on disk
174         fs_init(d, n_inode_blocks)
175
176         # Allocate the in-memory shared state of the file server
177         let fs_state = malloc({ .next: None,
178             .disk: d, .req_q: req_q, .free_lock: Lock(),
179             .n_inode_blocks: n_inode_blocks,
180             .ib_locks: [ RWlock(), ] * n_inode_blocks }):
181
182             # Start worker threads to handle client requests
183             for i in { 1 .. n_workers }:
184                 spawn eternal fs_worker(fs_state, i)
185
186 def read(req_q, ino, offset) returns result:
187     let res_q = malloc(Queue()):
188         put(req_q, { .type: "read", .ino: ino, .offset: offset, .q: res_q })
189         result = get(res_q)
190         free(res_q)
191
192 def write(req_q, ino, offset, data) returns result:
193     let res_q = malloc(Queue()):
194         put(req_q, { .type: "write", .ino: ino, .offset: offset, .data: data, .q: res_q })
195         result = get(res_q)
196         free(res_q)
197
198 def delete(req_q, ino) returns result:
199     let res_q = malloc(Queue()):
200         put(req_q, { .type: "delete", .ino: ino, .q: res_q })
201         result = get(res_q)
202         free(res_q)

```

Figure 19.6: [\[code/file_inode.hny\]](#) File system interface implementation

```

148 # A worker thread handles client requests
149 def fs_worker(fs_state_init, id):
150     var fs_state = fs_state_init
151     while True:
152         let req = get(fs_state→req_q)
153         let ib = req.ino / INODES_PER_BLOCK:
154             if req.type in { "write", "delete" }:
155                 write_acquire(?fs_state→ib_locks[ib])
156                 fs_update_request(fs_state, id, req, ib)
157                 write_release(?fs_state→ib_locks[ib])
158                 put(req.q, "ok")
159             else:
160                 assert req.type == "read"
161                 read_acquire(?fs_state→ib_locks[ib])
162                 let response = fs_read_request(fs_state, req, ib):
163                     read_release(?fs_state→ib_locks[ib])
164                 put(req.q, response)

```

Figure 19.7: [\[code/file.inode.hny\]](#) File server and worker threads

```

24 # Initialize the file system by writing the superblock, the free list, and
25 # the i-node blocks,
26 def fs_init(d, n_inode_blocks):
27     # Initialize the i-node blocks
28     for i in { 1 .. n_inode_blocks }:
29         disk.write(d, i, [ { .direct: None, .indir: None, .size: 0 }, ] * INODES_PER_BLOCK)
30
31     # Free the data blocks
32     var free_list = None
33     for i in { n_inode_blocks + 1 .. N_BLOCKS - 1 }:
34         disk.write(d, i, free_list)
35         free_list = i
36
37     # Write the superblock
38     disk.write(d, 0, { .n_inode_blocks: n_inode_blocks, .free_list: free_list })

```

Figure 19.8: [\[code/file.inode.hny\]](#) File system initialization


```

42  # Write to the disk
43  def do_write(fs_state, bno, block):
44      disk.write(fs_state→disk, bno, block)
45
46  # Allocate a disk block.
47  def fs_alloc(fs_state) returns bno:
48      acquire(?fs_state→free_lock)
49      var super = disk.read(fs_state→disk, 0)
50      bno = super.free_list
51      if bno != None:
52          super.free_list = disk.read(fs_state→disk, bno)
53          do_write(fs_state, 0, super)
54      release(?fs_state→free_lock)
55
56  # Free block bno.
57  def fs_free(fs_state, bno):
58      acquire(?fs_state→free_lock)
59      var super = disk.read(fs_state→disk, 0)
60      do_write(fs_state, bno, super.free_list)
61      super.free_list = bno
62      do_write(fs_state, 0, super)
63      release(?fs_state→free_lock)
64
65  # Read inode ino in block ib.
66  def fs_get_inode(fs_state, ib, ino) returns inode:
67      let inode_block = disk.read(fs_state→disk, 1 + ib):
68          inode = inode_block[ino % INODES_PER_BLOCK]
69
70  # Write inode ino in block ib.
71  def fs_put_inode(fs_state, ib, ino, inode):
72      var inode_block = disk.read(fs_state→disk, 1 + ib)
73      inode_block[ino % INODES_PER_BLOCK] = inode
74      do_write(fs_state, 1 + ib, inode_block)

```

Figure 19.9: [\[code/file_inode.hny\]](#) File system utility functions

```

78  # Handle a read request. A read lock on i-node block ib has been acquired.
79  def fs_read_request(fs_state, req, ib) returns result:
80      # Read the inode block and extract the inode
81      let inode = fs_get_inode(fs_state, ib, req.ino):
82          # Read the direct block. Return None if there is no direct block.
83          if req.offset == 0:
84              if inode.direct == None:
85                  result = None
86              else:
87                  result = disk.read(fs_state→disk, inode.direct)
88
89          # Read indirectly. If there is no indirect block return None
90          elif inode.indir == None:
91              result = None
92
93          # Read the indirect block and get the pointer to the data block,
94          # which may be None.
95          else:
96              let indir = disk.read(fs_state→disk, inode.indir):
97                  if indir[req.offset - 1] == None:
98                      result = None
99                  else:
100                     result = disk.read(fs_state→disk, indir[req.offset - 1])

```

Figure 19.10: [[code/file_inode.hny](#)] Handling of file read requests

```

104  # Handle an update request. A write lock on i-node block ib has been acquired.
105  def fs_update_request(fs_state, id, req, ib):
106      var inode = fs_get_inode(fs_state, ib, req.ino)
107      if req.type == "write":
108          if req.offset == 0:
109              if inode.direct == None:
110                  inode.direct = fs_alloc(fs_state)
111                  inode.size = max(inode.size, 1)
112                  fs_put_inode(fs_state, ib, req.ino, inode)
113                  do_write(fs_state, inode.direct, req.data)
114              else:
115                  if inode.indir == None:
116                      inode.indir = fs_alloc(fs_state)
117                      inode.size = max(inode.size, req.offset + 1)
118                      fs_put_inode(fs_state, ib, req.ino, inode)
119                      let bno = fs_alloc(fs_state)
120                      let indir = [ bno if i == (req.offset - 1) else None
121                                  for i in { 0 .. INDIR_PER_BLOCK - 1 } ]:
122                          do_write(fs_state, bno, req.data)
123                          do_write(fs_state, inode.indir, indir)
124                  else:
125                      var indir = disk.read(fs_state→disk, inode.indir)
126                      if indir[req.offset - 1] == None:
127                          indir[req.offset - 1] = fs_alloc(fs_state)
128                          do_write(fs_state, inode.indir, indir)
129                          do_write(fs_state, indir[req.offset - 1], req.data)
130                      if inode.size <= req.offset:
131                          inode.size = req.offset + 1
132                          fs_put_inode(fs_state, ib, req.ino, inode)
133              else:
134                  assert req.type == "delete"
135                  if inode.direct != None:
136                      fs_free(fs_state, inode.direct)
137                  if inode.indir != None:
138                      let indir = disk.read(fs_state→disk, inode.indir):
139                          for bno in indir:
140                              if bno != None:
141                                  fs_free(fs_state, bno)
142                  inode.direct = inode.indir = None
143                  inode.size = 0
144                  fs_put_inode(fs_state, ib, req.ino, inode)

```

Figure 19.11: [\[code/file_inode.hny\]](#) Handling of file write and delete requests

Finally, [Figure 19.11](#) contains the code to write to a file or to delete a file. The write operation first checks if the direct block is updated or a block accessible through an indirect block. If it is the direct block, the code checks to see if the block has already been allocated. Otherwise it needs to check if the indirect block has already been allocated as well as the data block. Data blocks, indirect blocks, and even the inode block may all have to be updated as part of this operation. Deleting a file puts all its blocks back on the free list and clears the inode.

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. After this, *count* must equal 1.

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 7.3](#). [Figure 20.3](#) shows how one might go about this. But locks are intended to solve synchronization issues between multiple threads. But an interrupt handler is not run by another thread—it is run by the same thread that experienced the interrupt. 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.

```

1  count = 0
2  done = False
3
4  finally count == 1
5
6  def handler():
7      count += 1
8      done = True
9
10 def main():
11     trap handler()
12     await done
13
14 spawn main()

```

Figure 20.1: [\[code/trap.hny\]](#) How to use **trap**

```

1  count = 0
2  done = False
3
4  finally count == 2
5
6  def handler():
7      count += 1
8      done = True
9
10 def main():
11     trap handler()
12     count += 1
13     await done
14
15 spawn main()

```

Figure 20.2: [\[code/trap2.hny\]](#) A race condition with interrupts

```

1  from synch import Lock, acquire, release
2
3  countlock = Lock()
4  count = 0
5  done = False
6
7  finally count == 2
8
9  def handler():
10     acquire(?countlock)
11     count += 1
12     release(?countlock)
13     done = True
14
15  def main():
16     trap handler()
17     acquire(?countlock)
18     count += 1
19     release(?countlock)
20     await done
21
22  spawn main()

```

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

```

1  count = 0
2  done = False
3
4  finally count == 2
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
17 spawn main()

```

Figure 20.4: [\[code/trap4.hny\]](#) Disabling and enabling interrupts

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 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 an 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 **acquire()** to the beginning of the **increment** method and **release()** to the end of **increment** and see what happens. This incorrect code can lead to 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


```
1  count = 0
2  done = False
3
4  finally count == 2
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
20 spawn main()
```

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

```

1  from synch import Lock, acquire, release
2
3  count = 0
4  countlock = Lock()
5  done = [ False, False ]
6
7  finally count == 4
8
9  def increment():
10     let prior = setintlevel(True):
11         acquire(?countlock)
12         count += 1
13         release(?countlock)
14         setintlevel(prior)
15
16  def handler(self):
17     increment()
18     done[self] = True
19
20  def thread(self):
21     trap handler(self)
22     increment()
23     await done[self]
24
25  spawn thread(0)
26  spawn thread(1)

```

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

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.

Exercises

20.1 The `put` method you implemented in [Exercise 13.1](#) 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) returns prior:
8      atomically:
9          prior = !pcnt
10         !pcnt += 1
11
12  def exch(pv) returns prior:
13      atomically:
14          prior = !pv
15          !pv = None
16
17  def produce(item):
18      items[inc(?back)] = item
19
20  def consume() returns next:
21      next = None
22      while next == None:
23          var i = 0
24          while (i < back) and (next == None):
25              next = 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 (a.k.a. *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 15.1](#), with one seqlock for the entire bank (i.e., no locks on individual accounts). Method `transfer` is an update operation; method `total` is a read-only operation. Explain how a seqlock can lead to starvation.

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, \dots, m_n in that order, then if Bob receives an application message m , then $m = m_i$ for some i and, unless m is the very first message, Bob will already have received application messages m_1, \dots, m_{i-1} (safety). Also, if the network is fair and Alice sends application message m , then eventually Bob should deliver m (liveness).

The *Alternating Bit Protocol* (Figure 22.1) 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 `()` to represent nothing 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)` models sending a message m to $pchan$, where

```

1  def net_send(pchan, msg):
2      atomically:
3          !pchan = msg if choose({ False, True }) else ()
4
5  def net_recv(pchan) returns msg:
6      atomically:
7          msg = !pchan
8          !pchan = ()
9
10 def app_send(net, seq, payload):
11     !seq = 1 - !seq
12     let m = { .seq: !seq, .payload: payload }:
13         var blocked = True
14         while blocked:
15             net_send(?net→s_chan, m)
16             let response = net_recv(?net→r_chan):
17                 blocked = (response == ()) or (response.ack != !seq)
18
19 def app_recv(net, seq) returns payload:
20     !seq = 1 - !seq
21     var blocked = True
22     while blocked:
23         let m = net_recv(?net→s_chan):
24             if m != ():
25                 net_send(?net→r_chan, { .ack: m.seq })
26                 if m.seq == !seq:
27                     payload = m.payload
28                     blocked = False

```

Figure 22.1: [\[code/abp.hny\]](#) Alternating Bit Protocol


```

1  import abp
2
3  const NMSGs = 10
4
5  invariant s_seq in { 0, 1 }
6  invariant r_seq in { 0, 1 }
7
8  network = { .s_chan: (), .r_chan: () }
9  s_seq = r_seq = 0
10
11 def sender():
12     for i in {1..NMSGs}:
13         abp.app_send(?network, ?s_seq, i)
14
15 def receiver():
16     var i = 1
17     while True:
18         let payload = abp.app_recv(?network, ?r_seq):
19             assert payload == i
20             i += 1
21
22 spawn sender()
23 spawn eternal receiver()

```

Figure 22.2: [\[code/abptest.hny\]](#) Test code for alternating bit protocol

pchan is either *?s_chan* or *?r_chan*. The method places either *m* (to model a successful send) or *()* (to model loss) in *!pchan.net_recv*{*pchan*} models checking *!pchan* for the next message. If there is a message, it sets *!pchan* to *()*.

Method **app_send**(*net, seq, msg*) retransmits { *.seq: !seq, .payload: msg* } until an acknowledgment for *!seq* is received. Method **app_recv**(*net, seq*) returns the next successfully received message (with the given sequence number bit) if any. Figure 22.2 shows how the methods may be used to send and receive a stream of **NMSGs** messages reliably. The number of messages must be bounded, because model checking requires a finite model. Note that the **receiver()** is spawned as **eternal** because it does not terminate.

Exercises

22.1 Chapter 16 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.

22.3 The alternating bit protocol is a special case of a *sliding window* protocol with a window size of 1 and using 2 sequence numbers. TCP uses a sliding window protocol. Using a window size of *W*, a client is able to send *W* messages at the same time before having to wait for an acknowledgment, but you'll need more than *W* sequence numbers to make it work. After receiving an acknowledgment, the window can be moved up. Implement a sliding window protocol.

Chapter 23

Leader Election

Leader election is the problem of electing a unique leader in a network of processors. Typically this is challenging because the processors have only limited information. In the version that we present, each processor has a unique identifier. The processors are organized in a ring, but each processor only knows its own identifier and the identifier of its successor on the ring. Having already looked into the problem of how to make the network reliable, we assume here that each processor can reliably send messages to its successor.

The protocol that we present elects as leader the processor with the highest identifier [CR79] and works in two phases: In the first phase, each processor sends its identifier to its successor. When a processor receives an identifier that is larger than its own identifier, it forwards that identifier to its successor as well. If a processor receives its own identifier, it discovers that it is the leader. That processor then starts the second phase by sending a message around the ring notifying the other processors of the leader's identifier.

Figure 23.1 describes the protocol and its test cases in Harmony. In Harmony, processors can be modeled by threads and there are a variety of ways in which one can model a network using shared variables. Here, we model the network as a set of messages. The `send` method atomically adds a message to this set. Messages are tuples with three fields: $(dst, id, found)$. dst is the identifier of the destination processor; id is the identifier that is being forwarded; and $found$ is a boolean indicating the second phase of the protocol. The `receive(self)` method looks for all messages destined for the processor with identifier $self$.

To test the protocol, the code first chooses the number of processors and generates an identifier for each processor, chosen non-deterministically from a set of `NIDS` identifiers. The code also keeps track in the variable `leader` of what the highest identifier is, so it can later be checked.

Method `processor(self, succ)` is the code for a processor with identifier $self$ and successor $succ$. It starts simply by sending its own identifier to its successor. The processor then loops until it discovers the identifier of the leader in the second phase of the protocol. A processor waits for a message using the Harmony **atomically when exists** statement. This statement takes the form

atomically when exists v in s : *statement block*

where s is a set and v is variable that is bound to an element of s . The properties of the statement are as follows:

```

1  const NIDS = 5      # number of identifiers
2
3  network = {}        # the network is a set of messages
4  leader = 0          # used for checking correctness
5
6  def send(msg):
7      atomically network |= { msg }
8
9  def receive(self) returns msg:
10     msg = { (id, found) for (dst, id, found) in network where dst == self }
11
12 def processor(self, succ):
13     send(succ, self, False)
14     var working = True
15     while working:
16         atomically when exists (id, found) in receive(self):
17             if id == self:
18                 assert self == leader
19                 send(succ, id, True)
20             elif id > self:
21                 assert self != leader
22                 send(succ, id, found)
23             if found:
24                 working = False
25
26 var ids, nprocs, procs = { 1 .. NIDS }, choose({ 1 .. NIDS }, [])
27 for i in { 0 .. nprocs - 1 }:
28     let next = choose(ids):
29         ids -= { next }
30         procs += [ next, ]
31         if next > leader:
32             leader = next
33 for i in { 0 .. nprocs - 1 }:
34     spawn processor(procs[i], procs[(i + 1) % nprocs])

```

Figure 23.1: [\[code/leader.hny\]](#) A leader election protocol on a ring

- it waits until s is non-empty;
- it is executed atomically;
- v is selected non-deterministically, like in the **choose** operator.

If a processor receives its own identifier, it knows its the leader. The Harmony code checks this using an assertion. In real code the processor could not do this as it does not know the identifier of the leader, but assertions are only there to check correctness. The processor then sends a message to its successor that the leader has been found. If the processor receives an identifier higher than its own, the processor knows that it cannot be the leader. In that case, it simply forwards the message. A processor stops when it receives a message that indicates that the leader has been identified.

Note that there is a lot of non-determinism in the specification, leading to a lot of executions that must be checked. First, every possible permutation of identifiers for the processors is tried. When there are multiple messages to receive by a processor, every possible order is tried (including receiving the same message multiple times). Fortunately, the **atomically when exists** statement is executed atomically, otherwise the body of the statement could lead to additional thread interleavings. Because in practice the different processors do not share memory, it is not necessary to check those interleavings.

Exercises

23.1 Check if the code finds a unique leader if identifiers are not unique.

23.2 Messages are added atomically to the network. Is this necessary? What happens if you remove the **atomically** keyword? Explain what happens.

Chapter 24

Transactions and Two Phase Commit

Modern databases support multiple clients concurrently accessing the data. They store data on disk, but we will ignore that in this book. (If you want to model a disk, this is probably best done as a separate thread that maintains the contents of the disk.) The complication we address here is that databases may be *sharded*, where different parts of the data are stored on different servers. The different servers may even be under different authoritative domains, such as multiple banks maintaining the accounts of their clients.

In database terminology, a *transaction* is an operation on a database. The operation can be quite complex, and the execution of a transaction should have the following two properties:

- *all-or-nothing*: a transaction should either complete, or it should be a no-op. A transaction should never partially execute and then give up because of some kind of error or something. Database people call this *atomicity*, but it is not the same kind of atomicity that we have been discussing in this book.
- *serialized*: any two concurrent transactions should appear to execute in some order. Database people call this *isolation*: one transaction should not be able to witness the intermediate state of another transaction in execution.

We will use as an example a distributed database that maintains bank accounts. For simplicity, we will model this as a collection of banks, each maintaining a single account. There are two kinds of transactions: *transfer* (similar to [Exercise 15.1](#)) and *check*. In this example, a transfer is a transaction that moves some funds between two accounts. A check is a transaction over all accounts and checks that the sum of the balances across the accounts remains the same.

Executing such transactions must be done with care. Consider what would happen if transactions are not all-or-nothing or are not serialized. A transfer consists of two operations: Withdrawing funds from one account and depositing the same amount of funds in the other. These two operations can be done concurrently, but if the withdrawal fails (for example, because there are not sufficient funds in the source account) then the whole transaction should fail and become a no-op. Even if this is not the case, a concurrent check transaction may accidentally witness a state in which either the withdrawal or the deposit happened, but not both. And matters get more complicated with multiple concurrent transfers.

