Concurrent Programming in Harmony

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

1. On Concurrent Programming 5
2. Introduction to Harmony 9
3. The Problem of Concurrent Programming 11
4. The Harmony Virtual Machine 16
5. Critical Sections 24
6. Peterson’s Algorithm 31
7. Harmony Methods and Pointers 39
8. Spinlock 43
9. Blocking 47
10. Concurrent Data Structures 55
11. Testing 62
12. Debugging 73
13. Conditional Waiting 78
   13.1 Reader/Writer Locks 78
   13.2 Bounded Buffer 80
14. Split Binary Semaphores 81
15. Starvation 86
16. Monitors 89
17. Deadlock 98
18. Actors and Message Passing 104
Chapter 1

On Concurrent Programming

Programming with concurrency is hard. On the one hand concurrency can make programs faster than sequential ones, but having multiple threads read and update shared variables concurrently and synchronize with one another makes programs more complicated than programs where only one thing happens at a time. Why are concurrent programs more complicated than sequential ones? There are, at least, two reasons:

- The execution of a sequential program is mostly *deterministic*. If you run it twice with the same input, the same output will be produced. Bugs are typically easily reproducible and easy to track down, for example by instrumenting the program. On the other hand, the output of running concurrent programs depends on how the execution of the various threads are *interleaved*. Some bugs may occur only occasionally and may never occur when the program is instrumented to find them (so-called *Heisenbugs*—overhead caused by instrumentation leads to timing changes that makes such bugs less likely to occur).

- In a sequential program, each statement and each function can be thought of as happening *atomically* (indivisibly) because there is no other activity interfering with their execution. Even though a statement or function may be compiled into multiple machine instructions, they are executed back-to-back until completion. Not so with a concurrent program, where other threads may update memory locations while a statement or function is being executed.

The lack of determinism and atomicity in concurrent programs make them not only hard to reason about, but also hard to test. Running the same test of concurrent code twice is likely to produce two different results. More problematically, a test may trigger a bug only for certain “lucky” executions. Due to the probabilistic nature of concurrent code, some bugs may be highly unlikely to get triggered even when running a test millions of times. And even if a bug does get triggered, the source of the bug may be hard to find because it is hard to reproduce.

This book is intended to help people with understanding and developing concurrent code, which includes programs for distributed systems. In particular, it uses a new 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 in the smallest number of steps.
Model checking is not a replacement for formal verification. Formal verification proves that a program is correct. Model checking only verifies that a program is correct for some model. Think of a model as a test program. Because model checking tries every possible execution, the test program needs to be simple. Otherwise it may take longer than we care to wait for or run out of memory. In particular, the model needs to have a relatively small number of reachable states.

If model checking does not prove a program correct, why is it useful? To answer that question, let us consider a sorting algorithm. Suppose we create a test program, a model, that tries sorting all lists of up to five numbers chosen from the set \{1, 2, 3, 4, 5\}. Model checking proves that for those particular scenarios the sorting algorithm works: the output is a sorted permutation of the input. In some sense it is an excellent test: it will have considered all corner cases, including lists where all numbers are the same, lists that are already sorted or reversely sorted, etc. If there is a bug in the sorting algorithm, most likely it would be triggered and the model checker would produce a scenario that would make it easy to find the source of the bug.

However, if the model checker does not find any bugs, we do not know for sure that the algorithm works for lists of more than five numbers or for lists that have values other than the numbers 1 through 5. Still, we would expect that the likelihood that there are bugs remaining in the sorting algorithm is small. That said, it would be easy to write a program that sorts all lists of up to five numbers correctly but fails to do so for a list of 6 numbers. (Hint: simply use an if statement.)

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

So, what is Harmony? Harmony is a concurrent programming language. It was designed to teach the basics of concurrent 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. Interestingly, Harmony does not have input/output statements, so there is no “Hello World” program.

The syntax and semantics of Harmony is similar to that of Python. Python is familiar to many programmers and is easy to learn and use. We will assume that the reader is familiar with the basics of Python programming. We also will assume that the reader understands some basics of machine architecture and how programs are executed. For example, we assume that the reader is familiar with the concepts of CPU, memory, register, stack, and machine instructions.