```

2  const NBANKS = 3
3  const NCOORDS = 2
4  const MAX_BALANCE = 1
5
6  network = {}
7
8  def send(msg):
9      atomically network |= { msg }
10
11 def bank(self, _balance):
12     var balance = _balance
13     var status, received = (), {}
14     while True:
15         atomically when exists req in network – received when req.dst == self:
16             received |= { req }
17             if req.request == "withdraw":
18                 if (status != ()) or (req.amount > balance):
19                     send({ .dst: req.src, .src: self, .response: "no" })
20                 else:
21                     status = balance
22                     balance -= req.amount
23                     send({ .dst: req.src, .src: self, .response: "yes", .funds: balance })
24             elif req.request == "deposit":
25                 if status != ():
26                     send({ .dst: req.src, .src: self, .response: "no" })
27                 else:
28                     status = balance
29                     balance += req.amount
30                     send({ .dst: req.src, .src: self, .response: "yes", .funds: balance })
31             elif req.request == "commit":
32                 assert status != ()
33                 status = ()
34             else:
35                 assert (status != ()) and (req.request == "abort")
36                 balance, status = status, ()

```

Figure 24.1: [[code/2pc.hny](#)] Two Phase Commit protocol: code for banks

```

40 def transfer(self, b1, b2, amt):
41     send({ .dst: b1, .src: self, .request: "withdraw", .amount: amt })
42     send({ .dst: b2, .src: self, .request: "deposit", .amount: amt })
43     atomically let msgs = { m for m in network where m.dst == self }
44     when { m.src for m in msgs } == { b1, b2 }:
45         if all(m.response == "yes" for m in msgs):
46             for m in msgs where m.response == "yes":
47                 send({ .dst: m.src, .src: self, .request: "commit" })
48         else:
49             for m in msgs where m.response == "yes":
50                 send({ .dst: m.src, .src: self, .request: "abort" })
51
52 def check(self, total):
53     let allbanks = { (.bank, i) for i in { 0 .. NBANKS - 1 } }:
54     for b in allbanks:
55         send({ .dst: b, .src: self, .request: "withdraw", .amount: 0 })
56     atomically let msgs = { m for m in network where m.dst == self }
57     when { m.src for m in msgs } == allbanks:
58         assert all(m.response == "yes" for m in msgs) =>
59             (sum(m.funds for m in msgs) == total)
60         for m in msgs where m.response == "yes":
61             send({ .dst: m.src, .src: self, .request: "abort" })
62
63 let balances = [ choose({ 0 .. MAX_BALANCE }) for _ in { 0 .. NBANKS - 1 } ]:
64     for i in { 0 .. NBANKS - 1 }:
65         spawn eternal bank((.bank, i), balances[i])
66     for i in { 1 .. NCOORDS }:
67         if choose({ "transfer", "check" }) == .transfer:
68             let b1 = choose({ (.bank, j) for j in { 0 .. NBANKS - 1 } })
69             let b2 = choose({ (.bank, j) for j in { 0 .. NBANKS - 1 } } - { b1 }):
70                 spawn transfer((.coord, i), b1, b2, 1)
71         else:
72             spawn check((.coord, i), sum(balances))

```

Figure 24.2: [\[code/2pc.hny\]](#) Two Phase Commit protocol: code for transaction coordinators

The Two-Phase Commit protocol [Gra78] is a protocol that can be used to implement transactions across multiple database servers—banks in this case. Each transaction has a *coordinator* that sends a **PREPARE** message to each of the servers involved in the transaction, asking them to prepare to commit to their part in a particular transaction. A server can either respond with **YES** if it is ready to commit and will avoid doing anything that might jeopardize this (like committing a conflicting transaction), or with **NO** if it does not want to participate in the transaction. If all servers respond with **YES**, then the coordinator can decide to *commit* the transaction. Otherwise the coordinator must decide to *abort* the transaction. In the second phase, the servers that responded with **YES** (if any) must be notified to inform them of the coordinator’s decision.

Different transactions can have different coordinators. In our implementation, each bank and each coordinator is a thread. Figure 24.1 shows the code for a bank. The state of a bank consists of the following local variables:

- *self*: the bank’s identifier;
- *balance*: the current balance in the account;
- *status*: either contains () if the bank is not involved in an ongoing transaction or contains the balance just before the transaction started;
- *received*: the set of messages received and handled so far.

Messages sent to a bank are dictionaries with the following fields:

- *.dst*: identifier of the bank;
- *.src*: identifier of the coordinator that sent the message;
- *.request*: request type, which is either *.withdraw*, *.deposit*, *.commit*, or *.abort*;
- *.amount*: amount to withdraw or deposit.

A bank waits for a message destined to itself that it has not yet received. In case of a withdrawal when the bank is idle and there are sufficient funds, the bank saves the current balance in *status* to indicate an ongoing transaction and what its original balance was. The bank then responds with a *.yes* message to the coordinator, including the new balance. Otherwise, the bank responds with a *.no* message. Deposits are similar, except that it is not necessary to check for sufficient funds. In case of a *.commit* message, the bank changes its status to (), indicating that there is no ongoing transaction. In case of a *.abort* message, the bank restores *balance* first.

Figure 24.2 contains the code for transfers and inquiries, as well as tests. The `receive()` method is used by coordinators in an **atomically when exists** statement to wait for a response from each bank involved in a transaction. Argument `self` is the identifier of the coordinator and `sources` is the set of banks. The method returns the empty set if there are not yet responses from all banks. Otherwise it returns a singleton set containing the set of responses, one for each source.

The `transfer()` method contains the code for the transfer transaction. Argument *self* is the identifier of the coordinator, *b1* is the source bank, *b2* is the destination bank, and *amt* is the amount to transfer. The coordinator sends a **PREPARE** message containing a *.withdraw* request to *b1* and a **PREPARE** message containing a *.deposit* request to *b2*. The coordinator then waits for responses from each. If both responses are *.yes*, then it commits the transaction, otherwise it aborts the transaction.

The `check()` method checks if the sum of the balances equals *total*, the sum of the initial balances. The code is similar to `transfer`, except that it always aborts the transaction—there is no need to ever commit it. As a code-saving hack: The balance inquiry is done by withdrawing \$0.

As for testing, the initial balances are picked arbitrarily between 0 and `MAX_BALANCE` (and Harmony as always will try every possible set of choices). Each coordinator chooses whether to do a transfer or a check. In case of a transfer, it also chooses the source bank and the destination bank.

While the protocol perhaps seems simple enough, there are a lot of `if` statements in the code, making it hard to reason about correctness. Model checking is useful to see if there are corner cases where the code does not work. While confidence increases by increasing the number of banks or the number of coordinators, doing so quickly increases the number of possible states so that model checking may become infeasible.

Exercises

24.1 In [Exercise 15.1](#) the code ran into a deadlock. Can the code in this chapter run into a deadlock? Explain.

24.2 Transactions can fail for two reasons: A transfer transaction can fail because of insufficient funds, but in general transaction can fail if there is a conflict with another transaction. The latter can be fixed by retrying the transaction until it commits. Implement this.

24.3 One way to reduce the number of conflicts between transactions is to distinguish read and write operations. Two read operations (in our case, operations that withdraw \$0 do not conflict, so a bank could have multiple ongoing read operations for different transactions. Implement this.

24.4 Two-phase commit can tolerate servers failing. If a server does not respond within some reasonable amount of time, the coordinator can abort the transaction. Implement this.

Chapter 25

Chain Replication

As you have probably experienced, computers can crash. If you are running a web service, you may not be able to afford a long outage. If you are running software that flies a plane, then an outage for any length of time could lead to a disaster. To deal with service outages caused by computers crashing, you may want to *replicate* the service onto multiple computers. As long as one of the computers survives, the service remains available.

Besides availability, it is usually important that the replicated service acts as if it were a single one. This requires that the replicas of the service coordinate their actions. The *Replicated State Machine Approach* [Lam78, Sch90] is a general approach to do just this. First, you model your service as a deterministic state machine. The replicas each run a copy of the state machine, started in the same state. As long as the replicas handle the same inputs in the same order, determinism guarantees that they produce the same outputs in the same order.

Figure 25.1 presents a Harmony specification of state machine replication. We model the state machine as a *history*: a sequence of operations. In a replicated state machine, the abstract network maintains this history as an ordered queue of messages. NOPS clients each place an operation on the network. The replicas process messages from the ordered network.

All but one of the replicas is allowed to crash. Crashes are modeled as interrupts, so we use Harmony’s **trap** clause to schedule one. When crashing, a replica simply stops. The model chooses one of the replicas that is not allowed to crash. Of course, a replica does not know whether it is immortal or not in practice—it should just assume that it is. The immortality of one of the replicas is only used for modeling the assumptions we make about the system.

The behavior is captured as before. Before an operation is added to the network, a client prints the operation (in this case, its own identifier). After a replica processes an operation, it prints a pair consisting of its own identifier and the operation. All replicas print the same operations in the same order until they crash. Figure 25.2 shows the allowed behaviors in case there are just two clients and two replicas. Because one of the replicas is immortal and clients do not crash, at least one of the replicas will print both operations (liveness). If both do, they do so in the same order (safety).

But in reality the network is not an ordered queue and better modeled as a set of messages. The trick now is to ensure that all replicas handle the same requests in the same order and to do so in a way that continues to work even if some strict subset of replicas crash. *Chain Replication* [vRS04] is such a replication protocol. In Chain Replication, the replicas are organized in a linear chain

```

1  const NREPLICAS = 3    # number of replicas
2  const NOPS = 2         # number of operations
3
4  network = []           # the network is a queue of messages
5
6  def crash():
7      stop()
8
9  def send(msg):
10     atomically network += [msg,]
11
12  def replica(self, immortal):
13     if not immortal:
14         trap crash()
15     var delivered = 0
16     while True:
17         atomically when len(network) > delivered:
18             let msg = network[delivered]:
19                 print(self, msg)
20                 delivered += 1
21
22  def client(self):
23     print(self)
24     send(self)
25
26  let immortal = choose {1..NREPLICAS}:
27     for i in {1..NREPLICAS}:
28         spawn eternal replica(i, i == immortal)
29  for i in {1..NOPS}:
30     spawn client(i)

```

Figure 25.1: [\[code/rsm.hny\]](https://code/rsm.hny) Replicated State Machine

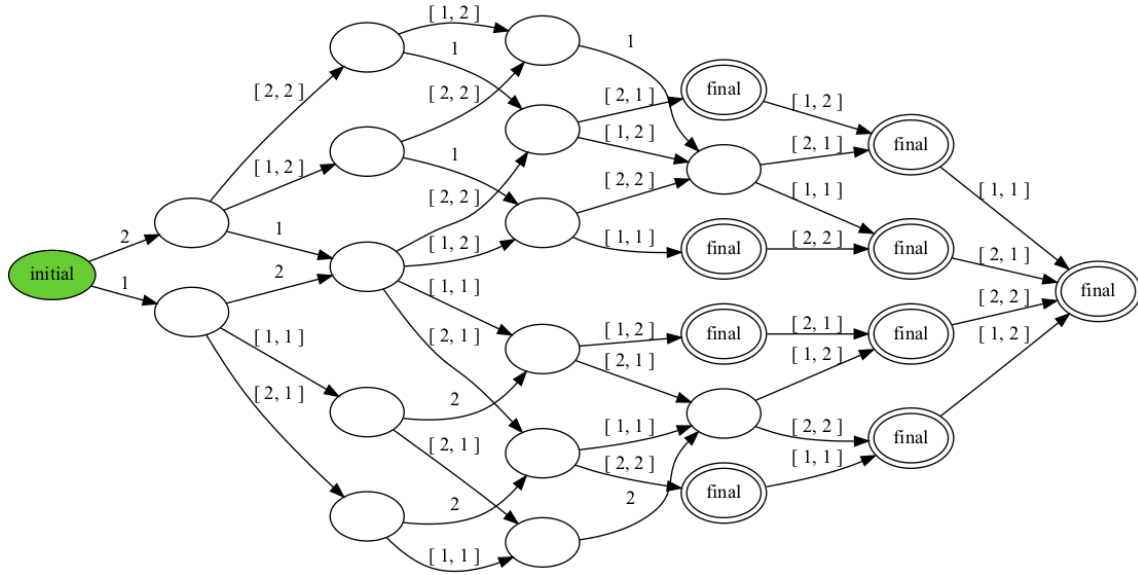


Figure 25.2: The DFA generated by Figure 25.1 when NOPS=2 and NREPLICAS=2

which may change as replicas crash. Importantly, at any point in time there is only one *head* and one *tail* replica.

Only the head is allowed to accept new operations from clients. When it does so, it adds the operation to the end of its history and sends the history to its successor on the chain. When the direct successor receives such a history, it makes sure that the history is an extension of its own and, if so, replaces its own history with the one received. It then sends the history on to its successor, if any. When an operation reaches the tail, the operation is what is known as *stable*—it has been reliably ordered and persisted.

In order for this to work, each replica needs to know who is its predecessor and who is its successor. So, when a replica fails, its neighbors should find out about it. In practice, one server can detect the failure of another server by pinging it. If a server does not receive a response to its ping within some maximum amount of time, then the server considers its peer crashed. Note that this, in general, is not a safe thing to do—the network or the peer may be temporarily slow but the peer is not necessarily crashed when the timer expires.

Nonetheless, we will assume here that failure detection does not make mistakes and that eventually every failure is detected. This is called the *Fail-Stop* failure model [SS83], which is distinct from the often more realistic *Crash Failure* model where processes can crash but accurate detection is not available. We will consider that more realistic failure model in the upcoming chapters. For chain replication, when a replica crashes, it will reliably notify the other replicas by broadcasting a message over the reliable network. Because failure detection is accurate, at most one replica can think it is the head at any time (and, if so, it is in fact the head). Moreover, when it has detected all its predecessors having failed, eventually some replica thinks it is the head. The same is true for the tail.

```

3  const NREPLICAS = 3    # number of replicas
4  const NOPS = 2         # number of operations (or clients)
5
6  network = {}          # the network is a set of messages
7
8  def send(self, dst, msg):    # send msg to replica dst
9      atomically network |= { (dst, (self, msg)) }
10
11 def broadcast(self, msg):    # broadcast msg to all
12     atomically for dst in {1..NREPLICAS}:
13         network |= { (dst, (self, msg)) }
14
15 def receive(self) returns msgs:    # return messages for me
16     msgs = { payload for (dst, payload) in network where (dst == self) }
17
18 def crash(self):            # server 'self' is crashing
19     broadcast(self, "crash")    # notify all other replicas
20     stop()
21
22 def is_prefix(hist1, hist2) returns result: # hist1 is a strict prefix of hist2
23     result = (len(hist1) < len(hist2)) and
24         all(hist1[i] == hist2[i] for i in {0..len(hist1)-1})

```

Figure 25.3: [[code/chain.hny](#)] Chain Replication (part 1)

```

3  const NREPLICAS = 3    # number of replicas
4  const NOPS = 2         # number of operations (or clients)
5
6  network = {}           # the network is a set of messages
7
8  def send(self, dst, msg):    # send msg to replica dst
9      atomically network |= { (dst, (self, msg)) }
10
11 def broadcast(self, msg):    # broadcast msg to all
12     atomically for dst in {1..NREPLICAS}:
13         network |= { (dst, (self, msg)) }
14
15 def receive(self) returns msgs:    # return messages for me
16     msgs = { payload for (dst, payload) in network where (dst == self) }
17
18 def crash(self):             # server 'self' is crashing
19     broadcast(self, "crash")    # notify all other replicas
20     stop()
21
22 def is_prefix(hist1, hist2) returns result: # hist1 is a strict prefix of hist2
23     result = (len(hist1) < len(hist2)) and
24         all(hist1[i] == hist2[i] for i in {0..len(hist1)-1})

```

Figure 25.4: [[code/chain.hny](#)] Chain Replication (part 2)

Figure 25.3 and Figure 25.4 show an implementation of chain replication. The network is modeled as a append-only set of messages of the form $(destination, (source, payload))$. When sending, a message is atomically added to this set. A client broadcasts its operation to all replicas.

Each replica maintains its own history *hist* and a chain configuration *config*. The replica executes a loop in which it receives and atomically handles messages until it crashes. As before, one of the replicas cannot crash. Because replicas do not want to handle the same message twice, each replica maintains a set *received* of messages it has already handled. Each replica then waits for a message on the network it has not already handled before.

When a replica receives a client request, it adds the request to a set *requests* that it maintains. A replica can only handle such a request if it is the head, but each replica can eventually become the head so it should carefully save all requests. (In theory, it can remove them as soon as they are on its history.) When a replica receives a failure notification, it updates its configuration accordingly. When a non-head replica receives a history that extends its own history, then the replica adopts the received history.

Next, if a replica is the head, it adds any requests it has received to its history unless they are already on there. If a replica is the tail, it “delivers” operations on its history (by printing the operation) that it has not already delivered. For this, it maintains a counter *delivered* that counts the number of delivered requests. Any replica that is not the tail sends its history to its successor in the chain.

The question is whether chain replication has the same behavior as the replicated state machine specification of Figure 25.1. This can be checked using the following two Harmony commands:

```
$ harmony -o rsm.hfa code/rsm.hny
$ harmony -B rsm.hfa code/chain.hny
```

The first command outputs the behavior DFA of `code/rsm.hny` in the file `rsm.hfa`. The second command checks that the behavior of `code/chain.hny` satisfies the DFA in `rsm.hfa`. Note that chain replication does not produce all the possible behaviors of a replicated state machine, but all its behaviors are valid.

The model has each replica send its entire history each time it extends its history. This is fine for modeling, but in practice that would not scale. In practice, a predecessor would set up a TCP connection to its successor and only send updates to its history along the TCP connection. Because TCP connections guarantee FIFO order, this would be identical to the predecessor sending a series of histories, but much more efficient.

Chapter 26

Working with Actions

So far we have mostly modeled concurrent activities using threads. Another way of modeling is by enumerating all the possible state transitions from any given state. For example, this is how things are commonly specified in TLA+. As in TLA+, we call such state transitions *actions*. In this chapter we will revisit modeling chain replication, but this time using actions.

Figure 26.1 and Figure 26.2 contain the new specification. The state of the replicas and the clients are stored in the variables *replicas* and *clients* respectively. Each type of action is captured using a lambda and a method. The method updates the state, while the lambda enumerates the possible actions of this type.

For example, consider the *crash* action. All replicas, except the replica that is immortal and the replicas that have already crashed, can crash. There is a lambda `crash` that generates a set of all possible crashes. Each element in the set is a *thunk*, that is, a delayed call of a method and an argument [Ing61]. For example, `?do_crash(1)` is the action representing replica 1 failing. If we look at the `do_crash(p)` method, all it does is set the *crashed* field of the replica. The specification does this for every type of action:

- `sendOperation`: a client broadcasts an operation to all replicas.
- `gotOperation`: the head replica adds the operation to its history.
- `sendHist`: a replica sends its history to its successor.
- `gotHist`: a replica accepts a history it has received.
- `deliver`: the tail delivers (prints) an operation.
- `crash`: a replica crashes.
- `detect`: a replica detects the crash of a peer.

The Harmony `action` module explores all possible behaviors of such a specification. It has a single method `explore` that takes a set of lambdas, each of which returning a set of possible actions.

So, which of the two types of specification do you prefer? One metric is readability, but that is subjective and depends on what you have experience with. Another object is the size of the state space, and in general control over the state space that is being explored. Threads have hidden state

```

3  import lists, action
4
5  const NREPLICAS = 3
6  const NOPS = 2
7
8  # Global state
9  let immortal = choose {1..NREPLICAS}:
10     replicas = { p: { .immortal: immortal == p, .crashed: False,
11                      .requests: {}, .hist: [], .config: {1..NREPLICAS},
12                      .received: {}, .delivered: 0 } for p in {1..NREPLICAS} }
13     clients = { c: { .sent_request: False } for c in {1..NOPS} }
14
15  const is_head = lambda(p): p == min(replicas[p].config) end
16  const is_tail = lambda(p): p == max(replicas[p].config) end
17
18  def is_successor(self, p) returns result:
19     let succ = { q for q in replicas[self].config where q > self }:
20     result = False if succ == {} else (p == min(succ))
21
22  def do_sendOperation(c):
23     print(c)
24     clients[c].sent_request = True
25     for p in {1..NREPLICAS}:
26         replicas[p].requests |= { c }
27
28  const sendOperation = lambda(): { ?do_sendOperation(c)
29     for c in {1..NOPS} where not clients[c].sent_request } end
30
31  def do_gotOperation(self, op):
32     replicas[self].hist += [op,]
33
34  const gotOperation = lambda(): { ?do_gotOperation(p, op)
35     for p in {1..NREPLICAS}
36     where not replicas[p].crashed and is_head(p)
37     for op in replicas[p].requests
38     where op not in replicas[p].hist } end
39
40  def do_sendHist(self, p):
41     replicas[p].received |= { replicas[self].hist }

```

Figure 26.1: [\[code/chainaction.hny\]](#) Chain Replication specification using actions (part 1)

```

45  const sendHist = lambda(): { ?do_sendHist(p, q)
46      for p in {1..NREPLICAS}
47          where not replicas[p].crashed
48      for q in {1..NREPLICAS}
49          where is_successor(p, q) and (replicas[p].hist not in replicas[q].received)
50      } end
51
52  def do_gotHist(self, hist):
53      replicas[self].hist = hist
54
55  const gotHist = lambda(): { ?do_gotHist(p, hist)
56      for p in {1..NREPLICAS} where not replicas[p].crashed
57      for hist in replicas[p].received where (len(replicas[p].hist) < len(hist))
58          and lists.startswith(hist, replicas[p].hist) } end
59
60  def do_deliver(self):
61      print(self, replicas[self].hist[replicas[self].delivered])
62      replicas[self].delivered += 1
63
64  const deliver = lambda(): { ?do_deliver(p)
65      for p in {1..NREPLICAS} where not replicas[p].crashed and
66          is_tail(p) and (len(replicas[p].hist) > replicas[p].delivered) } end
67
68  def do_crash(self):
69      replicas[self].crashed = True
70
71  const crash = lambda(): { ?do_crash(p)
72      for p in {1..NREPLICAS}
73          where not replicas[p].crashed and not replicas[p].immortal } end
74
75  def do_detect(self, p):
76      replicas[self].config -= { p }
77
78  const detect = lambda(): { ?do_detect(p, q)
79      for p in {1..NREPLICAS} where not replicas[p].crashed
80      for q in {1..NREPLICAS} where replicas[q].crashed and
81          (q in replicas[p].config) } end
82
83  action.explore({sendOperation, gotOperation, sendHist,
84                  gotHist, deliver, crash, detect})

```

Figure 26.2: [\[code/chainaction.hny\]](#) Chain Replication specification using actions (part 2)

such as their stacks, program counters, and local variables, adding to the state space in sometimes unexpected ways. With an action-based specification all state is explicit, and all state changes are explicit. This can be advantageous. On the other hand, the thread-based specification is easier to turn into an actual running implementation.

Chapter 27

Replicated Atomic Read/Write Register

A *register* is an object that you can read or write (Figure 27.1). In a distributed system, examples include a shared file system (each file is a register) or a key/value store (each key corresponds to a register). A simple shared register implementation would have its value maintained by a server, and clients can read or write the shared register by exchanging messages with the server. We call two operations such that one does not finish before the other starts *concurrent*. Since messages are delivered one at a time to the server, concurrent operations on the shared register appear atomic. In particular, we have the following three desirable properties:

1. All write operations are ordered;
2. A read operation returns either the last value written or the value of a concurrent write operation.
3. If read operation r_1 finishes before read operation r_2 starts, then r_2 cannot return a value that is older than the value returned by r_1 .

It is instructive to look at the test program and its output in Figure 27.2. This is for the case when there is only a single reader thread (identified as “1”) and a single writer thread (identified as “-1”), but already there are many cases to consider. Each thread prints information just before and just after doing their single operation. The output shows all possible interleavings in the form of a DFA. Note that if the read operation starts after the write operation has completed, then the read operation must return the new value. However, if the two operations interleave in some way, then the read operation can return either the old or the new value.

Unfortunately, a server is a *single point of failure*: If it fails, all its clients suffer. We would therefore like to find a solution that can survive the crash of a server. While we could use Chain Replication to replicate the register, in this chapter we will use a solution that does not assume that crashes can be accurately detected.

We will again replicate the register object: Maintain multiple copies, but we will not use the replicated state machine approach. One could, for example, imagine that clients write to all copies and read from any single one. While this solves the single-point-of-failure problem, we lose all the

```

1  reg = None
2
3  def init():
4      pass
5
6  def read(uid) returns contents:
7      atomically contents = reg
8
9  def write(uid, v):
10     atomically reg = v

```

Figure 27.1: [\[code/register.hny\]](#) An atomic read/write register

nice properties above. For one, it is not guaranteed that all servers receive and process all write operations in the same order.

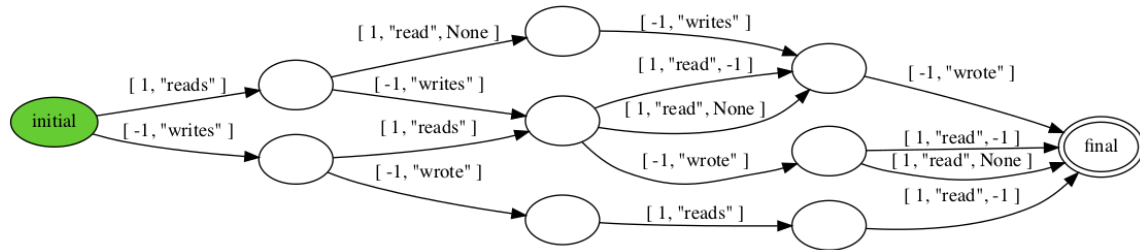
We present a protocol preserving these properties that is based on the work by Hagit Attiya, Amotz Bar-Noy, and Danny Dolev [ABND95]. In order to tolerate F failures, it uses $N = 2F + 1$ servers. In other words, the register survives as long as a strict majority of its copies survive. All write operation will be ordered by a unique *logical timestamp* (see also Chapter 10). Each server maintains not only the value of the object, but also the timestamp of its corresponding write operation.

Each read and write operation consists of two *phases*. In a phase, a client broadcasts a request to all servers and waits for responses from a majority ($N - F$ or equivalently $F + 1$ servers). Note that because we are assuming that no more than F servers can fail, doing so is safe, in that a client cannot indefinitely block as long as that assumption holds.

In the first phase, a client asks each server for its current timestamp and value. After receiving $N - F$ responses, the client determines the response with the highest timestamp. In case of a write operation, the client then computes a new unique timestamp that is strictly higher than the highest it has seen. To make this work, timestamps are actually lexicographically ordered tuples consisting of an integer and the unique identifier of the client that writes the value. So, if (t, c) is the highest timestamp observed by client c' , and c' needs to create a new timestamp, it can select $(t + 1, c')$. After all $(t + 1, c') > (t, u)$ and no other client will create the same timestamp.

Suppose client c' is trying to write a value v . In phase 2, client c' broadcasts a request containing timestamp $(t + 1, c')$ and v . Each server that receives the request compares $(t + 1, c')$ against its current timestamp. If $(t + 1, c')$ is larger than its current timestamp, it adopts the new timestamp and its corresponding value v . In either case, the server responds to the client. Upon c' receiving a response from $N - F$ servers, the write operation completes. In case of a read operation, client c' simply *writes back* the highest timestamp it saw in the first phase along with its corresponding value.

Figure 27.3 contains the code for a server, as well as the code for read and write operations. For efficiency of model checking, the servers are anonymous—otherwise we would have to consider every permutation of states of those servers. Because the servers are anonymous, they may end



```

1  import register
2
3  const NREADERS = 2
4  const NWRITERS = 1
5
6  def reader(i):
7      print(i, "reads")
8      let v = register.read(i):
9          print(i, "read", v)
10
11  def writer(i):
12      print(i, "writes")
13      register.write(i, i)
14      print(i, "wrote")
15
16  register.init()
17  for i in { 1 .. NREADERS }:
18      spawn reader(i)
19  for i in { 1 .. NWRITERS }:
20      spawn writer(-i)

```

Figure 27.2: [\[code/abdtest.hny\]](#) Behavioral test for atomic read/write registers and the output for the case that $NREADERS = NWRITERS = 1$

```

1  import bags
2
3  const F = 1
4  const N = (2 * F) + 1
5
6  network = bags.empty()
7
8  def send(m): atomically network = bags.add(network, m)
9
10 def server():
11     var t, v, received = (0, None), None, {}
12     while True:
13         atomically when exists m in { k for k in keys network - received
14             where k.type in {"read", "write"} }:
15             received |= { m }
16             if (m.type == "write") and (m.value[0] > t):
17                 t, v = m.value
18             send({ .type: .response, .dst: m.src, .value: (t, v) })
19
20 def init():
21     for i in { 1 .. N }: spawn eternal server()
22
23 def receive(uid, phase) returns quorums:
24     let msgs = { m:c for m:c in network
25         where (m.type == .response) and (m.dst == (uid, phase)) }:
26     quorums = bags.combinations(msgs, N - F)
27
28 def read(uid) returns contents:
29     send({ .type: "read", .src: (uid, 1) })
30     atomically when exists msgs in receive(uid, 1):
31         let (t, v) = max(m.value for m in keys msgs):
32         send({ .type: "write", .src: (uid, 2), .value: (t, v) })
33         contents = v
34     atomically when exists msgs in receive(uid, 2):
35         pass
36
37 def write(uid, v):
38     send({ .type: "read", .src: (uid, 1) })
39     atomically when exists msgs in receive(uid, 1):
40         let (t, _) = max(m.value for m in keys msgs)
41         let nt = (t[0] + 1, uid):
42         send({ .type: "write", .src: (uid, 2), .value: (nt, v) })
43     atomically when exists msgs in receive(uid, 2):
44         pass

```

Figure 27.3: [\[code/abd.hny\]](#) An implementation of a replicated atomic read/write register

up sending the same exact message, but clients are waiting for a particular number of messages. Because of this, we will model the network as a bag of messages.

A server initializes its timestamp t to $(0, \text{None})$ and its value to **None**. Each server also keeps track of all the requests its already received so it doesn't handle the same request twice. The rest of the code is fairly straightforward.

Read and write operations are both invoked with a unique identifier uid . Both start by broadcasting a *.read* request to all servers and then waiting for a response from $N - F$ servers. The `receive()` function uses the *bag.combinations* method to find all combinations of subsets of responses of size $N - F$. The second phase of each operation is similar.

[Figure 27.2](#) can be used to test this protocol, although you will notice that the model checker cannot deal with cases involving more than three client threads. Three is just enough to check the third property listed above (using one writer and two readers). Doing so illustrates the importance of the second phase of the **read** operation. You can comment out Lines 34, 36, and 37 in [Figure 27.3](#) and to elide the second phase and see what goes wrong.

One may wonder how failures can occur in this model. They are not explicitly modeled, but Harmony tries every possible execution. This includes executions in which the clients terminate before F of the servers start executing. To the clients, this is indistinguishable from executions in which those servers have failed.

Chapter 28

Distributed Consensus

Distributed consensus is the problem of having a collection of processors agree on a single value over a network. For example, in state machine replication, the state machines have to agree on which operation to apply next. Without failures, this can be solved using leader election: First elect a leader, then have that leader decide a value. But consensus often has to be done in adverse circumstances, for example in the face of processor failures.

Each processor *proposes* a value, which we assume here to be from the set $\{0, 1\}$. By the usual definition of consensus, we want the following three properties:

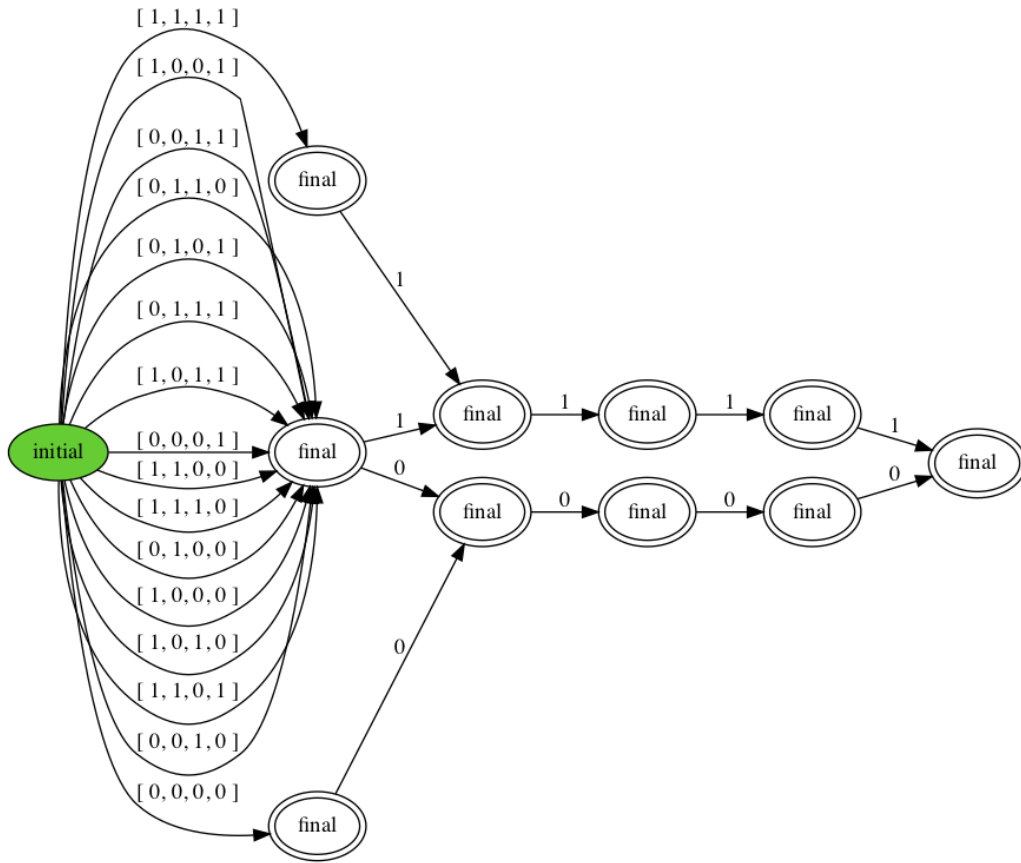
1. *Validity*: a processor can only decide a value that has been proposed;
2. *Agreement*: if two processors decide, then they decide the same value.
3. *Termination*: each processor eventually decides.

The consensus problem is impossible to solve in the face of processor failures and without making assumptions about how long it takes to send and receive a message [FLP85]. Here we will not worry about *Termination*.

Figure 28.1 presents a specification for binary consensus—the proposals are from the set $\{0, 2\}$. In this case there are four processors. The proposal of processor i is in $proposals[i]$. The *decision* is chosen from the set of proposals. Each processor may or may not print the decision—capturing the absence of the *Termination* property. It may be that no decisions are made, but that does not violate either Validity or Agreement. Thus the behavior of the program is to first print the array of proposals, followed by some subset of processors printing their decision. Notice the following properties:

- there are $16 = 2^4$ possible proposal configurations;
- all processors that decide decide the same value;
- if all processors propose 0, then all processors that decide decide 0;
- if all processors propose 1, then all processors that decide decide 1.

This is just the specification—in practice we do not have a shared variable in which we can store the decision a priori. We will present a simple consensus algorithm that can tolerate fewer than



```

1  const N = 4
2
3  proposals = [ choose({0, 1}) for i in {0..N-1} ]
4  decision = choose { x for x in proposals }
5
6  def processor(proposal):
7      if choose { False, True }:
8          print decision
9
10 print proposals
11 for i in {0..N-1}:
12     spawn processor(proposals[i])

```

Figure 28.1: [\[code/consensus.hny\]](#) Distributed consensus code and behavior DFA

$1/3^{\text{rd}}$ of processors failing by crashing. More precisely, constant F contains the maximum number of failures, and we will assume there are $N = 3F + 1$ processors.

Figure 28.2 presents our algorithm. Besides the *network* variable, it uses a shared list of proposals and a shared set of decisions. In this particular algorithm, all messages are broadcast to all processors, so they do not require a destination address. The N processors go through a sequence of *rounds* in which they wait for $N - F$ messages, update their state based on the messages, and broadcast messages containing their new state. The reason that a processor waits for $N - F$ rather than N messages is because of failures: Up to F processors may never send a message and so it would be unwise to wait for all N . You might be tempted to use a timer and time out on waiting for a particular processor. But how would you initialize that timer? While we will assume that the network is reliable, there is no guarantee that messages arrive within a particular time. We call a set of $N - F$ processors a *quorum*. A quorum must suffice for the algorithm to make progress.

The state of a processor consists of its current round number (initially 0) and an estimate (initially the proposal). Therefore, messages contain a round number and an estimate. To start things, each processor first broadcasts its initial round number and initial estimate. The number of rounds that are necessary to achieve consensus is not bounded. But Harmony can only check finite models, so there is a constant `NROUNDS` that limits the number of rounds.

In Line 21, a processor waits for $N - F$ messages using the Harmony **atomically when exists** statement. Since Harmony has to check all possible executions of the protocol, the `receive(round, k)` method returns all *subbags* of messages for the given round that have size $k = N - F$. The method uses a dictionary comprehension to filter out all messages for the given *round* and then uses the *bag.combinations* method to find all combinations of size k . The **atomically when exists** statement waits until there is at least one such combination and then chooses an element, which is bound to the *quorum* variable. The body of the statement is then executed atomically. This is usually how distributed algorithms are modeled, because they can only interact through the network. There is no need to interleave the different processes other than when messages are delivered. By executing the body atomically, a lot of unnecessary interleavings are avoided and this reduces the state space that must be explored by the model checker significantly.

The body of the **atomically when exists** statement contains the core of the algorithm. Note that $N - F = 2F + 1$, so that the number of messages is guaranteed to be odd. Also, because there are only 0 and 1 values, there must exist a majority of zeroes or ones. Variable `count[0]` stores the number of zeroes and `count[1]` stores the number of ones received in the round. The rules of the algorithm are simple:

- update *estimate* to be the majority value;
- if the quorum is unanimous, decide the value.

After that, proceed with the next round.

To check for correct behavior, run the following two commands:

```
$ harmony -o consensus.hfa code/consensus.hny
$ harmony -B consensus.hfa code/bosco.hny
```

Note that the second command prints a warning: “**behavior warning: strict subset of specified behavior.**” Thus, the set of behaviors that our algorithm generates is a subset of the behavior that the specification allows. Figure 28.3 shows the behavior, and indeed it is not the

```

1  import bags
2
3  const F = 1
4  const N = (3 * F) + 1
5  const NROUNDS = 3
6
7  proposals = [ choose({0, 1}) for i in {0..N-1} ]
8  network = bags.empty()
9
10 def broadcast(msg):
11     atomically network = bags.add(network, msg)
12
13 def receive(round, k) returns quorum:
14     let msgs = { e:c for (r,e):c in network where r == round }:
15         quorum = bags.combinations(msgs, k)
16
17 def processor(proposal):
18     var estimate, decided = proposal, False
19     broadcast(0, estimate)
20     for round in {0..NROUNDS-1}:
21         atomically when exists quorum in receive(round, N - F):
22             let count = [ bags.multiplicity(quorum, i) for i in { 0..1 } ]:
23                 assert count[0] != count[1]
24                 estimate = 0 if count[0] > count[1] else 1
25                 if count[estimate] == (N - F):
26                     if not decided:
27                         print estimate
28                         decided = True
29                 assert estimate in proposals    # check validity
30                 broadcast(round + 1, estimate)
31
32 print proposals
33 for i in {0..N-1}:
34     spawn processor(proposals[i])

```

Figure 28.2: [\[code/bosco.hny\]](#) A crash-tolerant consensus protocol

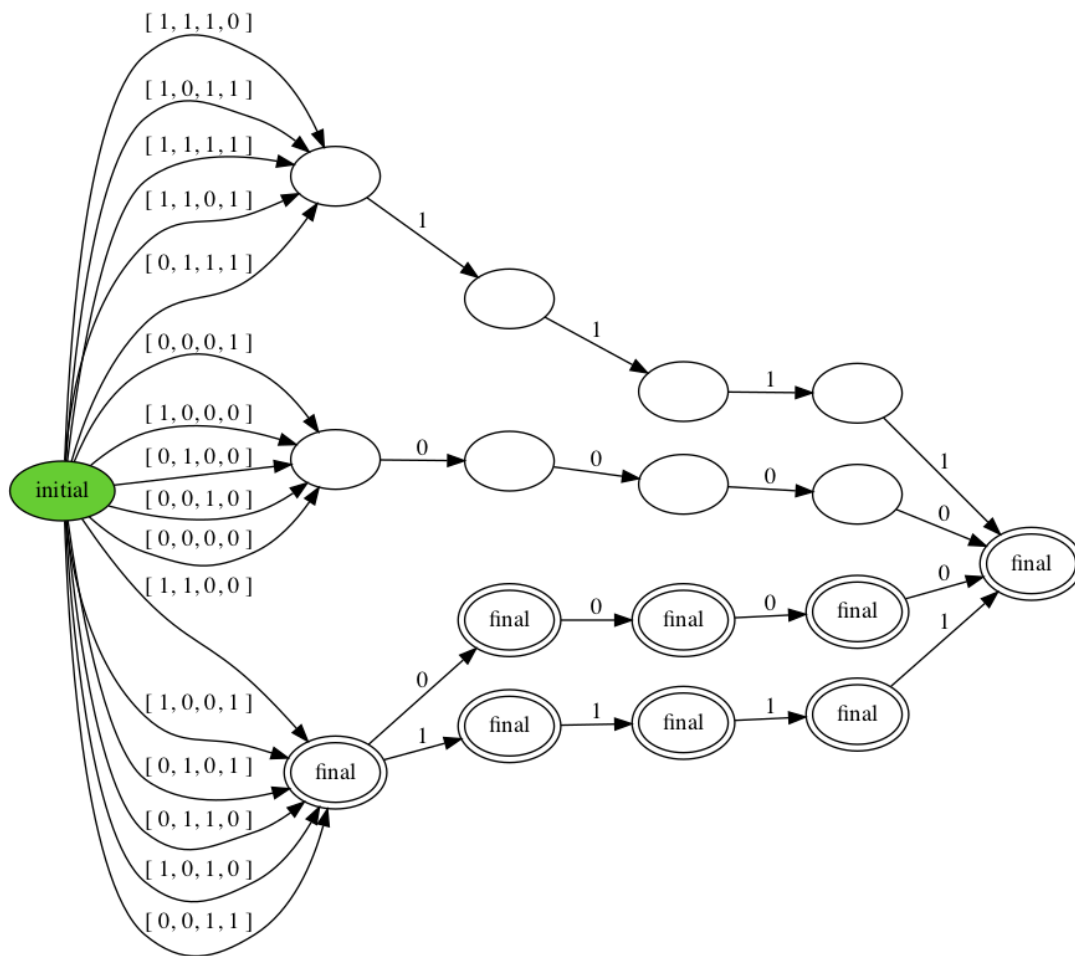


Figure 28.3: The behavior DFA for [Figure 28.2](#)

same as the behavior of [Figure 28.1](#). This is because in our algorithm the outcome is decided a priori if more than twothirds of the processors have the same proposal, whereas in the consensus specification the outcome is only decided a priori if the processors are initially unanimous. Another difference is that if the outcome is decided a priori, all processors are guaranteed to decide.

While one can run this code within little time for $F = 1$, for $F = 2$ the state space to explore is already quite large. One way to reduce the state space to explore is the following realization: Each processor only considers messages for the round that it is in. If a message is for an old round, the processor will ignore it; if a message is for a future round, the processor will buffer it. So, one can simplify the model and have each processor wait for *all* N messages in a round instead of $N - F$. The processor would still have to choose to consider just $N - F$ out of those N messages, but executions in which some processors are left behind in all rounds are no longer considered. It still includes executions where some subset of $N - F$ processors only choose each other messages and essentially ignore the messages of the remaining F processors, so the resulting model is just as good.

Another way to reduce the state space to explore is to leverage symmetry. First of all, it does not matter who proposes a particular value. Also, the values 0 and 1 are not important to how the protocol operates. So, with 5 processors ($F = 2$), say, we only need to explore the cases where no processors propose 1, where exactly one processors proposes 1, and where 2 processors proposes 1.

[Figure 28.4](#) shows the code for this optimized model. Running this with $F = 2$ does not take very long and this approach is a good blueprint for testing other round-based protocols (of which there are many).

Exercises

28.1 The algorithm as given works in the face of crash failures. A more challenging class to tolerate are *arbitrary failures* in which up to F processors may send arbitrary messages, including conflicting messages to different peers (equivocation). The algorithm can tolerate those failures if you use $N = 5F - 1$ processors instead of $N = 3F - 1$. Check that.

28.2 In 1983, Michael Ben-Or presented a randomized algorithm that can tolerate crash failures with just $N = 2F - 1$ processors [\[BO83\]](#). Implement this algorithm.

```

1  import bags
2
3  const F = 1
4  const N = (3 * F) + 1
5  const NROUNDS = 3
6
7  let n_zeroes = choose { 0 .. N / 2 }:
8      proposals = ([0,] * n_zeroes) + ([1,] * (N - n_zeroes))
9  network = bags.empty()
10
11 def broadcast(msg):
12     atomically network = bags.add(network, msg)
13
14 def receive(round) returns quorum:
15     let msgs = { e:c for (r,e):c in network where r == round }:
16         quorum = {} if bags.size(msgs) < N else { msgs }
17
18 def processor(proposal):
19     var estimate, decided = proposal, False
20     broadcast(0, estimate)
21     for round in {0..NROUNDS-1}:
22         atomically when exists msgs in receive(round):
23             let choices = bags.combinations(msgs, N - F)
24             let quorum = choose(choices)
25             let count = [ bags.multiplicity(quorum, i) for i in { 0..1 } ]:
26                 assert count[0] != count[1]
27                 estimate = 0 if count[0] > count[1] else 1
28                 if count[estimate] == (N - F):
29                     if not decided:
30                         print estimate
31                         decided = True
32                         assert estimate in proposals          # validity
33                         broadcast(round + 1, estimate)
34
35 print proposals
36 for i in {0..N-1}:
37     spawn processor(proposals[i])

```

Figure 28.4: [\[code/bosco2.hny\]](#) Reducing the state space

Chapter 29

Paxos

Paxos [Lam98] is the most well-known family of consensus protocols for environments in which few or no assumptions are made about timing. In this chapter, we present a basic version of a Paxos protocol, one that is *single-decree* (only tries to make a single decision). It uses two kinds of processors: *leaders* and *acceptors*. In order to tolerate F crash failures, you need at least $F + 1$ leaders and $2F + 1$ acceptors, but leaders and acceptors can be colocated, so in total only $2F + 1$ independently failing processors are needed. Here we provide only a rudimentary introduction to Paxos; for more detailed information refer to [Lam98].

As in the consensus protocol of Chapter 28, Paxos uses rounds of messaging. The communication pattern, however, is different. Similar to the atomic read/write register protocol in Chapter 27, Paxos uses two kinds of rounds: “Phase 1” and “Phase 2” rounds. Rounds are identified by a so-called *ballot number* combined with the phase number. Different leaders are in charge of different ballot numbers. Leaders broadcast “Type A” messages to the acceptors, which respond point-to-point with “Type B” messages.

Figure 29.1 and Figure 29.2 contain the code for this Paxos protocol. Paxos is perhaps best understood starting with the second phase. At the end of the first phase, the leader broadcasts a 2.A message (Phase 2, Type A) to the acceptors containing the ballot number and a proposal and then waits for $N - F$ matching 2.B responses from the acceptors. If each response contains the ballot number and the proposal, then the proposal is deemed decided. But one or more of the responses can contain a higher ballot number, in which case the leader has to try again with an even higher ballot number. This is where the first phase comes in.

It is not possible that an acceptor responds with a smaller ballot number. This is because acceptors maintain two state variables. One is *ballot*, the highest ballot number they have seen. Second is a variable called *last_accepted* that, if not **None**, contains the last proposal the acceptor has *accepted* and the corresponding ballot number. The acceptor also contains a set *received* that contains (ballot, phase) tuples identifying all rounds that the ballot has already participated in. An acceptor waits for a message for a round that is not in *received*. If its ballot number is higher than what it has seen before, the acceptor moves into that ballot. If the phase is 2, then the acceptor accepts the proposal and remembers when it did so by saving the (ballot, proposal) tuple in *last_accepted*. In all cases, the acceptor responds with the current values of *ballot* and *last_accepted*.

In its first phase, a leader of a ballot must come up with a proposal that cannot conflict with a proposal of an earlier ballot that may already have been decided. To this end, the leader broadcasts

```

3  import bags
4
5  const F = 1
6  const NACCEPTORS = (2 * F) + 1
7  const NLEADERS = F + 1
8  const NBALLOTS = 2
9
10 network = bags.empty()
11
12 proposals = [ choose({0, 1}) for i in {0..NLEADERS-1} ]
13
14 def send(msg):
15     atomically network = bags.add(network, msg)
16
17 def receive(ballot, phase) returns quorum:
18     let msgs = { e:c for (b,p,t,e):c in network
19                 where (b,p,t) == (ballot, phase, "B") } :
20     quorum = bags.combinations(msgs, NACCEPTORS - F)
21
22 print proposals
23 for i in {0..NLEADERS - 1}:
24     spawn leader(i + 1, proposals[i])
25 for i in {1..NACCEPTORS}:
26     spawn eternal acceptor()

```

Figure 29.1: code/paxos.hny A version of the Paxos protocol, Part 1

```

30 def leader(self, proposal):
31     var ballot, estimate, decided = self, proposal, False
32     send(ballot, 1, "A", None)
33     while ballot <= NBALLOTS:
34         atomically when exists quorum in receive(ballot, 1):
35             let accepted = { e for e: in quorum where e != None }:
36             if accepted != {}:
37                 _, estimate = max(accepted)
38             send(ballot, 2, "A", estimate)
39         atomically when exists quorum in receive(ballot, 2):
40             if bags.multiplicity(quorum, (ballot, estimate)) == (NACCEPTORS - F):
41                 assert estimate in proposals # validity
42                 if not decided:
43                     print estimate
44                     decided = True
45                 ballot += NLEADERS
46             if ballot <= NBALLOTS:
47                 send(ballot, 1, "A", None)
48
49 def acceptor():
50     var ballot, last_accepted, received = 0, None, {}
51     while True:
52         atomically when exists b,p,e in { (bb,pp,ee) for bb,pp,tt,ee: in network
53         where ((bb,pp) not in received) and (tt == "A") }:
54             received |= { (b, p) }
55         if b >= ballot:
56             ballot = b
57             if p == 2:
58                 last_accepted = (ballot, e)
59             send(b, p, "B", last_accepted)

```

Figure 29.2: [\[code/paxos.hny\]](#) A version of the Paxos protocol, Part 2

a 2.A message to the acceptors and awaits $N - F$ of their *last_accepted* values. If all those acceptors responded with **None**, then the leader is free to choose its own proposal. Otherwise the leader updates its proposal with the one corresponding to the highest ballot number. The leader then moves on to the second round.

To run and check the Paxos code, do the following (leveraging the consensus specification of [Figure 28.1](#)):

```
$ harmony -o consensus.hfa -cN=2 code/consensus.hny
$ harmony -B consensus.hfa code/paxos.hny
```

You should get a warning that our implementation of Paxos does not generate all possible behaviors. This is because we only run the protocol for a finite number of ballots, and therefore at least one of the ballots will be successful. With an unlimited number of ballots, Paxos may never decide unless you make some liveness assumptions.

Exercises

29.1 Perhaps the trickiest detail of the algorithm is that, in Line 37 of [Figure 29.2](#), the leader selects the proposal with the highest ballot number it receives. Replace the **max** operator in that statement with **choose** and see if it finds a problem. First try with `NBALLOTS = 2` and then with `NBALLOTS = 3`. (Warning, the latter may take a long time.) If it finds a problem, analyze the output and see what went wrong.

29.2 [\[MWA⁺19\]](#) discusses a buggy version of Paxos. In this version, the responses to the second phase are matched not by ballot number but by the value of the proposal. Implement this version and, using Harmony, find the problem this causes.

Chapter 30

Needham-Schroeder Authentication Protocol

The Needham-Schroeder protocol [NS78] is a security protocol in which two parties authenticate one another by exchanging large and recently created random numbers called *nonces* that nobody else should be able to read. The nonces should only be used once for an instantiation of the protocol between honest participants (i.e., participants that follow the protocol). The version we describe here uses *public key cryptography* [DH76]: with public key cryptography it is possible to create a message for a particular destination that only that destination can read. We denote with $\langle m \rangle_p$ a message m encrypted for p so that only p can decrypt the message and see that it contains m .

Suppose Alice wants to communicate with Bob. The three critical steps in the Needham-Schroeder protocol are as follows:

1. Alice creates a new nonce N_A and sends $\langle 1, A, N_A \rangle_{\text{Bob}}$ to Bob;
2. Upon receipt, Bob creates a new nonce N_B and sends $\langle 2, N_A, N_B \rangle_{\text{Alice}}$ to Alice;
3. Alice sends $\langle 3, N_B \rangle_{\text{Bob}}$ to Bob.

When Bob receives $\langle 1, A, N_A \rangle_{\text{Bob}}$, Bob does not know for sure that the message came from Alice, and even if it came from Alice, it does not know if Alice sent the message recently or if it was replayed by some adversary. When Alice receives $\langle 2, N_A, N_B \rangle_{\text{Alice}}$, Alice *does* know that, if Bob is honest, (1) Bob and only Bob could have created this message, and (2) Bob must have done so recently (since Alice created N_A). When Bob receives $\langle 3, N_B \rangle_{\text{Bob}}$, Bob decides that it is Alice that is trying to communicate at this time. Since Bob created N_B recently and sent it encrypted to Alice, Bob does not have to worry that the type 3 message was an old message that was replayed by some adversary. Also, if Alice is honest, it seems only Alice can have seen the message containing N_B .

Thus, the intended security properties of this protocol are symmetric. Assuming Alice and Bob are both honest:

- if Alice finishes the protocol with Bob and received B_N from Bob, then nobody but Alice and Bob can learn N_B .