Harmony is heavily influenced by Leslie Lamport’s work on TLA+, TLC, and PlusCal [Lam02, Lam09], Gerard Holzmann’s work on Promela and SPIN [Hol11], and University of Washington’s DSLabs system [MWA+19]. Harmony is designed to have a lower learning curve than those systems, but is not as powerful. When you finish this book and want to learn more, we strongly encourage checking those out. Another excellent resource is Fred Schneider’s book “On Concurrent Programming” [Sch97]. (This chapter is named after that book.)

The book proceeds as follows:

- **Chapter 2** introduces the Harmony programming language, as it provides the language for presenting synchronization problems and solutions.
Chapter 3 illustrates the problem of concurrent programming through a simple example in which two threads are concurrently incrementing a counter.

Chapter 4 presents the Harmony virtual machine to understand the problem underlying concurrency better.

Chapter 5 introduces the concept of a critical section and presents various flawed implementations of critical sections to demonstrate that implementing a critical section is not trivial.

Chapter 6 introduces Peterson’s Algorithm, an elegant (although not very efficient or practical) solution to implementing a critical section.

Chapter 7 gives some more details on the Harmony language needed for the rest of the book.

Chapter 8 introduces atomic locks for implemented critical sections.

Chapter 9 presents Harmony modules that support locks and other synchronization primitives.

Chapter 10 gives an introduction to building concurrent data structures.

Chapter 11 discusses approaches to testing concurrent code in Harmony.

Chapter 12 instead goes into how to find a bug in concurrent code using the Harmony output.

Chapter 13 talks about threads having to wait for certain conditions. As examples, it presents the reader/writer lock problem and the bounded buffer problem.

Chapter 14 presents Split Binary Semaphores, a general technique for solving synchronization problems.

Chapter 15 talks about starvation: the problem that in some synchronization approaches threads may not be able to get access to a resource they need.

Chapter 16 presents monitors and condition variables, another approach to thread synchronization.

Chapter 17 describes deadlock where a set of threads are indefinitely waiting for one another to release a resource.

Chapter 18 presents the actor model and message passing as an approach to synchronization.

Chapter 19 describes barrier synchronization, useful in high-performance computing applications such as parallel simulations.

Chapter 20 discusses how to handle interrupts, a problem closely related to—but not the same as—synchronizing threads.

Chapter 21 introduces non-blocking or wait-free synchronization algorithms, which prevent threads waiting for one another more than a bounded number of steps.

Chapter 22 presents a problem and a solution to the distributed systems problem of having two threads communicate reliably over an unreliable network.
• Chapter 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 presents a protocol for a fault-tolerant replicated object that supports only read and write operations.

• Chapter 27 demonstrates a fault-tolerant distributed consensus algorithm (aka protocol) expressed in Harmony.

• Chapter 28 shows how one can specify and check the well-known Paxos consensus protocol.

• Chapter 29 demonstrates finding a bug in the original Needham-Schroeder authentication protocol.

The appendices contain various details about Harmony itself, including an appendix that explains how Harmony works (Appendix D) and an appendix on installing and running Harmony (Appendix E).
Chapter 2

Introduction to Harmony

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

- Harmony only supports basic operator precedence or associativity. Use parentheses liberally to remove ambiguity.
- Harmony does not (currently) support floating point, iterators, or I/O; Harmony does support for loops and various “comprehensions.”
- Python is object-oriented, supporting classes with methods and inheritance; Harmony has objects but does not support classes. On the other hand, Harmony supports pointers to objects and methods.

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

Figure 2.1 shows a simple example of a Harmony program. (The code for examples in this book can be found in the code folder under the name listed in the caption of the example.) The

```
const N = 10

def triangle(n):  # computes the n’th triangle number
    result = 0
    for i in {1..n}:  # for each integer from 1 to n inclusive
        result += i  # add i to result

x = choose({0..N})  # select an x between 0 and N inclusive
assert triangle(x) == ((x * (x + 1)) / 2)
```

Figure 2.1: [code/triangle.hny] Computing triangle numbers.
example is sequential and has a method \texttt{triangle} that takes an integer number as argument. Each method has a variable called \texttt{result} that eventually contains the result of the method (there is no \texttt{return} statement in Harmony). The method also has a variable called \texttt{n} containing the value of the argument. The \{ \texttt{x..y} \} notation generates a set containing the numbers from \texttt{x} to \texttt{y} (inclusive). (Harmony does not have iterators and in particular does not have a \texttt{range} operator.) The last two lines in the program are the most interesting. The first assigns to \texttt{x} some unspecified value in the range \texttt{0..N} and the second verifies that \texttt{triangle}(\texttt{x}) equals \texttt{x}(\texttt{x} + 1)/2.