```

1  network = {}
2
3  dest = choose({ None, "Bob", "Corey" })
4
5  def send(msg):
6      atomically network |= { msg }
7
8  def alice():
9      if dest != None:
10         send({ .dst: dest,
11                .contents: { .type: 1, .nonce: "nonceA", .initiator: "Alice" } })
12         atomically when exists m in network when (m.dst == "Alice")
13             and (m.contents.type == 2) and (m.contents.nonce == "nonceA"):
14             send({ .dst: dest, .contents: { .type: 3, .nonce: m.contents.nonce2 } })
15
16  def bob():
17      atomically when exists m in network when (m.dst == "Bob")
18          and (m.contents.type == 1) and (m.contents.initiator == "Alice"):
19          send({ .dst: "Alice",
20                 .contents: { .type: 2, .nonce: m.contents.nonce, .nonce2: "nonceB" } })
21      atomically when exists m in network when (m.dst == "Bob")
22          and (m.contents.type == 3) and (m.contents.nonce == "nonceB"):
23          assert dest == "Bob"
24
25  def corey():
26      var received, nonces, msgs = {}, { "nonceC" }, {}
27      while True:
28          atomically when exists m in network - received when m.dst == "Corey":
29              received |= { m }
30              nonces |= { m.contents.nonce }
31              if m.contents.type == 2:
32                  nonces |= { m.contents.nonce2 }
33              for dst in { "Alice", "Bob" } for n in nonces:
34                  msgs |= { { .dst: dst, .contents: { .type: 1, .nonce: n, .initiator: ini } }
35                           for ini in { "Alice", "Bob" } }
36                  msgs |= { { .dst: dst, .contents: { .type: 2, .nonce: n, .nonce2: n2 } }
37                           for n2 in nonces }
38                  msgs |= { { .dst: dst, .contents: { .type: 3, .nonce: n } } }
39              send(choose(msgs - network))
40
41  spawn alice()
42  spawn eternal bob()
43  spawn eternal corey()

```

Figure 30.1: [\[code/needhamschroeder.hny\]](https://code/needhamschroeder.hny) Needham-Schroeder protocol and an attack

- if Bob finishes the protocol with Alice and received A_N from Alice, then nobody but Bob and Alice can learn N_A .

After the protocol, Alice can include N_A in messages to Bob and Bob can include N_B in messages to Alice to authenticate the sources of those messages to one another.

Figure 30.1 shows the protocol implemented in Harmony. A message $\langle m \rangle_p$ is encoded in Harmony as a dictionary $\{\text{.dst} : p, \text{.contents} : m\}$. The code for Alice and Bob simply follows the steps listed above.

Unfortunately, the protocol turns out to be incorrect, but it took 17 years before somebody noticed [Low95]. Model checking can be used to find the bug [Low96]. To demonstrate the bug, we need to model the environment. In particular, we introduce a third party, which we will call Corey. We want to make sure that Corey cannot impersonate Alice or Bob. However, it is possible that Alice tries to set up an authenticated connection to Corey using the Needham-Schroeder protocol. That in itself should not be a problem if the protocol were correct.

The code in Figure 30.1 has Alice either not do anything, or has Alice try to set up a connection to either Bob or Corey. Bob only accepts connections with Alice. Corey, when receiving a message that it can decrypt, will try to find an attack by sending every possible message to every possible destination. In particular, it keeps track of every nonce that it has seen and will try to construct messages with them to send to Alice and Bob. If Bob finishes the protocol, it checks to see if Alice actually tried to connect to Bob. If not, the assertion fails and an attack is found.

Running the code in Figure 30.1 quickly finds a viable attack. The attack goes like this:

1. Alice creates a new nonce N_A and sends $\langle 1, A, N_A \rangle_{\text{Corey}}$ to Corey;
2. Upon receipt, Corey sends $\langle 1, A, N_A \rangle_{\text{Bob}}$ to Bob;
3. Upon receipt, Bob, believing it is engaging in the protocol with Alice, creates a new nonce N_B and sends $\langle 2, N_A, N_B \rangle_{\text{Alice}}$ to Alice;
4. Alice thinks the message came from Corey (because it contains N_A , which Alice created for Corey and sent to Corey) and sends $\langle 3, N_B \rangle_{\text{Corey}}$ to Corey.
5. Corey learns N_B and sends $\langle 3, N_B \rangle_{\text{Bob}}$ to Bob.
6. Bob receiving $\langle 3, N_B \rangle_{\text{Bob}}$ is identical to the situation in which Alice tried to set up a connection to Bob, so Bob now thinks it is talking to Alice, even though Alice never tried to communicate with Bob.

The security property is violated. In particular, Bob, duped by Corey, finished the protocol with Alice and received A_N , and even though Bob and Alice are both honest, Corey has a copy of A_N . So, Corey is now able to impersonate Alice to Bob (but not vice versa because Alice did not try to authenticate Bob).

Exercises

30.1 Figure out how to fix the protocol.

30.2 There were two versions of the Needham-Schroeder protocol: the Symmetric Key protocol and the Public Key protocol. In this chapter we only discussed the latter, but the former also had a problem. See if you can find it using Harmony.

Bibliography

- [ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.
- [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.
- [BO83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, PODC’83, pages 27–30, New York, NY, USA, 1983. ACM.
- [CES71] Edward G. Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.
- [CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.
- [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.
- [CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.
- [DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

- [Dij62] Edsger W. Dijkstra. EWD-35: Over de sequentialiteit van procesbeschrijvingen. circulated privately, approx. 1962.
- [Dij64] Edsger W. Dijkstra. EWD-108: Een algorithmen ter voorkoming van de dodelijke omarming. circulated privately, approx. 1964.
- [Dij65a] Edsger W. Dijkstra. EWD-123: Cooperating Sequential Processes. circulated privately, 1965.
- [Dij65b] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–569, September 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.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [Gra78] Jim N. Gray. *Notes on data base operating systems*, pages 393–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978.
- [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.
- [Ing61] Peter Z. Ingerman. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM*, 4(1):55–58, jan 1961.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.
- [Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

- [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.
- [Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, November 1995.
- [Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, Berlin, Heidelberg, 1996. 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.
- [MWA⁺19] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys ’19, New York, NY, USA, 2019. Association for Computing Machinery.
- [NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.
- [Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115 – 116, 1981.
- [Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [Sch97] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, Berlin, Heidelberg, 1997.
- [SS83] Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.
- [vRS04] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, San Francisco, California, USA, December 6-8, 2004, pages 91–104. USENIX Association, 2004.

Appendix A

Harmony Language Reference

A.1 Value Types and Operators

[Chapter 4](#) provides an introduction to Harmony values. Below is a complete list of Harmony value types with examples:

Type	Name	Example
Boolean	"bool"	False , True
Integer	"int"	..., -2, -1, 0, 1, 2, ...
String	"str"	"example", <i>.example</i>
Bytecode Address	"pc"	(method names and lambdas)
List	"list"	[1, 2, 3,], ((1, 2), 3), [1], ()
Dictionary	"dict"	{ <i>.account</i> : 12345, <i>.valid</i> : False }, {::}
Set	"set"	{}, { 1, 2, 3 }, { False , <i>.id</i> , 3 }
Address	"address"	?lock, ?flags[2], ?f(2), None
Context	"context"	(generated by stop or save expression)

In Harmony, there is no distinction between tuples (denoted with parentheses) and lists (denoted by square brackets). That is, their format is either (e, e, \dots, e) or $[e, e, \dots, e]$. They map indexes (starting at 0) to Harmony values. If the list has two or more elements, then the final comma is optional.

Method **type** e returns the type name of Harmony value e .

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;
- strings are lexicographically ordered;
- bytecode addresses are ordered by their integer values;

- lists are lexicographically ordered;
- 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;
- Except for **None**, an address is a pair of a function and a list of arguments. Addresses are lexicographically ordered accordingly. **None** is the smallest address.
- contexts (Section C.3) are ordered deterministically in an unspecified way.

Harmony supports the following comparison operators:

$e == e$	equivalence
$e != e$	inequivalence
$e < e, e <= e, e > e, e >= e$	ordering

In Harmony programs, the ordering operators are only allowed on pairs of integers, pairs of strings, pairs of lists, or pairs of sets. In the case of strings and lists, the ordering is lexicographical. In the case of sets, the ordering is partial based on the subset relation. Comparison operators can be chained: $x <= y == z$ means $x \leq y \wedge y = z$, although y is evaluated once in the former and twice in the latter expression. Note that evaluation of a chain stops as soon as one of the comparisons fails. So, $1 < 0 < x$ does not evaluate x . (This is called *short-circuiting*.)

Boolean

The boolean type has only two possible values: **False** and **True**. Unlike Python, in Harmony booleans are distinct from integers, and in particular **False** \neq 0. In statements and expressions where booleans are expected, it is not possible to substitute values of other types. That is, unlike Python, Harmony has no notion of *truthiness* of non-boolean values.

Operations on booleans include:

$e \text{ and } e$	conjunction
$e \text{ or } e$	disjunction
$e ==> e$	implication
not e	negation
$v \text{ if } e \text{ else } v'$	v or v' depending on e
any s	disjunction for set or list s
all s	conjunction for set or list s

The meanings of **or**, **and**, and $==>$ are perhaps best explained by putting them in terms of the ternary **if else** operator:

- $x \text{ or } y$ means **True if** x **else** y

- x **and** y means **False if not x else y**
- $x \Rightarrow y$ means **True if not x else y**

Consistent with Python, this means that the result of the operation may not be a Boolean. For example, **(False or 2) == 2**. Also, the operators are not commutative. **2 or False** is an illegal expression because you cannot use an integer as a condition for **if**. We recommend using only Booleans for these operators, so that their outputs are also guaranteed to be a Boolean. Finally, note that the righthand side of a boolean expression may not be evaluated. For example, **True or x** evaluates to **True** without evaluating x (another example of short-circuiting).

Integer

The integer type supports any whole number. Harmony supports decimal integers, hexadecimal integers (start with '0x'), binary integers (start with '0b'), and octal integers (start with '0o'). In the model checker, integers are currently implemented by two's complement 60-bit words. The model checker checks for overflow on various operations.

Operations on integers include:

$-e$	negation
$e + e$	sum
$e - e$	difference
$e * e$	product
$e / e, e // e$	integer division
$e \% e, e \bmod e$	integer division remainder
$e ** e$	power
$\sim e$	binary inversion
$e \& e$	binary and
$e e$	binary or
$e \wedge e$	binary exclusive or
$e << e$	binary shift left
$e >> e$	binary shift right
$\{ e..e' \}$	set of integers from e to e' inclusive
abs e	absolute value
bin e	string containing e in binary format
hex e	string containing e in hexadecimal format
oct e	string containing e in octal format

String

A string is a sequence of zero or more unicode characters. If it consists entirely of alphanumerical characters or underscore characters and does not start with a digit, then a string can be represented by a “.” followed by the characters. For example, *.example* is the same as the string "example".

Native operations on strings include the following:

$s[k]$	indexing
$s[k] = v$	updating the character at index k to string v
$s\ s\ \dots$	concatenation
$s + s$	concatenation
$s * n$	n copies of s concatenated
v [not] in s	check if v is [not] a substring in s
del $s[x]$	remove the character from string variable s at index x
len s	the length of s
keys s	the set $\{0 \dots \mathbf{len}(s) - 1\}$
str e	string representation of any value e
int e	convert e to an integer
int ($e, base$)	convert e to an integer in the given base
list e	the list containing the characters in e
set e	the set containing the characters in e

Set

In Harmony, you can create a set of any collection of Harmony values. Its syntax is $\{v_0, v_1, \dots\}$. Python users: Note that in Harmony the empty set is denoted as $\{\}$. (In Python, $\{\}$ means the empty dictionary, which is represented as $\{:\}$ in Harmony.) As in Python, you can also use **set**() to create an empty set, or **set**(t) to create a set of the elements in a list t .

The **sets** module ([Section B.6](#)) contains various convenient routines that operate on sets. Native operations on sets include:

len s	cardinality
$s < s$	strict subset
$s \leq s$	subset
$s > s$	strict superset
$s \geq s$	superset
$s - s$	set difference
$s \& s$	intersection
$s s$	union
$s \wedge s$	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
list s	sorted list of the elements of set s
sorted s	sorted list of the elements of set s
sum s	compute the sum of a set of integers

Harmony also supports *set comprehension*. In its simplest form, $\{\mathbf{f}(v) \textbf{ for } v \textbf{ in } s\}$ returns a set that is constructed by applying \mathbf{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: $\{x + y \textbf{ for } x \textbf{ in } s \textbf{ for } y \textbf{ in } s \textbf{ where } (x * y) == 4\}$ returns a set that is constructed by summing pairs of elements from s that, when multiplied, have the value 4.

List or Tuple

In Harmony, there is no distinction between a list or a tuple. 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 the same as the value 1.

The **lists** module ([Section B.5](#)) contains various convenient routines that operate on lists or tuples. Native operations on lists or tuples include the following:

$t[k]$	indexing
$t[k] = v$	updating entry k to v
$t + t$	concatenation
$t + t$	concatenation
$t * n$	n copies of t concatenated
$v \textbf{ [not] in } t$	check if v is [not] a value in t
del $t[x]$	remove the entry at index x
len t	the length of t
keys t	the set $\{0 .. \mathbf{len}(t) - 1\}$
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
reversed t	the elements of list t in reverse order
set t	create a set from the elements in a list
sorted t	sorted list from the elements of list t
sum t	compute the sum of a list of integers
zip $[t1, ...]$	zip together lists $t1, ...$

Lists and tuples support comprehension. In its most basic form: $[\mathbf{f}(v) \textbf{ for } v \textbf{ in } t]$. For example, to check if any element in a list t is even, you can write: **any** $((x \% 2) == 0 \textbf{ for } x \textbf{ in } t)$.

The domain of a list L of length n , interpreted as a function, are the integers $0..n-1$. It is illegal to read $L[n]$. However, unlike Python, it is possible to write into $L[n]$. For example, if variable x contains $[1, 2]$, then the statement $x[2] = 3$ results in x having the value $[1, 2, 3]$.

Dictionary

A dictionary maps a set of values (known as *keys*) to another set of values. The generic syntax of a dictionary is $\{k_0 : v_0, k_1 : v_1, \dots\}$. Different from Python, the empty dictionary is written as $\{:\}$ (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 can also be created with the **dict** builtin function. Its argument is a set or a list of (key, value) pairs. In case of a set and there are multiple pairs with the same key, only the one with the maximum value survives. In case of a list and there are multiple pairs with the same key, only the last one in the list survives.

Dictionaries support comprehension. The basic form is: $\{ \mathbf{f}(v) : \mathbf{g}(v) \text{ for } v \text{ in } s \}$.

Operations on dictionaries include the following:

$d \ k$	indexing
$d \ k = v$	updating field k to v
del $d[k]$	remove the entry with key k
len d	the number of keys in d
keys d	the set of keys in d
set d	the set of keys in d
list d	the list of keys in d
k [not] in d	check if k is [not] a key 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
$d \ \& \ d \ \& \ \dots$	dictionary intersection
$d \ \ d \ \ \dots$	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 a string like *.id*, then you might prefer the notation $k.id$.

A bag (or multiset) is often represented by a dictionary that maps each element to its multiplicity. For example, $\{ 10:2, 12:1 \}$ is the bag containing two copies of 10 and one copy of 12. 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.

The **bags** module ([Section B.3](#)) contains various convenient routines that operate on bags. Native operations on bags include the following:

v [not] in b	check if v is [not] in b
$t \ \& \ t \ \& \ \dots$	bag intersection
$t \ \ t \ \ \dots$	bag union

Bytecode Address

A *bytecode address* is an integer that can be used to index into Harmony bytecode. When you define a method or a lambda function, you are creating a constant of the bytecode address type. You can create lambda functions similarly to Python, except that the expression has to end on the keyword **end**. For example: **lambda**(*x,y*): *x+y* **end**.

Address

A Harmony address is a type of *thunk* consisting of a curried function and a list of arguments [Ing61]. A thunk delays the invocation of the curried function. A function can be a constant or a variable and the arguments are all Harmony values. Given an address $p = ?a[b][c]...$, you can use the notation $!p$ to evaluate it. Harmony will first evaluate a , then apply the result to b , then apply the result to c , and so on.

As a simple example, $?5$ is the address of the constant 5, and therefore $!5$ evaluates to 5. Now consider the following program:

```
1  let p = ?5:
2      assert !p == 5
3      !p = 5
4      !p = 4
```

The only line in this program that fails is the last one, as you are not allowed to store 4 at the address of 5.

a can be a constant that maps Harmony values to Harmony values: dictionaries, lists, and strings. In that case, $?a[b]$ refers to the value of entry b in a .

The most common use of addresses is when a is a shared variable. In that case $!a$ evaluates to the current value of a .

Finally, a can be a bytecode address (method or lambda). $?a(b)$ is then the thunk representing a delayed call to method a and argument b . In this case, $!a(b)$ evaluates $a(b)$. This can be used to implement *generators*. For example, the following Harmony program, perhaps surprisingly, does not run into failing assertions:

```
1  counter = 0
2
3  def f():
4      counter += 1
5      result = counter
6
7  let p = ?f():
8      if !p != 1: assert False
9      if !p != 2: assert False
10     if !p != 3: assert False
```

Internally, Harmony uses the address of a method variable and sometimes you see them on the stack during a computation. If k is a method variable, then its address is output as $?@k$. However,