“Running” this Harmony program will try all possible executions, which includes all possible values for \texttt{x}. Try it out (here \$ represents a shell prompt):

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

(For this to work, make sure \texttt{harmony} is in your command shell’s search path.) Essentially, the \texttt{choose(S)} operator provides the input to the program by selecting some value from the set \texttt{S}, while the \texttt{assert} statement checks that the output is correct. If the program is correct, the output of Harmony is the size of the “state graph” (13 states in this case). If not, Harmony also reports what went wrong, typically by displaying a summary of an execution in which something went wrong.

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

```
$ harmony -c N=100 triangle.hny
#states 103
103 components, 0 bad states
No issues
$
```

**Exercises**

2.1 See what happens if, instead of initializing \texttt{result} to 0, you initialize it to 1. (You do not need to understand the error report at this time. They will be explained in more detail in Chapter 4.)

2.2 Write a Harmony program that computes squares by repeated adding. So, the program should compute the square of \texttt{x} by adding \texttt{x} to an initial value of 0 \texttt{x} times.
Chapter 3

The Problem of Concurrent Programming

Concurrent programming, aka multithreaded programming, involves multiple threads running in parallel while sharing variables. Figure 3.1 shows two programs. Program (a) is sequential. It sets shared to \texttt{True}, asserts that \texttt{shared = True}, and finally sets \texttt{shared} to \texttt{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 \texttt{f()} and \texttt{g()} in parallel. If method \texttt{g()} runs and completes before \texttt{f()}, then the assertion in \texttt{f()} will fail when \texttt{f()} runs. Harmony will find this problem. This problem is an example of non-determinism: methods \texttt{f()} and \texttt{g()} can run in either order.

Figure 3.2 presents a more subtle example that illustrates non-atomicity. The program initializes two shared variables: an integer \texttt{count} and an array \texttt{done} with two booleans. The program then spawns two threads. The first runs \texttt{incrementer(0)}; the second runs \texttt{incrementer(1)}.
Method `incrementer` takes a parameter called `self`. It increments `count` and sets `done[self]` to `True`. It then waits until the other thread is done. (*await* `c` is shorthand for *while not c: pass.*) After that, method `incrementer` verifies that the value of `count` equals 2.

Note that although the threads are spawned one at a time, they will execute concurrently. It is, for example, quite possible that `incrementer(1)` finishes before `incrementer(0)` even gets going. And because Harmony tries every possible execution, it will consider that particular execution as well. What would the value of `count` be at the end of that execution?

```python
count = 0
done = [False, False]

def incrementer(self):
    count = count + 1
    done[self] = True
    await done[1 - self]
    assert count == 2

spawn incrementer(0)
spawn incrementer(1)
```

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

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

What is going on is that the Harmony program is compiled to machine instructions, and it is the machine instructions that are executed by the underlying Harmony machine. The details of this appear in Chapter 4, but suffice it to say that the machine has instructions that load values from memory and store values into memory. Importantly, it does not have instructions to atomically increment or decrement values in shared memory locations. So, to increment a value in memory, the machine must do at least three machine instructions. Conceptually:

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

When running multiple threads, each essentially runs an instantiation of the machine, and they do so in parallel. As they execute, their machine instructions are interleaved in unspecified and often unpredictable ways. A program is correct if it works for any interleaving of threads. Harmony will try all possible interleavings of the threads executing machine instructions.
If the threads run one at a time, then $count$ will be incremented twice and ends up being 2. However, the following is also a possible interleaving of $\text{incrementer}(0)$ and $\text{incrementer}(1)$:

1. $\text{incrementer}(0)$ loads the value of $count$, which is 0;
2. $\text{incrementer}(1)$ loads the value of $count$, which is still 0;
3. $\text{incrementer}(1)$ adds 1 to the value that it loaded (0), and stores 1 into $count$;
4. $\text{incrementer}(0)$ adds 1 to the value that it loaded (0), and stores 1 into $count$;
5. $\text{incrementer}(0)$ sets $\text{done}[0]$ to True;
6. $\text{incrementer}(1)$ sets $\text{done}[1]$ to True.

The result in this particular interleaving is that $count$ ends up being 1. This is known as a race condition. When running Harmony, it will report violations of assertions. It also provides an example of an interleaving, like the one above, in which an assertion fails.

If one thinks of the assertion as providing the specification of the program, then clearly its implementation does not satisfy its specification. Either the specification or the implementation (or both) must have a bug. We could change the specification by changing the assertion as follows:

```python
assert (count == 1) or (count == 2)
```

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

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

### Exercises

The following exercises are intended to show you that while it is just as easy to write concurrent programs in Python, it is much easier to find concurrency bugs using Harmony.

#### 3.1 Harmony programs can usually be easily translated into Python. For example, Figure 3.3 is a Python version of Figure 3.2.

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

1. Actually, Harmony still complains, this time about a data race, about which you will learn in Chapter 4.
import threading

count = 0
done = [False, False]

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

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

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

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

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

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

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

3.2 Figure 3.4 is a version of Figure 3.3 that has each incremener thread increment count N times. Run Figure 3.4 10 times (using Python). Report how many times the assertion fails and what the value of count was for each of the failed runs. Also experiment with lower values of N. How large does N need to be for assertions to fail? (Try powers of 10 for N.)

3.3 Can you think of a fix to Figure 3.2? Try one or two different fixes and run them through Harmony. Do not worry about having to come up with a correct fix at this time—the important thing is to develop an understanding of concurrency. (Also, you do not get to use a lock, yet.)
import threading

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

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

threading.Thread(target=incrementer, args=(0,)).start()
threading.Thread(target=incrementer, args=(1,)).start()
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 places the bytecode for file \texttt{x.htm} in file \texttt{x.hvm}. The model checker, called \textit{Charm}, places its output in a file called \texttt{x.hco} and a stylized version of the same output in an HTML file called \texttt{x.htm}.

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 (\texttt{False} and \texttt{True}), integers (but not floating point numbers), strings (enclosed by double quotes), sets of Harmony values, and dictionaries that map Harmony values to other Harmony values. Another type of Harmony value is the \textit{atom}. It is essentially just a name. An atom is denoted using a period followed by the name. For example, \texttt{.main} is an atom.

Harmony makes extensive use of dictionaries. A dictionary maps keys to values. Unlike Python, any Harmony value can be a key, including another dictionary. Dictionaries are written as \{\texttt{k0 : v0, k1 : v1, ...}\}. If \textit{d} is a dictionary, and \texttt{k} is a key, then the following expression retrieves the Harmony value that \texttt{k} maps to in \textit{d}:

\[
d b
\]

The meaning of \texttt{d a b ...} is (((\texttt{d a}) b) ...). This notation is unfamiliar to Python programmers, but in Harmony square brackets can be used in the same way as parentheses, so you can express the same thing in the form that is familiar to Python programmers:

\[
d[k]
\]

However, if \textit{d} = \{\texttt{.count : 3}\}, then you can write \texttt{d.count} (which has value 3) instead of having to write \texttt{d[.count]} (although both will work). Thus, using atoms, a dictionary can be made to look much like a Python object.
Tuples are special forms of dictionaries where the keys are the indexes into the tuple. For example, the tuple (5, False) is the same Harmony value as { 0:5, 1:False }. The empty tuple is written as (). As in Python, you can create singleton tuples by including a comma. For example, (1,) is a tuple consisting just of the number 1. Importantly, (1) = 1 ≠ (1,).

Again, square brackets and parentheses work the same in Harmony, so [a, b, c] (which looks like a Python list) is the same Harmony value as (a, b, c) (which looks like a Python tuple), which in turn is the same Harmony value as { 0:a, 1:b, 2:c }. So, if x = [False, True], then x[0] = False and x[1] = True, just like in Python. However, when creating a singleton list, make sure you include the comma, as in [False,]. The expression [False] just means False.

Harmony is not an object-oriented language, so objects don’t have built-in methods. However, Harmony does have some powerful operators to make up for some of that. For example, dictionaries have two handy unary operators. If d is a dictionary, then keys d (or equivalently keys(d)) returns the set of keys and len d returns the size of this set.

Section A.1 provides details on all the values that Harmony currently supports.