```

1  def fork():
2      atomically:
3          let (r, ctx) = save True:
4              result = r
5          if r:
6              go ctx (False, None)
7
8  def main():
9      if fork():
10         print "parent"
11     else:
12         print "child"
13
14  spawn eternal main()

```

Figure A.1: Using **save** and **go** to implement **fork()**

at the Harmony language level there is no such thing as the address of a local variable. Consider the following two programs:

1	$x = 1$	1	var $x = 1$
2	let $p = ?x$:	2	let $p = ?x$:
3	$x = 2$	3	$x = 2$
4	assert $!p == 2$	4	assert $!p == 1$

In the program on the left, x is a shared variable, and $?x$ is the location of variable x . In the program on the right, x is a local variable. $?x$ evaluates x and then takes its address, so in this case $?x$ equals $?1$.

Like C, Harmony supports the shorthand $p \rightarrow v$ for the expression $(!p).v$.

Context

A context value (a.k.a. continuation) captures the state of a thread. A context is itself composed over various Harmony values.

The following operations generate contexts:

save e	returns a Harmony value
stop p	saves context in $!p$ and stops the thread

The **save** e expression, when invoked, returns a tuple (e, c) where c is the context value of the thread right after the **save** instruction finished. Using **go** c r the context can be turned into a new

thread that, instead, returns r from the **save** e expression. See [Figure A.1](#) for an example of how this could be used to *fork* a thread into two threads.

The **stop** p expression stores the context of the thread right after the expression in $!p$ (i.e., p must be an address value) and terminates the thread. The thread can later be reinstantiated with **go** $!p$ r , in which case the **stop** expression returns r . A thread can be for ever suspended using **stop** **None** or just **stop**().

A.2 Statements

Harmony currently supports the following statements (below, S is a list of statements and an *lvalue* is an expression you can use on the left-hand side of an assignment statement):

e	e is an expression
$lv = [lv =] \dots e$	lv is an lvalue and e is an expression
$lv [op] = e$	op is one of $+$, $-$, $*$, $/$, $//$, $\%$, $\&$, $ $, \wedge , and , or
assert b [$, e$]	b is a boolean. Optionally report value of expression e
await b	b is a boolean
const $a = e$	a is a bound variable, e is a constant expression
def m a [returns v]: S	m is an identifier, a a bound variable, and v a variable
del lv	delete lv
finally e	e is a boolean expression that must hold in each final state
for $a[:b]$ in e [where c]: S	a and b are bound variables, e is a set, dictionary, or string
from m import ...	m identifies a module
global v , ...	v is a shared global variable
go c e	c is a context, e is an expression
if b : S else : S	b is a boolean, S is a list of statements
import m , ...	m identifies a module
invariant e	e is an invariant (must always hold)
let $a = e$: S	a is a bound variable, e is an expression
pass	do nothing
print e	e is an expression
sequential v , ...	v has sequential consistency
spawn [eternal] lv [$, t$]	lv is an lvalue expression, t is the thread-local state
trap lv	lv is an lvalue expression
var $v = e$	v is a new variable, e is an expression
when b : S	b is a boolean, S a list of statements
when exists a in e : S	a is a bound variable, e is an expression
while b : S	b is a boolean, S a list of statements

- Bound variables are read-only.
- A statement can be preceded by the **atomically** keyword to make the statement atomic.
- Multiple **for** statements can be combined into a single statement.
- Multiple **let** and **when** statements can be combined into a single statement.

Single expression evaluation

Any expression by itself can be used as a statement. The most common form of this is a function application, for example: `f()`. This statement evaluates `f()` but ignores its result. It is equivalent to the assignment statement `_ = f()`.

Assignment

The statement `x = 3` changes the state by assigning 3 to variable `x`. `x` may be a local variable or a shared variable. The statement `x = y = 3` first updates `y`, then `x`. The statement `x[f()] = y[g()] = h()` first evaluates `f()` to compute the address of `x[f()]`, then evaluates `g()` to compute the address of `y[g()]`, then evaluates `h()`, then assigns the resulting value to `y[g()]` (using its previously computed address), and finally assigns the same value to `x[f()]` (again using its previously computed address). The statement `a, b = c` assumes that `c` is a tuple with two values. It first evaluates the addresses of `a` and `b` and first assigns to the latter and then the former. If `c` is not a tuple with two values, then Harmony will report an error.

Assigning to `_` (underscore) evaluates the righthand side expression but is otherwise a no-op. The left-hand side can also contain constants. For example `(3, x) = (3, True)` assigns `True` to `x`. However, `(3, x) = (4, True)` fails.

The statement `x += 3` loads `x`, adds 3, and then stores the results in `x`. In this case, it is equivalent to `x = x + 3`. However, in general this is not so. For example, `x[f()] += 3` only evaluates `f()` once. Unlike Python, however, `x += [3,]` is equivalent to `x = x + [3,]` in Harmony. (In Python, the following two compound statements lead to different results for `y`: `x = y = []`; `x += [3]` and `x = y = []`; `x = x + [3]`.)

assert

The statement `assert b` evaluates `b` and reports an error if `b` is false. It should be considered a no-op—it is part of the *specification*, not part of the *implementation* of the algorithm. In particular, it specifies an invariant: Whenever the program counter points at the location where the `assert` statement is, then `b` is always true. The expression `b` is evaluated atomically. Moreover, the expression is not allowed to change the state. If it does change the state, Harmony will report an error.

As in Python, you can specify an additional expression: `assert b, e`. The value of `e` will be reported as part of the error should `b` evaluate to false.

atomically

A statement can be preceded by the **atomically** keyword to make the statement atomic. The statement **atomically**: `S1`; `S2`; ... evaluates statements `S1`, `S2`, ... atomically. This means that the statement runs indivisibly—no other thread can interleave in the atomic statement. The only exception to this is if the atomic block executes a **stop** expression. In this case, another thread can run. When the original thread is resumed (using a **go** statement), it is once again atomically executing.

atomically statements are useful for specification and implementing synchronization primitives such as test-and-set. They are also useful for testing. They are not a replacement for lock/unlock

and should not be used for synchronization in general implementation code. Lock/unlock does allow other threads to run concurrently—just not in the same critical section.

await

The statement **await** *b* is equivalent to **when** *b*: **pass**. It is intended to improve readability of your code.

const

The expression **const** *N* = 3 introduces a new constant *N* with the value 3. Evaluating *N* does not lead to loading from a memory location. The assignment can be overridden with the **-c** flag: **harmony -cN=4** executes the model checker with 4 assigned to *N* instead of 3. Harmony also supports **const** *N*, *M* = 3, 4, which assigns 3 to *N* and 4 to *M*. Harmony has limited support for *constant folding*. For example, **const** *N* = 3 + 4 assigns value 7 to constant *N*.

def

The statement **def** *m* *a* [**returns** *r*]: *S1*; *S2*: ... defines a new bytecode address constant *m* referring to a method that takes an argument *a* and executes the statements *S1*, *S2*, The argument *a* can be a tuple pattern similar to those used in **let** and **for** statements. Examples include *()*, *(x)*, *(x, y)*, and *(x, (y, z))*. The given local variable names are assigned upon application and are read-only. Optionally, a result variable *r* can be declared. Harmony does not support a **return** statement that *breaks out* of the code before executing the last statement.

del

The statement **del** *x* removes variable *x* from the state. *x* can be either a local or a shared variable. For example, the statement **del** *x.age* removes the *.age* field from dictionary *x*. Harmony automatically removes top-level local variables that are no longer in use from the state in order to attempt to reduce the number of states that are evaluated during model checking.

del can also be used to remove elements from a list. *x* = [*a*, *.b*, *.c*]; **del** *x*[1] results in *x* having value [*a*, *.c*].

finally

The statement **finally** *c* declares that boolean expression *c* must hold in each final state. *c* is only allowed to read shared variables and is evaluated in each final state. If it evaluates to **False**, Harmony reports an error. Harmony also reports an error if the expression evaluates to a value other than **False** or **True**.

for ... in ... [where ...]

The statement **for** *x* **in** *y*: *S1*; *S2*: ... iterates over *y* and executes for each element the statements *S1*, *S2*, *y* must be a set, list, dictionary, or string. *y* is evaluated only once at the beginning of the evaluation of this statement. In case of a set, the result is sorted (using Harmony's global order on all values). In case of a dictionary, the statement iterates over the keys in order. For

each element, the statements **S1**, **S2**, ... are executed with local variable *y* having the value of the element. *x* can be a pattern such as (*a*) or (*a*, (*b*, *c*)). If the pattern cannot be matched, Harmony detects and error.

Harmony also supports the form **for** *k:v* **in** *y*: **S1**; **S2**; This works similar, except that *k* is bound to the key and *v* is bound to the value. If *y* is not a dictionary, then *k* ranges from 0 to **len**(*y*) - 1.

The statement also supports nesting and filtering. **for** *x1* **in** *y1* **for** *x2* **in** *y2*: **S1**; **S2**; ..., is equivalent to the statement **for** *x1* **in** *y1*: **for** *x2* **in** *y2*: **S1**; **S2**; Filtering is of the form **for** *x* **in** *y* **where** *z*: **S1**; **S2**; For example, **for** *x* **in** 1 .. 10 **where** (*x* % 2) == 0: **S1**; **S2**; ... only evaluates statements **S1**, **S2**, ... for even *x*, that is, 2, 4, 6, 8, and 10.

Harmony does not support **break** or **continue** statements.

from ... import

The statement **from** *x* **import** *a*, *b*, ... imports module *x* and makes its constants *a*, *b*, ... also constants in the current module. If a module is imported more than once, its code is only included the first time. The constants will typically be the names of methods (bytecode address constants) within the module.

You can import all constants from a module *m* (including bytecode address constants) using the statement **from** *m* **import** *. This, however, excludes constants whose names start with the character *_*: those are considered *private* to the module.

global

The statement **global** *v*, ... tells the compiler that the given variables are shared global variables.

go

The statement **go** *c* *e* starts a thread with context *c* that has executed a **stop** or **save** expression. The **stop** or **save** expression returns value *e*. The same context can be started multiple times, allowing threads to *fork*. See [Figure A.1](#) for an example.

if ... [elif ...]* [else]

Harmony supports **if** statements. In its most basic form, **if** *c*: **S1**; **S2**; ... evaluates *c* and executes statements **S1**, **S2**, ... if and only if boolean expression *c* evaluated to true. Harmony checks that *c* is either **False** or **True**—if neither is the case, Harmony reports an error. The statement **if** *c*: **S1**, **S2**, ... **else**: **T1**; **T2**; ... is similar, but executes statements **T1**, **T2**, ... if and only if *c* evaluated to **False**. You can think of **elif** *c*: as shorthand for **else**: **if** *c*:

import

The statement **import** *m1*, *m2*, ... imports modules *m1*, *m2*, ... in that order. If a module is imported more than once, its code is only included the first time. The constants (including method constants) and shared variables declared in that module can subsequently be referenced by prepending “*m*.”. For example, method **f**() in imported module *m* is invoked by calling *m.f*(). If you would prefer to invoke it simply as **f**(), then you have to import using the statement **from** *m* **import** *f*.

invariant

The statement **invariant** *c* declares that boolean expression *c* is an invariant. *c* is only allowed to read shared variables and is evaluated atomically after every state change. If it ever evaluates to **False**, Harmony reports an error. Harmony also reports an error if the expression evaluates to a value other than **False** or **True**.

Invariants can be useful to specify the type of a global variable. For example, you can write **invariant** (**type**(*x*) == "int") **and** ((*x* % 2) == 0) to state that *x* is an even integer variable.

let

You can introduce new bound variables in a method using the **let** statement. The statement **let** *a* = *b*: **S1**; **S2**, ... evaluates *b*, assigns the result to read-only variable *a*, and evaluates statements **S1**, **S2**, **let** supports pattern matching, so you can write **let** *x*, (*y*, *z*) = *b*: **S1**; **S2**, This will only work if *b* is a tuple with two elements, the second of which also being a tuple with two elements—if not, Harmony will report an error.

let statements may be nested, such as **let** *a1* = *b1* **let** *a2* = *b2*: **S1**; **S2**; Doing so can improve readability by reducing indentation compared to writing them as separate statements. Compare the following two examples:

```
1  let a = y:  
2      let b = z:  
3          ...
```

```
1  let a = y  
2  let b = z:
```

pass

The **pass** statement does nothing.

print

The statement **print** *e* evaluates *e* and adds the result to the print log. The print log is used to create an “external behavior DFA” for the Harmony program.

sequential

In Harmony, shared variable **Load** and **Store** operations are atomic and have *sequential consistency*. However, Harmony does check for *data races*. A data race occurs when two or more threads simultaneously access the same shared variable, with at least one of the accesses being a **Store** operation *outside of an atomic block*. If so, Harmony will report an error. This error can be suppressed by declaring the shared variable as sequential. In particular, the statement **sequential** *x*, *y*, ... specifies that the algorithm assumes that the given variables have sequential consistency.

Note that few modern processors support sequentially consistent memory by default, as sequentially consistent memory involves high overhead.

spawn

The statement **spawn** *lv* starts a new thread that evaluates lvalue expression *lv*. The most typical form is **spawn** *f(a)*, where *f* is some method called with argument *a*. However, if *c* is a thunk, one could also call **spawn** !*c*, say.

Harmony normally checks that all threads eventually terminate. If a thread may never terminate, you should spawn it with **spawn eternal** *m a* to suppress those checks.

trap

The statement **trap** *lv* specifies that the current thread should evaluate *lv* at some future, unspecified, time. The statement models a timer interrupt or any kind of asynchronous event to be handled by the thread. Such interrupts can be disabled by setting the interrupt level of the thread to **True** using the **setintlevel** operator.

var

You can introduce new local variables in a method using the **var** statement. The statement **var** *a* = *b* evaluates *b* and assigns the result to local variable *a*. **var** supports pattern matching, so you can write **var** *x*, (*y*, *z*) = *b*. This will only work if *b* is a tuple with two elements, the second of which also being a tuple with two elements—if not, Harmony will report an error.

when

The statement **when** *c*: *S1*; *S2*; ... executes statements *S1*, *S2*, ... after waiting until *c* evaluates to **True**. **when** statements are most useful when combined with the **atomically** keyword. If *waiting* is an unused local variable, then **atomically when** *c*: *S1*; *S2*; ... is equivalent to

```
1  var waiting = True
2  while waiting:
3      atomically:
4          if c:
5              S1
6              S2
7              ...
8              waiting = False
```

Multiple **let** and **when** statements can be combined. The expressions before the colon are re-evaluated repeated until all **when** conditions are satisfied.

when exists ... in ...

The statement **when exists** *x* **in** *y*: *S1*; *S2*; ... requires that *y* evaluates to a set value. The statement does the following three things:

- it waits until y is non-empty;
- it selects one element of y non-deterministically (using a **choose** expression);
- it executes statements $S1, S2, \dots$ with the selected element bound to read-only variable x .

x may be a pattern, like in **let**, **for**, and **def** statements. Harmony reports an error if y evaluates to a value that is not a set.

when statements are most useful when combined with the **atomically** keyword. If *waiting* is an unused local variable, then **atomically when exists x in y : $S1; S2; \dots$** is equivalent to

```

1  var waiting = True:
2      while waiting:
3          atomically:
4              if y != {}:
5                  let x = choose(y):
6                      S1
7                      S2
8                      ...
9                  waiting = False

```

The statement is particularly useful in programming network protocols when having to wait for one or more messages and executing a set of actions atomically after the desired messages have arrived.

while

The statement **while c : $S1; S2; \dots$** executes statements $S1, S2, \dots$ repeatedly as long as c evaluates to **True**. Harmony does not support **break** or **continue** statements.

A.3 Harmony is not object-oriented

Python is object-oriented, but Harmony is not. For Python programmers, 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 6](#)), 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 bytecode address constants. These constants can be added

to dictionaries. For example, in [Figure 6.1](#) you can add the `Peterson_enter` and `Peterson_exit` methods to the `Peterson_mutex` dictionary like so:

```
1 { .turn: 0, .flags: [ False, False ], .enter: Peterson_enter, .exit: Peterson_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:

```
1 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 might write:

```
1 lock = Lock()
2 x = y = ?lock
```

or, using the `alloc` module,

```
1 from alloc import malloc
2 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`.

A.4 Constants, Global and Local Variables

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

Mutable method variables can be declared using the **returns** clause of a **def** statement or using **var**. Bound variables, which are immutable, can be declared in **def** statements (i.e., arguments), **let** statements, **for** loops, and **when exists** statements. Each thread has a mutable variable called **this** that contains the thread-local state. Method variables are tightly scoped and cannot be shared between threads. While in theory one method can be declared within another, they cannot share local variables either. All other variables are global and must be initialized before spawned threads start executing.

A.5 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 $a\ b\ c\ \dots$. 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 a simple string, you can also write $a.b$ for indexing.

There are five levels 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, then boolean operators, then comparison operators, and then the remaining operators.

Unary operators bind more strongly than binary operators, thus $-a[1] - a[2]$ parses as $(-(a[1])) - (a[2])$. $!a[1]$ parses as $!(a[1])$. Avoiding other ambiguities, Harmony does not allow expressions of the form $a\ @b$ where $@$ is some kind of unary operator. You have to write this as either $a[@b]$ or $a(@b)$.

Associative operators ($+$, $*$, $|$, $\&$, $^$, **and**, **or**) are interpreted as general n -ary operators, and you are allowed to write $a + b + c$. However, ambiguous expressions such as $a\ |\ b\ \&\ c$ or $a == b$ **and** c are illegal. In such cases you have to add parentheses or brackets to indicate what the intended evaluation order is, such as $(a\ |\ b)\ \&\ c$. The only exception is simple arithmetic expressions such as $a - b - c$, which is the same 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** \dots , 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.

A.6 Tuples, Lists, and Pattern Matching

Harmony tuples and lists are equivalent. 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 .

Harmony does not support special slicing syntax like Python. Use the `subseq` method in the `lists` module ([Section B.5](#)) instead.

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 a 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 .

```

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 A.2: [\[code/stacktest.hny\]](#) Testing a stack implementation.

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**, **let**, and **when exists** statements.

A.7 Dynamic Allocation

Harmony supports various options for dynamic allocation. By way of example, consider a stack. [Figure A.2](#) presents a test program for a stack. We present four different stack implementations to illustrate options for dynamic allocation:

1. [Figure A.3](#) uses a single list to represent the stack. **push** and **pop** operations update the list;
2. [Figure A.4](#) also uses a list but, instead of updating the list, it replaces the list with a new one for each operation;
3. [Figure A.5](#) 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;

```

1  def Stack() returns stack:
2      stack = []
3
4  def push(st, v):
5      (!st)[len(!st)] = v
6
7  def pop(st) returns next:
8      let n = len(!st) - 1:
9          next = (!st)[n]
10         del (!st)[n]

```

Figure A.3: [[code/stack1.hny](#)] Stack implemented using a dynamically updated list.

```

1  import lists
2
3  def Stack() returns stack:
4      stack = []
5
6  def push(st, v):
7      !st += [v,]
8
9  def pop(st) returns next:
10     let n = len(!st) - 1:
11         next = (!st)[n]
12         !st = lists.subseq(!st, 0, n)

```

Figure A.4: [[code/stack2.hny](#)] Stack implemented using static lists.

```

1  def Stack() returns stack:
2      stack = ()
3
4  def push(st, v):
5      (!st) = (v, !st)
6
7  def pop(st) returns next:
8      let (top, rest) = !st:
9          next = top
10         !st = rest

```

Figure A.5: [[code/stack3.hny](#)] Stack implemented using a recursive tuple data structure.

```

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

```

Figure A.6: [\[code/stack4.hny\]](#) Stack implemented using a linked list.

4. Figure A.6 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`.

A.8 Comments

Harmony supports the same commenting conventions as Python. In particular, anything after a `#` character on a line is ignored. You can also enclose comments on separate lines within triple quotes. In addition, Harmony supports nested multi-line comments of the form `(* comment *)`.

A.9 Type Checking

Harmony is dynamically typed. You can add type annotations to your program in the form of assertions and invariants. For example:

```
1  invariant (type(x) == "int") and ((x % 2) == 0)
2  x = choose { 0, 2, 4, 6 }
3
4  def double(n) returns result:
5      assert type(n) == "int"
6      result = n * 2
7      assert type(result) == "int"
8
9  def main():
10     x = double(x)
11
12  spawn main()
```

The invariant in Line 1 states that x is an even integer. The assertion in Line 5 states that the argument to function `double` is an integer. The assertion in Line 7 states that the return value of the function is also an integer. Harmony checks these types as it evaluates the program.

Appendix B

Modules

Harmony modules provide convenient access to various data structures, algorithms, and synchronization paradigms. They are all implemented in the Harmony language itself (so you can look at their code) although some methods have also been implemented directly into the underlying model checker for more efficient model checking.

Currently there are the following modules:

action	Section B.1	support for action-based specifications
alloc	Section B.2	dynamic memory allocation
bags	Section B.3	multi-sets
hoare	Section B.4	Hoare module interface
lists	Section B.5	common operations on lists
sets	Section B.6	common operations on sets
synch	Section B.7	synchronization
thread	Section B.8	fork/join interface to threads

B.1 The action module

The **action** module supports *action-based* specification. Such a specification consists of a explicit global state and rules for how to make state transitions. [Chapter 26](#) provides an example. The module has only one method:

explore (x)	explore the state space
------------------------	-------------------------

Here x is a set of lambdas, each of which can return a set of *thunks*, each representing a possible action (state change). The union of the results of the lambdas should generate all possible actions. A thunk represents a method and its arguments that updates the state accordingly.

B.2 The alloc module

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

<code>malloc(<i>v</i>)</code>	return a pointer to a memory location initialized to <i>v</i>
<code>free(<i>p</i>)</code>	free an allocated memory location <i>p</i>

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.

B.3 The bags module

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

<code>empty()</code>	returns an empty bag
<code>fromSet(<i>s</i>)</code>	create a bag from set <i>s</i>
<code>fromList(<i>t</i>)</code>	convert list <i>t</i> into a bag
<code>multiplicity(<i>b</i>, <i>e</i>)</code>	count how many times <i>e</i> occurs in bag <i>b</i>
<code>bchoose(<i>b</i>)</code>	like <code>choose(<i>s</i>)</code> , but applied to a bag
<code>add(<i>b</i>, <i>e</i>)</code>	add one copy of <i>e</i> to bag <i>b</i>
<code>remove(<i>b</i>, <i>e</i>)</code>	remove one copy of <i>e</i> from bag <i>b</i>
<code>combinations(<i>b</i>, <i>k</i>)</code>	return set of all <i>subbags</i> of size <i>k</i>

B.4 The hoare module

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

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

B.5 The lists module

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

subseq (t, b, f)	return a <i>slice</i> of list t starting at index b and ending just before f
append (t, e)	returns $t + [e]$
head (t)	return the first element of list t
tail (t)	return all but the first element of list t
index (t, e)	return the index of element e in list t
startswith (t, s)	returns whether s is a prefix of t
filter (f, t)	returns a list of elements of t satisfying function f
map (f, t)	returns a list of elements of t mapped by function f
permuted (t)	returns a permutation of set t
list (t)	convert a 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 <code>enumerate</code>
qsort (t)	returns a copy of t sorted using quicksort
foldl (t, f, z)	left fold with f a binary method and z the initial value
foldr (t, f, z)	right fold with f a binary method and z the initial value
reduce (f, t, z)	same as foldl (t, f, z)

B.6 The sets module

The **sets** module implements the following methods:

add (s, e)	returns $s \cup \{e\}$
remove (s, e)	returns $s \setminus \{e\}$
subsets (s)	returns the set of subsets of s
union (s)	returns the union of the elements of s
filter (f, s)	returns a set of elements of s satisfying function f
map (f, s)	returns a set of elements of s mapped by function f
cartesian (d)	d is a list of sets. Returns the Cartesian product.
combinations (s, k)	returns set of all subsets of size k
reduce (f, t, z)	same as Python's functools reduce ()

B.7 The synch module

The **synch** module provides the following methods:

<code>atomic_load(<i>p</i>)</code>	atomically evaluate <code>!p</code>
<code>atomic_store(<i>p</i>, <i>v</i>)</code>	atomically assign <code>!p = v</code>
<code>tas(<i>lk</i>)</code>	test-and-set on <code>!lk</code>
<code>cas(<i>ptr</i>, <i>old</i>, <i>new</i>)</code>	compare-and-swap on <code>!ptr</code>
<code>BinSema(<i>v</i>)</code>	return a binary semaphore initialized to <i>v</i>
<code>Lock()</code>	return a binary semaphore initialized to False
<code>acquire(<i>bs</i>)</code>	acquire binary semaphore <code>!bs</code>
<code>release(<i>bs</i>)</code>	release binary semaphore <code>!bs</code>
<code>Condition()</code>	return a condition variable
<code>wait(<i>c</i>, <i>lk</i>)</code>	wait on condition variable <code>!c</code> and lock <i>lk</i>
<code>notify(<i>c</i>)</code>	notify a thread waiting on condition variable <code>!c</code>
<code>notify_all(<i>c</i>)</code>	notify all threads waiting on condition variable <code>!c</code>
<code>Semaphore(<i>cnt</i>)</code>	return a counting semaphore initialized to <i>cnt</i>
<code>P(<i>sema</i>)</code>	procure <code>!sema</code>
<code>V(<i>sema</i>)</code>	vacate <code>!sema</code>
<code>Queue()</code>	return a synchronized queue object
<code>get(<i>q</i>)</code>	return next element of <i>q</i> , blocking if empty
<code>put(<i>q</i>, <i>item</i>)</code>	add <i>item</i> to <i>a</i>

B.8 The thread module

The `thread` module implements the fork/join interface to threads.

<code>fork(<i>thunk</i>)</code>	spawn <i>thunk</i> and return a thread handle
<code>join(<i>handle</i>)</code>	wait for the thread to finish and return its result

For example, the following code doubles each element of *data* in parallel and then sums the result when done.