Harmony Bytecode

A Harmony program is translated into HVM bytecode. To make it amenable to efficient model checking, the HVM is not an ordinary virtual machine, but its architecture is nonetheless representative of conventional computers and virtual machines such as the Java Virtual Machine.

Instead of bits and bytes, a HVM manipulates Harmony values. A HVM has the following components:

- Code: This is an immutable and finite list of HVM instructions, generated from a Harmony program. The types of instructions will be described later.

- Shared memory: A HVM has just one memory location containing a Harmony value.

- Threads: Any thread can spawn an unbounded number of other threads and threads may terminate. Each thread has a program counter that indexes into the code, a stack of Harmony values, and two private registers that each contain a Harmony value.

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

It may seem strange that there is only one memory location. However, this is not a limitation because Harmony values are unbounded trees. The shared memory is a dictionary that maps atoms (names of shared variables) to other Harmony values. We call this a directory. Thus, a directory represents the state of a collection of variables named by the atoms. Because directories are Harmony values themselves, directories can be organized into a tree. Each node in a directory tree is then identified by a sequence of Harmony values, like a path name in the file system hierarchy. We call such a sequence an address. For example, in Figure 3.2 the memory is a dictionary with two entries: .count and .done. And the value of entry .done is a dictionary with keys 0 and 1. So,
Figure 4.1: The first part of the HVM bytecode corresponding to Figure 3.2.
for example, the address of done[0] is the sequence [ .done, 0 ]. An address is itself a Harmony value.

Compiling the code in Figure 3.2 results in the HVM bytecode listed in Figure 4.1. You can obtain this code by invoking harmony with the -a flag like so:

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

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

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

At program counter 6 is the code for the incrementer method. All methods start with a Frame instruction and end with a Return instruction. Section C.1 provides a list of all HVM machine instructions, in case you want to read about the details. The Frame instruction lists the name of the method and the names of its arguments. The code generated from count := count + 1 in line 5 of Up.hny is as follows (see Figure 4.1):

7. The Load instruction pushes the value of the count variable onto the stack.
8. The Push instruction pushes the constant 1 onto the stack of the thread.
9. 2-ary is a + operation with 2 arguments. It pops two values from the stack (the value of count and 1), adds them, and pushes the result back onto the stack.
10. The **Store** instruction pops a Harmony value (the sum of the \textit{count} variable and 1) and stores it in the \textit{count} variable.

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

Harmony can report the following failure types:

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

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

- **Active Busy Waiting**: There are states in which some thread cannot make progress without the help of another thread but does not block;

- **Data Race**: There are states in which two or more threads concurrently access a shared variable, at least one of which is a store operation.

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

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

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

The four turns in the execution are as follows:

1. Thread \texttt{\_init\_} (with identifier T0) sets shared variable \textit{count} to 0 and shared variable \textit{done} to \texttt{[False, False]}.
2. Thread \texttt{incrementer(0)} (with identifier T1) executes instructions 6 through 9, loading the value of \textit{count} but stopping just before storing 1 into \textit{count};
3. Thread \texttt{incrementer(1)} (with identifier T2) executes instructions 6 through 18, storing 1 into \textit{count} and storing \texttt{True} into \textit{done}[1];
Figure 4.3: The HTML output of running Harmony on Figure 3.2. There are four sections. The top shows the turns to the problematic state. Each turn shows a thread identifier, the method the thread is executing, a “timeline” containing a block for each instruction executed, the program counter of the thread, and the values of the shared variables at those turns. The middle section shows the line of code that is being executed. The bottom left shows the bytecode of the program. The bottom right shows the state of each thread. Each row in the stack trace of a thread shows the method that is being evaluated and the values of its method variables.
4. Thread `incrementer(0)` continues execution, executing instructions 10 through 28 storing value 1 into `count` (instruction 10), storing `True` into `done[0]` (instructions 11 through 15) finding that `done[1]` is `True` (instructions 16 through 21), and finally detecting that the assertion is violated (instructions 22 through 28).

Harmony also generates an HTML file that allows exploring more details of the execution interactively. Open the suggested HTML file and you should see something like Figure 4.3.

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

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

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

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

- **Runnable**: the thread is runnable but not currently running. In Harmony, threads are interleaved and so at most one thread is actually running;
- **Running**: the thread is currently executing instructions;
- **Terminated**: the thread has completed all its instructions;
- **Failed**: the thread has encountered an error, such as violating an assertion or divide by zero;
- **Blocked**: the thread cannot make progress until another thread has updated the shared state. For example, this occurs when one of the implementers is waiting for the other to set its `done` flag;
- **Atomic**: the thread is in `atomic` mode, not allowing other threads to be scheduled. This is, for example, the case when an assertion is being checked;
- **Read-only**: the thread is in `read-only` mode, not able to modify shared state. Assertions can execute arbitrary code including methods, but they are not allowed to modify the shared state.

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

When you load the HTML file, it shows the state after executing the last instruction. As mentioned above, you can go to any point in the execution by clicking on
```python
    count = 0
    entered = done = [False, False]

def incremer(self):
    entered[self] = True
    if entered[1 - self]:  # if the other thread has already started
        await done[1 - self]  # wait until it is done
    count = count + 1
    done[self] = True
    await done[1 - self]
    assert count == 2

spawn incremer(0)
spawn incremer(1)
```

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

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

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

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

**Exercises**

4.1 Figure 4.4 shows an attempt at trying to fix the code of Figure 3.2. Run it through Harmony and see what happens. Based on the error output, describe in English what is wrong with the code by describing, in broad steps, how running the program can get into a bad state.

4.2 What if we moved line 5 of Figure 4.4 to after the if statement (between lines 7 and 8)? Do you think that would work? Run it through Harmony and describe either why it works or why it does not work.
Chapter 5

Critical Sections

Hopefully you have started thinking of how to solve the concurrency problem and you may already have prototyped some solutions. In this chapter we will go through a few reasonable but broken attempts. At the heart of the problem is that we would like make sure that, when the count variable is being updated, no other thread is trying to do the same thing. This is called a critical section (aka critical region) [Dij65]: a set of instructions where only one thread is allowed to execute at a time.

Critical sections are useful when accessing a shared data structure, particularly when that access requires multiple underlying machine instructions. A counter is a very simple example of a data structure, but as we have seen it too requires multiple instructions. A more involved one would be accessing a binary tree. Adding a node to a binary tree, or re-balancing a tree, often requires multiple operations. Maintaining “consistency” is certainly much easier if during this time no other thread also tries to access the binary tree. Typically, you want some invariant property of the data structure to hold at the beginning and at the end of the critical section, but in the middle the invariant may be temporarily broken—this is not a problem as critical sections guarantee that no other thread will be able to see it. An implementation of a data structure that can be safely accessed by multiple threads and free of race conditions is called thread-safe.

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

Figure 5.1: [code/csbarebones.hny] A bare bones critical section with two threads.
A critical section is often modeled as threads in an infinite loop entering and exiting the critical section. Figure 5.1 shows the Harmony code. Here `@cs` is a label, identifying a location in the HVM bytecode. The first thing we need to ensure is that there can never be two threads in the critical section. This property is called *mutual exclusion*. We would like to place an assertion at the `@cs` label that specifies that only the current thread can be there. Harmony in fact supports this. It has an operator `atLabel`, where `L` is the atom containing the name of the label (in this case, `cs`). The operator returns a bag (multiset) of (method, argument) pairs of threads executing at that label. In our case there are two threads: `(thread, 0)` and `(thread, 1)`. A bag is represented by a dictionary that maps each element in the bag to the number of times the element appears in the bag. Method `atLabel` only exists for specification purposes—do not use it in normal code. If you run the code through Harmony, the assertion should fail because there is no code yet for safely entering and exiting the critical section.

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

```
await False
```

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

Mutual exclusion is an example of a *safety property*, a property that ensures that *nothing bad will happen*, in this case two threads being in the critical section. What we need now is a *liveness property*: we want to ensure that *eventually something good will happen*. There are various possible liveness properties we could use, but here we will propose the following informally: if (1) there exists a non-empty set `S` of threads that are trying to enter the critical section and (2) threads in the critical section always leave eventually, then eventually one thread in `S` will enter the critical section. We call this *progress*.

In order to detect violations of progress, and other liveness problems in algorithms in general, Harmony requires that every execution must be able to reach a state in which all threads have terminated. Clearly, even if mutual exclusion holds in Figure 5.1, the spawned threads never terminate. We will instead model threads in critical sections using the framework in Figure 5.2: a thread can *choose* to enter a critical section more than once, but it can also choose to terminate, even
without entering the critical section ever. (Recall that Harmony will try every possible execution, and so it will evaluate both choices.)