```

1  from thread import *
2
3  data = { 1, 2, 4 }
4
5  def main():
6      let double = lambda x: 2*x end
7      let map = { fork(?double(k)) for k in data }:
8          print sum(join(t) for t in map)
9
10 spawn main()
```

Appendix C

The Harmony Virtual Machine

The Harmony Virtual Machine (HVM, [Chapter 4](#)) has the following state:

code	a list of HVM machine instructions
variables	a dictionary mapping strings 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 the program counter set to 0. The thread 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 strides, 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.

C.1 Machine Instructions

Apply m	call method m
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	retrieve an element from a iterable type
Del $[v]$	delete shared variable v
DelVar $[v]$	delete thread variable v
Dup	duplicate the top element of the stack
Finally bca	bca is the bytecode address of a lambda that returns a boolean
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
Invariant bca	bca is the bytecode address of a lambda that takes arguments pre , $post$ and returns a boolean
Jump p	set program counter to p
JumpCond $e\ p$	pop expression and, if equal to e , set program counter to p
Load $[v]$	evaluate the address on the stack (or load shared variable v)
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
Print	pop a value and add to the print history
Push c	push constant c onto the stack
ReadonlyInc/Dec	increment/decrement the read-only counter of this context
Return $[v\ [, d]]$	pop return address, push v (or default value d), 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 [eternal]	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:

- Even though Harmony code does not allow taking addresses of thread variables, both shared and thread variables can have addresses.
- The **Load**, **Del**, **DelVar**, and **Stop** instructions have an optional variable name: If omitted the top of the stack must contain the address of the variable.
- The **Store** instruction has an optional variable name. The **StoreVar** instruction can even have a nested tuple of variable names such as $(a, (b, c))$. 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 **Frame** instruction pushes the value of the thread register (*i.e.*, the values of the thread variables) onto the stack. 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.

- The **Apply** instruction is unnecessary as it can be implemented using **2-ary Closure** and **Load**. However, method calls are frequent enough to warrant a faster mechanism, reducing model checking time.
- The **Return** instruction has an optional result variable and default value. If neither is specified, the result value is on top of the stack. Otherwise it tries to read the local variable. If the variable does not exist, the default value is used or an error is thrown.
- Every **Stop** instruction must immediately be followed by a **Continue** instruction.
- There are two versions of **AtomicInc**: *lazy* or *eager*. When *eager*, an atomic section immediately causes a *switch point* (switch between threads). When *lazy*, the state change does not happen until the first **Load**, **Store**, or **Print** instruction. If there are no such instructions, the atomic section may not even cause a switch point.

The n -**Ary** instruction can have many different operators as argument. [Section A.1](#) describes many of these operators, but some are used internally only. The current set of such operators are as follows:

AddArg	pop an argument and an address and push an address with the argument added
Closure	pop an argument and a function and push an address with the single argument
DictAdd	pop a value, a key, and a dictionary, and push an updated dictionary
ListAdd	pop a value and a list, and push a new list with the given value added to the end
SetAdd	pop a value and a set, and push a new set with the given value added

C.2 Addresses and Method Calls

Syntactically, Harmony does not make a distinction between methods calls and indexing in Harmony dictionaries, lists, and strings. This is because Harmony makes all four look like functions that map a value to another value. Beuses dynamic types, an expression like $a\ b$ could mean that variable a contains a bytecode address and a method call must be made with b as argument, or index b must be looked up in the a value. Things can get more complicated for an expression like $a\ b\ c$, which means $((a\ b)\ c)$: $a\ b$ could return a bytecode address or an indexable Harmony value.

To deal with this, Harmony has a fairly unique address type. An address consists of a function and a list of arguments, which we will denote here as $\langle f, [a_0, a_1, \dots] \rangle$. If a is a shared variable, then the address of $a\ b\ c$ is $\langle \$, [“a”, b, c] \rangle$, where $\$$ is the function that maps the names of shared variables to their values. In particular, $\$(“a”)$ is the value of variable a . A function can also be a bytecode address or an indexable Harmony value. So, if a is instead a method (i.e., a bytecode address constant), then the address would be $\langle a, [b, c] \rangle$. In the Harmony Virtual Machine, the $\$$ function is represented as the bytecode address -1 .

To evaluate the Harmony expression $a\ b\ c$, Harmony first generates its address (evaluating the expression left to right). If a is a variable name, then the function in the address depends on whether it is a shared variable or a thread variable. After the address is computed and pushed onto the stack, the **Load** instruction evaluates the address, possibly in multiple steps in an iterative manner.

A basic step of evaluating $\langle function, arguments \rangle$ proceeds as follows:

1. If *arguments* is empty, replace the address by *function* and proceed to the next instruction.
2. If *function* is an indexable Harmony value (list, string, or dictionary), *arg* is the first argument, and *remainder* are the remaining arguments, then replace the address by $\langle function[arg], remainder \rangle$ and repeat.
3. If *function* is $\$$, then replace the address by $\langle \$(arg), remainder \rangle$ and repeat.
4. If *function* is a bytecode address, then push *remainder*, the value of the current program counter (still pointing to the **Load** instruction), and *arg* onto the stack and set the program counter to *function*. The **Return** instruction pushes $\langle r, remainder \rangle$, where r is the result of the function, and restores the program counter so it executes the **Load** instruction again.

The Harmony Virtual Machine can sometimes to multiple of these basic steps in one big step. For example, if $a\ b\ c$ is a memory address, the **Load** instruction will finish in a single atomic step. Both **Load** and **Return** are optimized in such ways.

C.3 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:

program counter	contain an integer value pointing into the bytecode
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 strings (names of method variables) to values
thread-local variables	a dictionary mapping strings (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:

- 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 strings to arbitrary values. The strings correspond to the variable names in a Harmony program.

C.4 Formal Specification

Most of the Harmony Virtual Machine is specified in TLA+. Given a Harmony program, you can output the TLA+ specification for the program using the following command:

```
$ harmony -o program.tla program.hny
```

For most Harmony programs, including Peterson’s algorithm and the Dining Philosophers in this book, the result is complete enough to run through the TLC model checker.

Appendix D

How Harmony Works

This appendix gives a very brief overview of how Harmony works. In a nutshell, Harmony goes through the following four phases:

1. The Harmony *compiler* turns your Harmony program into bytecode. A recursive descent parser and code generator written in Python (see `harmony.py`) turns an `x.hny` program into `x.hvm`, a JSON file containing the corresponding bytecode.
2. The Harmony *model checker* evaluates the state space that the program (now in bytecode) can generate. The model checker is written in C as it needs to be highly efficient (see `charm.c`). The model checker starts from the initial state, and then, iteratively, checks for each state that it has found what next steps are possible and generates the next states using the Harmony virtual machine ([Appendix C](#)). If the model is finite, eventually the model checker will generate a graph with all possible states, known as a *Kripke structure*. If there is a problematic path in the Kripke structure (see below), then it will report the shortest such path in the `x.hco` output file in JSON format.
3. The Kripke structure is interpreted as a nondeterministic epsilon automaton. Each step that prints a value is labeled with that value, while each step that doesn't is labeled with ϵ . In the next phase, the ϵ -NFA is converted into a minimized DFA. This DFA is written to the `x.hfa` output file.
4. The `x.hco` output file is translated twice by `harmony.py`. There is a so-called *brief output* that is written to standard output. The rest depends on whether there was a problem with the execution or not. If there was a problem, the more comprehensive output is placed in the `x.htm` HTML output file, allowing you to navigate the problematic path and all the details of each of the states on the path.

D.1 Compiler

The Harmony compiler, in order to stay true to the Harmony source program, does not do much in the way of optimizations. The main optimizations that it does are:

- Constant folding: (simple) expressions consisting only of constants are evaluated by the compiler rather than by the model checker;
- Jump threading: Harmony eliminates jump to jump instructions;
- Dead variable elimination: Harmony removes method variables that are no longer in use from the state in order to reduce the state space to be explored.

D.2 Model Checker

The Harmony model checker, called *Charm*, takes the output from the compiler and explores the entire state space in breadth-first order. Even though Harmony does not really support input, there are three sources of non-determinism that make this exploration non-trivial:

- *choose expressions*: Harmony’s ability to let the program choose a value from a set;
- *thread interleaving*: different threads run pseudo-concurrently with their instructions interleaved in arbitrary ways;
- *interrupts*: Harmony programs can set interrupts that can go off at arbitrary times.

A thread can be in *atomic* mode or not. In atomic mode, the execution of the thread is not interleaved with other threads. A thread can also be in *read-only* mode or not. In read-only mode, the thread cannot write or delete shared variables.

Charm has some tricks to significantly reduce the state space to explore.

- A thread can have local state (program counter, stack, method variables, and thread-local state variables). That state is called the *context* of the thread. The context of a thread cannot be accessed by other threads, nor by **invariant** or **finally** statements. So, the model checker only interleaves threads at **Load**, **Store**, and **Del** instructions where a thread interacts with global variables.
- Threads are anonymous, and therefore two or more threads can have the same context. The state of the model checker therefore maintains a *bag* (multiset) of contexts rather than a *set* of contexts. Thus even if there are hundreds of threads, there may be only tens of possible context states.

That said, *state space explosion* is still a possibility, and Harmony programmers should keep this in mind when writing and testing their programs. Do not be too ambitious: Start with small tests and gradually build them up as necessary.

The model checker stops either when it finds a failing execution or when it has explored the entire state space, whichever comes first. An execution can fail for a variety of reasons:

- An invariant failing: Harmony evaluates all invariants in all states that it finds—if one fails, Harmony stops further exploration;
- An assertion failing;
- A behavior violation: This is when the sequence of printed values are not recognized by the provided DFA (using the **-B** flag);

- A *silly* error: This includes reading variables that have not been assigned, trying to add a set to an integer, taking the length of something that is not a set or a dictionary, and so on;
- An infinite loop: A thread goes into an infinite loop without accessing shared variables.

D.3 Automata Conversion

The output of the model checker is a graph (a so-called *Kripke structure*) that is typically very large. If there are no failed transitions, Harmony generates a DFA of the print behavior. Starting with the original Kripke structure, the edges are inspected. This graph of nodes (states) and edges (transitions) forms a Non-deterministic Finite Automaton (NFA) with ϵ -transitions (transitions without print operations). Harmony turns the NFA into a DFA and minimizes the DFA (although not strictly necessary). The DFA can be fed into another run of the model checker to check that its print operations are consistent with the provided DFA.

D.4 Model Checker Output Analysis

If some execution failed, then Harmony will simply report the path of that failing execution. But otherwise there may be the following outcomes:

- No issues: No failing executions and each program can terminate;
- Non-terminating states: Some executions lead to some form of deadlock or other issue that causes some (non-eternal) threads not to be able to terminate;
- Race conditions: There are executions in which two threads access the same shared state variable, with at least one of those accesses being a **Store** operation;
- Busy waiting: Executions in which threads are actively waiting for some condition, usually by releasing and reacquiring locks.

In order to diagnose these outcomes, Harmony must analyze the graph.

The first thing that Harmony does is to locate non-terminating states, if any. To do this, Harmony first determines the *strongly connected components* of the graph using Tarjan's algorithm. A component (subgraph) of a graph is strongly connected if each vertex (state) in the component can be reached from each other vertex. The components then form a Directed Acyclic Graph (DAG). The DAG is easier to analyze than the original graph. One can easily determine the sink components (the components with no outgoing edges). If such a component has non-eternal threads in it, then each state in that component is a non-terminating state.

To find race conditions, the model checker looks in the graph for states in which there are multiple threads that can make a step. If there is a step in which multiple threads access the same shared variable, at least one of those accesses is a store operation, and at least one of those threads is not in atomic mode, then Harmony reports the shortest path to such a state.

To show how Harmony detects busy waiting, we will first show how Harmony determines if a thread is blocked or not. A thread is considered blocked if it cannot terminate without the help of another thread. For example, a thread waiting for a lock is blocked and cannot terminate until another thread releases the lock. Determining whether a thread is blocked in a particular state can

be done within the confines of the connected component: The analyzer tries all possible executions of the thread. If it cannot “escape” the connected component by doing so, it is considered blocked. A thread is considered *busy waiting* if it is blocked, but it is also changing the shared state while doing so. A thread that is waiting on a spinlock only observes the state.

In the output, each thread has a unique identifier: $T0$ is the initialization thread; Tn is the n^{th} spawned thread that executes. This seems to contradict the fact that Harmony threads are anonymous. The output analyzer assigns these identifiers *a posteriori* to the threads in the state graph by keeping track, along the reported execution path, what state each thread is in. So, by examining the initial context of the thread that is running from some particular state, it can determine if that context corresponds to the current context of some thread that ran previously or if the context belongs to a new thread that has not run before.

Appendix E

Simplified Grammar

The next pages show a compact version of the complete Harmony grammar. The simplified grammar ignores indentation rules.

```

1  block: statement [[NEWLINE | ';'] statement]*;
2
3  statement
4      : e      # usually a function call
5      | e '=' [e '=']* e      # assignment
6      | e aug_assign e      # augmented assignment
7      | assert e [',' e]
8      | atomically statement
9      | atomically ':' block
10     | await e
11     | const bv '=' e
12     | def bv [returns id]? ':' block
13     | del e [',' e]*
14     | finally e
15     | from id import id [',' id]*
16     | global id [',' id]*
17     | go e e
18     | if e ':' block [elif e ':' block]* [else ':' block]?
19     | import id [',' id]*
20     | invariant e
21     | pass
22     | print e
23     | sequential id [',' id]*
24     | spawn e
25     | trap e
26     | var bv '=' e
27     | while e ':' block
28     | letwhen ':' block      # let/when statement
29     | comprehension ':' block      # for statement
30     ;
31
32 comprehension: for_clause [for_clause | where_clause]*;
33 letwhen: [let_clause | when_clause]+;
34 for_clause: for bv in e;
35 where_clause: where e;
36 let_clause: let bv '=' e;
37 when_clause: when e | when exists bv in e;
38
39 aug_assign
40     : '+' | '-' | '*' | '**=' | '/' | '//=' | '%=' | 'mod='
41     | '>>=' | '<<=' | 'and=' | 'or=' | '=>=' | '&=' | '|=' | '^='
42     ;

```

```

1  e # expression
2  : False | True | None | '{:}'
3  | [0-9]+ | 0x[0-9a-fA-F]+ | 0b[0-1]+ | 0o[0-7]+ # integer
4  | "... " | '...' | """...""" | ' ' '...' ' ' | '!' id # string forms
5  | id
6  | unary e
7  | e binary e
8  | e e # application
9  | [e,]* e? # tuple/list
10 | '{' [e,]* e? '}' # set
11 | '{' [e ':' e,]* [e ':' e] '}' # dictionary
12 | '{' e ':' e '}' # range
13 | e comprehension # list comprehension
14 | '{' e comprehension '}' # set comprehension
15 | '{' e ':' e comprehension '}' # dict comprehension
16 | '(' e? ')'
17 | '[' e? ']'
18 | e if e else e
19 | lambda bv: e end
20 | save e
21 | stop id
22 ;
23
24 unary
25 : '!' | '?' | '!' | '~' | abs | all | any | bin | choose | dict
26 | hex | int | keys | len | list | max | min | not | oct
27 | reversed | set | sorted | str | sum | type | zip
28 ;
29
30 binary
31 : '+' | '-' | '*' | '/' | '/' | '%' | mod | '**' | '|' | '&' | '^'
32 | '<<' | '>>' | '==' | '!=' | '<' | '<=' | '>' | '>=' | not? | in
33 | or | and | '=>'
34 ;
35
36 bv # bounded variable(s)
37 : id
38 | [bv ','] + bv
39 | '(' bv ')'
40 | '[' bv ']'
41 ;
42
43 id: [_a-zA-Z][_a-zA-Z0-9]*; # identifier

```

Appendix F

Directly checking linearizability

We want a concurrent queue to behave consistently with a sequential queue in that all **put** and **get** operations should 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 9.3](#) 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 9.3](#) 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 F.1](#) is a copy of [Figure 9.3](#) 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 F.2](#) shows how linearizability can be tested. The test program is similar to the sequential test program ([Figure 10.1](#)) but starts a thread for each operation. The operations are executed concurrently on the concurrent queue implementation of [Figure F.1](#), but they are executed sequentially on the sequential queue specification of [Figure 9.1\(a\)](#). 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


```

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(), .time: 0 }
6
7  def _linpoint(q):
8      atomically:
9          this.qtime = q→time
10         q→time += 1
11
12  def put(q, v):
13      let node = malloc({ .value: v, .next: None }):
14          acquire(?q→lock)
15          if q→tail == None:
16              q→tail = q→head = node
17          else:
18              q→tail→next = node
19              q→tail = node
20          _linpoint(q)
21          release(?q→lock)
22
23  def get(q):
24      acquire(?q→lock)
25      let node = q→head:
26          if node == None:
27              result = None
28          else:
29              result = node→value
30              q→head = node→next
31              if q→head == None:
32                  q→tail = None
33              free(node)
34      _linpoint(q)
35      release(?q→lock)

```

Figure F.1: [\[code/queue.lin.hny\]](#) Queue implementation with linearization points

```

1  import queueelin, queuespec
2
3  const NOPS = 4
4  const VALUES = { 1..NOPS }
5
6  sequential qtime
7  qtime = 0
8
9  implq = queueelin.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         queueelin.put(implq, v)
17         await qtime == this.qtime
18         queuespec.put(specq, v)
19     else:
20       let v = queueelin.get(implq):
21         await qtime == this.qtime
22       let w = queuespec.get(specq):
23         assert v == w
24     atomically qtime += 1
25
26 for i in {1..NOPS}:
27   spawn thread()

```

Figure F.2: [\[code/qtestconc.hny\]](#) Concurrent queue test

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.

Appendix G

Manual Pages

NAME

Harmony — the Harmony compiler and model checker

SYNOPSIS

```
harmony [options] filename
```

DESCRIPTION

harmony is a compiler and model checker for the Harmony programming language. **harmony** compiles Harmony into bytecode and then model checks the bytecode. The result is analyzed for failing assertions and invariants, non-terminating conditions such as deadlock and infinite loops, race conditions, deviations from specifications, and busy waiting. There are three phases:

- *compile*: parses Harmony source code and generates Harmony virtual machine code;
- *model check*: generates a graph of all reachable states from the Harmony virtual machine code while checking for safety violations;
- *analysis*: checks the graph for non-termination, race conditions, and busy waiting.

The Harmony file name extensions are as follows:

- **.hny**: Harmony source code;
- **.hvm**: Harmony virtual machine code (in JSON format);
- **.hco**: Harmony output (in JSON format);
- **.hvb**: Harmony verbose output (human readable);
- **.hfa**: Harmony finite automaton, describing the possible **print** outputs (in JSON format).

In addition, **harmony** can also generate **.tla** (TLA+), **.htm** (HTML), **.gv**: (Graphviz DOT version of **.hfa** output), **.png**: (PNG version of **.hfa** output), and **.tex**: (LaTeX formatted source code).

By default, running “**harmony x.hny**” will generate **x.hvm**, **x.hco**, **x.hvb**, **x.png**, and **x.hvm** files. Harmony will also, by default, automatically start a web browser to display the **x.hvm** file. Various options can be used to change the behavior.

When importing a module using **import x**, **harmony** will try to find the corresponding **.hny** file in the following order:

1. check if the module file is specified with the **-m** or **--module** option;
2. see if a file by the name *x.hny* is present in the same directory as the source file;
3. see if a file by the name *x.hny* is present in the installation’s **modules** directory.

OPTIONS

Output file options:

- **-o filename.gv**: specify the name of the file where the **graphviz** (DOT) output should be stored;
- **-o filename.hco**: specify the name of the file where model checker output should be stored;
- **-o filename.hfa**: specify the name of the file where the Harmony finite automaton should be stored;
- **-o filename.htm**: specify the name of the file where the HTML output should be stored;
- **-o filename.hvb**: specify the name of the file where the verbose output should be stored;
- **-o filename.hvm**: specify the name of the file where the Harmony virtual machine code should be stored;
- **-o filename.png**: specify the name of the file where the PNG output should be stored;
- **-o filename.tla**: generate a TLA+ file specifying the behaviors of the Harmony virtual machine code;
- **-o filename.tex**: generate a LaTeX+ file containing the formatted source code.

Other options:

- **-a**: compile only and list machine code (with labels);
- **-A**: compile only and list machine code (without labels);
- **-B filename.hfa**: check Harmony code against output behaviors described in **filename.hfa** (result of another Harmony run);

- `-c, --const constant=expression`: set the value of the given constant (which must be defined in the code) to the result of evaluating the given expression;
- `-m, --module module=filename.hny`: load the given module instead of looking in default locations;
- `--noweb`: do not start a web browser upon completion;
- `-v, --version`: print the **harmony** version number.
- `-w #workers`: specify the number of concurrent threads the model checker uses.

Appendix H

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]. Figure 5.6 presents the algorithm. 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 trivial. You have to understand a little bit 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 (`--init--()`) and its program counter is 0. A *trace* is a sequence of steps starting from the initial state, resulting in a sequence of states. 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.

It is often useful to classify states. *Initial*, *final*, and *reachable*, and *unreachable* are all examples of classes of states. Figure H.1 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.

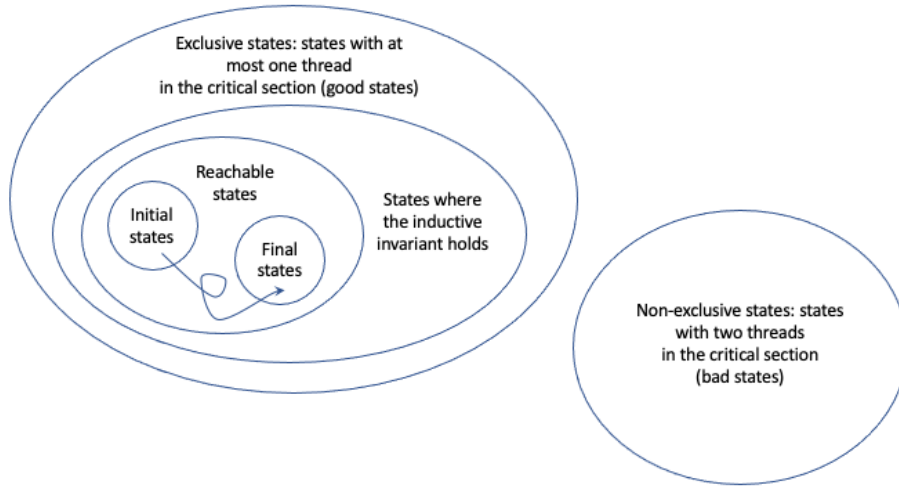


Figure H.1: Venn diagram classifying all states and a trace

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: $\mathcal{F}(i) = \text{thread}(i)@[10 \dots 17] \Rightarrow \text{flags}[i]$, that is, if thread i is executing in lines 10 through 17, then $\text{flags}[i]$ 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 $flags[i] = \text{true}$
2. states in which $\neg \text{thread}(i)@[10 \dots 17]$ (because false implies anything)

In each case, we need to show that if thread i takes a step, then $\mathcal{F}(i)$ 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 from line 17 to line 18. But executing the line would also take the thread out of lines $10 \dots 17$, so $\mathcal{F}(i)$ continues to hold. In the second case (thread i is not executing in lines $10 \dots 17$), the only step that would cause thread i to execute in lines $10 \dots 17$ would be the step in line 9. 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: $\text{thread}(i)@cs \Rightarrow flags[i]$: 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 will use a similar technique to prove the exclusivity is invariant.

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: $\text{thread}(i)@cs \Rightarrow \mathcal{C}(i)$. However, because thread $1 - i$ is executing concurrently, this property does not hold. You can use Harmony to verify this. Just place the following command in the critical section of the program:

assert (**not** $flags[1 - self]$) **or** ($turn == self$)

When running Harmony, this assertion will fail. You can check the HTML output to see what happened. 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. Then $\mathcal{C}(0)$ is violated. But it suggests a new property: $\mathcal{G}(i) = \text{thread}(i)@cs \Rightarrow \mathcal{C}(i) \vee \text{thread}(1 - i)@12$. 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 12.

$\mathcal{G}(i)$ 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 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 about to execute Line 12, we know that both $\mathcal{C}(0)$ and $\mathcal{C}(1)$ 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 is 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. $flags[1] = \mathbf{False}$;
2. $turn = 0$;
3. thread 1 is about to execute Line 12.

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 about to execute Line 12 (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 about to execute Line 12 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 $flags[0] = \mathbf{True}$ 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 about to execute Line 12. We already saw that this is not an invariant. (The author fixed the Wikipedia page with the help of Fred B. Schneider.)

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. Moreover, analyzing the counterexample provided by Harmony may well suggest how to fix the predicate.

Exercises

H.1 Can you find one or more inductive invariants for the algorithm in [Figure 5.7](#) 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:
```

```
  X0: flagX = True  
  X1: if not flagY goto X4  
  X2: flagX = False  
  X3: goto X0  
  X4: ...critical section...  
  X5: flagX = False
```

```
thread Y:
```

```
  Y0: flagY = True  
  Y1: if not flagX goto Y4  
  Y2: flagY = False  
  Y3: goto Y0  
  Y4: ...critical section...  
  Y5: flagY = False
```

Appendix I

Split Binary Semaphores

The Split Binary Semaphore (SBS) approach is a general technique for implementing conditional waiting. SBS was originally proposed by Tony Hoare and popularized by Edsger Dijkstra [Dij79]. A *binary semaphore* is a generalization of a lock. While a lock is always initialized in the released state, a binary semaphore—if so desired—can be initialized in the acquired state. SBS is an extension of a critical section that is protected by a lock. 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 and therefore uses just one binary semaphore (because $n = 0$). But, 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 binary 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, then it must select one and open that gate. If there is no such gate, then it must open the main gate.

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

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, then 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() returns lock:
4      lock = {
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 > 0 or 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 I.1: [\[code/rwlock.sbs.hny\]](#) 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 I.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 (0 or 1);
- *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 the thread 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 works 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 *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 binary semaphore associated with the waiting gate. This causes the thread to block until some other 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. The writer then calls `_release_one()`. `_release_one()` finds that there are no writers in the critical section and there are two readers waiting. The writer 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()`.

```

1  import synch
2
3  def Monitor() returns monitor:
4      monitor = synch.Lock()
5
6  def enter(mon):
7      synch.acquire(mon)
8
9  def exit(mon):
10     synch.release(mon)
11
12 def Condition() returns condition:
13     condition = { .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 I.2: [\[modules/hoare.hny\]](#) Implementation of Hoare monitors

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 reader threads that are holding the reader/writer lock.

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`.

```

1  import hoare
2
3  def BoundedBuffer(size) returns buffer:
4      buffer = {
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 Queue() returns empty:
13      empty = BoundedBuffer(4)
14
15  def put(bb, item):
16      hoare.enter(?bb→mon)
17      if bb→count == bb→size:
18          hoare.wait(?bb→prod, ?bb→mon)
19      bb→buf[bb→tail] = item
20      bb→tail = (bb→tail % bb→size) + 1
21      bb→count += 1
22      hoare.signal(?bb→cons, ?bb→mon)
23      hoare.exit(?bb→mon)
24
25  def get(bb) returns next:
26      hoare.enter(?bb→mon)
27      if bb→count == 0:
28          hoare.wait(?bb→cons, ?bb→mon)
29      next = bb→buf[bb→head]
30      bb→head = (bb→head % bb→size) + 1
31      bb→count -= 1
32      hoare.signal(?bb→prod, ?bb→mon)
33      hoare.exit(?bb→mon)

```

Figure I.3: [\[code/boundedbuffer_hoare.hny\]](#) Bounded Buffer implemented using a Hoare monitor

Harmony does not have language support for monitors, but it has a module called **hoare**. [Figure I.2](#) shows its implementation. A Hoare 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. Method **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 SBS. Method **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 I.3](#) 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 I.2](#)). 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.

Exercises

I.1 Several of the calls to **_release_one()** in [Figure I.1](#) can be replaced by simply releasing *mutex*. Which ones?

I.2 Implement a solution to the producer/consumer problem using split binary semaphores.

I.3 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.)

I.4 Write a test program for your bound lock that checks that no more than *M* threads can acquire the bound lock.

I.5 Write a test program for bound locks that checks that up to *M* threads can acquire the bound lock at the same time.

I.6 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

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

in [Figure I.1](#). (Similarly, you may want to try to implement an *upgrade* of a read lock to a write lock. Why is this problematic?)

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 initial design of Harmony. Haobin even contributed some code to the Harmony compiler. Many thanks are also due to William Ma who refactored the Harmony code to make it easier to maintain. He also wrote the first version of the behavior automaton generator and created the first graphs using the graphviz tool. I have had lots of discussions with him about a wide range of improvements to the Harmony language, many of which came to fruition. I also want to thank Ariel Kellison with whom I discussed approaches to formally specify the Harmony virtual machine in TLA+.

Kevin Sun and Anthony Yang built a beautiful VSCode extension for Harmony called Harmony-Lang and proceeded to build an animator for Harmony executions and two cloud-based Harmony offerings, which you can learn about at <http://harmony.cs.cornell.edu>. They also developed much of that web site and made valuable suggestions for improvements to the Harmony language. Later they were joined by Shi Chong Zhao, Robin Li, and Seth Norman, all of whom made significant contributions to Harmony.

I also would like to acknowledge my regular conversation about Harmony with Sasha Sandler of the Oracle Cloud Infrastructure group. He is an early industrial adopter of Harmony and has used it successfully to find and fix bugs in industrial settings. His insights have been invaluable.

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. Being a foremost security expert, he also assisted significantly with the chapter on the Needham-Schroeder protocol.

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 have learned so much from his papers.

Tim Teitelbaum provided numerous suggestions for improving the text and formatting of the book.

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, CJ Lee, Harshul Sahni, Hartek Sabharwal, Heather Zheng, Jack Rehmann, Jacob Brugh, Liam Arzola, Lorenzo Alvisi, Maria Martucci, Melissa Reifman, Nalu Concepcion, Phillip O'Reggio, Saleh Hassen, Sunwook Kim, Terryn Jung, Trishita Tiwari, Xiangyu Zhang, Yidan Wang, Zach Garcia, 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, at home, during the turbulent months of May and June 2020.

Index

- acknowledgment, 142
- acquire, 58
- action, 160
- actor model, 107
- address, 28, 43
- alloc module, 208
- alternating bit protocol, 142
- atomic instruction, 51
- atomicity, 9

- bag, 28
- bags module, 208
- barrier synchronization, 110
- big lock, 66
- blocked thread, 57
- blocking queue, 107
- Bohrbug, 9
- bounded buffer, 87
- broadcast, 91
- busy waiting, 84
- bytecode, 26

- choose operator, 13
- client/server model, 90
- coarse-grained lock, 66
- constant, 18
- context, 27
- continuation, 27
- corner case, 10
- critical region, 33
- critical section, 33

- data race, 51
- deadlock, 101
- deadlock avoidance, 104
- determinism, 9, 154

- dictionary, 26
- dining philosopher, 101
- directory, 28
- distributed system, 142
- dynamic allocation, 63

- exception, 132

- failure, 142
- fairness, 97
- fine-grained lock, 66
- flow control, 90
- formal verification, 10

- go statement, 58

- hand-over-hand locking, 66
- Harmony method, 43
- Harmony Virtual Machine, 26
- Heisenbug, 9
- hoare module, 208
- HVM, 26

- import statement, 45
- inconsistent, 51
- inductive invariant, 231
- interleaving, 22
- interrupt, 132
- interrupt-safety, 132
- invariant, 10, 230

- Kripke structure, 218

- lists module, 208
- liveness property, 34
- lock, 35, 51
- lock granularity, 66

logical timestamp, [165](#), [223](#)

machine instruction, [22](#)

Mesa, [91](#)

message passing, [107](#)

model checking, [9](#)

module, [45](#)

monitor, [238](#)

multiset, [28](#)

mutual exclusion, [34](#)

network, [142](#)

non-blocking synchronization, [139](#)

non-determinism, [230](#)

notify, [91](#)

notify_all, [91](#)

pattern matching, [202](#)

Peterson's Algorithm, [39](#), [230](#)

pipeline, [87](#)

pointer, [43](#)

producer/consumer problem, [87](#)

program counter, [27](#)

progress, [34](#)

property, [97](#)

protocol, [142](#)

race condition, [22](#)

reachable state, [230](#)

reader/writer lock, [84](#)

register, [27](#)

release, [58](#)

replication, [154](#)

reserve debugging, [82](#)

retransmission, [142](#)

safety property, [34](#)

seqlock, [141](#)

sequence number, [142](#)

sequential, [9](#)

sequential consistency, [28](#)

sets module, [209](#)

shared variable, [9](#)

signal, [238](#)

single point of failure, [164](#)

spinlock, [51](#), [53](#)

split binary semaphore, [235](#)

stack machine, [28](#)

starvation, [54](#), [97](#)

state, [230](#)

state machine replication, [154](#)

step, [230](#)

stop expression, [58](#)

stride, [211](#)

synch module, [51](#), [209](#)

synchronized queue, [107](#)

TAS, [53](#)

test, [9](#)

test-and-set, [53](#)

thread, [9](#), [19](#), [34](#)

thread module, [210](#)

thread safety, [34](#)

thread variable, [230](#)

thread-local, [27](#)

thunk, [160](#)

Time Of Check Time Of Execution, [59](#)

TOCTOE, [59](#)

trace, [230](#)

virtual machine, [26](#)

wait, [91](#), [238](#)

wait-free synchronization, [141](#)

Glossary

actor model is a concurrency model where there are no shared variables, only threads with private variables that communicate through message passing. 107

atomic instruction a machine instruction that may involve multiple memory load and/or store operations and is executed atomically. 51

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. 9

barrier synchronization is when a set of threads execute in rounds, waiting for one another to complete each round. 110

behavior is a sequence of states. A trace uniquely defines a behavior but not vice versa. 70

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. 57

busy waiting (a.k.a. spin-waiting) is when a thread waits in a loop for some application-defined condition instead of blocking. 84

concurrent execution (a.k.a. parallel execution) is when there are multiple threads executing and their machine instructions are interleaved in an unpredictable manner. 9

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. 91

conditional critical section is a critical section with, besides mutual exclusion, additional conditions on when a thread is allowed to enter the critical section. 235

context (a.k.a. continuation) describes the state of a running thread, including the value of its program counter, the values of its variables (stored in its register), and the contents of its stack. 27

critical section (a.k.a. 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. 33

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. 51

deadlock is when there are two or more threads waiting indefinitely for one another to release a resource. 101

determinism is when the outcome of an execution is uniquely determined by the initial state. 9

fairness is when each thread eventually can access each resource it needs to access with high probability. 97

invariant is a binary predicate over states that must hold for every reachable state of a thread. 10

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. 223

lock an object that can be owned by at most one thread at a time. Useful for implementing mutual exclusion. 35

machine instruction is an atomic operation on the Harmony virtual machine, executed by a thread. 19

model checking is a formal verification method that explores all possible executions of a program, which must have a finite number of states. 9

monitor is a programming language paradigm that supports mutual exclusion as well as waiting for resources to become available. 238

mutual exclusion is the property that two threads never enter the same critical section. 34

non-blocking synchronization (a.k.a. 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. 139

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. 87

property describes a set of execution traces or behaviors or histories that are allowed by a program. Safety properties are properties in which !dblquote(no bad things happen,)! such as violating mutual exclusion in a critical section. Liveness properties are properties where !dblquote(something good eventually happens,)! like threads being able to enter the critical section if they want to. 97

race condition describes when multiple threads access shared state concurrently, leading to undesirable outcomes. 23

reader/writer lock is a lock on a resource that can be held by multiple threads if they all only read the resource. 84

replication maintains multiple copies of some resource to improve availability in the face of failures. 154

semaphore is a counter that can be atomically incremented and decremented, but blocks the thread until the counter is larger than zero first. 235

sequential consistency is a consistency model in which shared memory accesses are executed in an order consistent with the program order. 40

sequential execution is when there is just one thread executing, as opposed to concurrent execution. 9

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. 9

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. 51

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. 28

starvation is when a thread cannot make progress because it is continuously losing a competition with other threads to get access to a resource. 97

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. 230

state machine replication is a replication technique in which a collection of deterministic state machines process the same inputs in the same order. 154

step is the execution of a machine instruction by a thread, updating its state. 230

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. 19

thread safety is when the implementation of a data structure allows concurrent access with well-defined semantics. 33

thread variable is a variable that is private to a single thread and stored in its register. 230

trace is a sequence of steps, starting from an initial state. 230