A thread that is in the critical section cannot terminate until after leaving the critical section. We will now consider various approaches toward implementing this specification.

You may already have heard of the concept of a lock and have realized that it could be used to implement a critical section. The idea is that the lock is like a baton that at most one thread can own (or hold) at a time. A thread that wants to enter the critical section at a time must obtain the lock first and release it upon exiting the critical section.

Using a lock is a good thought, but how does one implement one? Figure 5.3 presents a mutual exclusion attempt based on a naïve (and, as it turns out, broken) implementation of a lock. Initially the lock is not owned, indicated by lockTaken being False. To enter the critical section, a thread waits until lockTaken is False and then sets it to True to indicate that the lock has been taken. The thread then executes the critical section. Finally the thread releases the lock by setting lockTaken back to False.

Unfortunately, if we run the program through Harmony, we find that the assertion still fails. Figure 5.4 shows the Harmony output. Thread 0 finds that the lock is available, but just before it stores True in lockTaken in instruction 9, thread 1 gets to run. (Recall that you can hover your mouse over a machine instruction in order to see what it does.) Because lockTaken is still False, it too believes it can acquire the lock, and stores True in lockTaken and moves on to the critical section. Finally, thread 0 moves on, also stores True into lockTaken and also moves into the critical section. Thread 0 is the one that detects the problem. The lockTaken variable suffers from the same sort of race condition as the count variable in Figure 3.2: testing and setting the lock consists
flags = [False, False]

def thread(self):
    while choose({False, True}):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        # Critical section
        @cs: assert atLabel(cs) == {thread, self: 1}

        # Leave critical section
        flags[self] = False

    spawn thread(0)
    spawn thread(1)
**Issue: Non-terminating state**

<table>
<thead>
<tr>
<th>Turn</th>
<th>Thread</th>
<th>Instructions Executed</th>
<th>PC</th>
<th>flags 0</th>
<th>flags 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T0: <em>init</em> ()</td>
<td><img src="image" alt="Instructions" /></td>
<td>54</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>T1: thread(0)</td>
<td><img src="image" alt="Instructions" /></td>
<td>16</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>T2: thread(1)</td>
<td><img src="image" alt="Instructions" /></td>
<td>16</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

**code/naiveFlags.hny:** *Naïve use of turn variable to solve mutual exclusion.*

```python
turn = 0
def thread(self):
    while choose({False, True}):  # Enter critical section
        turn = 1 - self
        await turn == self  # Critical section
        @cs: assert atLabel(cs) == {(thread, self): 1}
        # Leave critical section
        spawn thread(0)
        spawn thread(1)
```

Figure 5.7: *Naïve use of turn variable to solve mutual exclusion.*

---

Figure 5.6: The HTML output of running Harmony on Figure 5.5.
of several instructions. It is thus possible for both threads to believe the lock is available and to obtain the lock at the same time.

Preventing multiple threads from updating the same variable, Figure 5.5 presents a solution based on each thread having a flag indicating that it is trying to enter the critical section. A thread can write its own flag and read the flag of its peer. After setting its flag, the thread waits until the other thread (1 − self) is not trying to enter the critical section. If we run this program, the assertion does not fail. In fact, this solution does prevent both threads being in the critical section at the same time.

To see why, first note the following invariant: if thread i is in the critical section, then flags[i] == True. Without loss of generality, suppose that thread 0 sets flags[0] at time t_0. Thread 0 can only reach the critical section if at some time t_1, t_1 > t_0, it finds that flags[1] == False. Because of the invariant, flags[1] == False implies that thread 1 is not in the critical section at time t_1. Let t_2 be the time at which thread 0 sets flags[0] to False. Thread 0 is in the critical section sometime between t_1 and t_2. It is easy to see that thread 1 cannot enter the critical section between t_1 and t_2, because flags[1] == False at time t_1. To reach the critical section between t_1 and t_2, it would first have to set flags[1] to True and then wait until flags[0] == False. But that does not happen until time t_2.

However, if you run the program through Harmony (Figure 5.6), it turns out the solution does have a problem: if both try to enter the critical section at the same time, they may end up waiting for one another indefinitely. (This is a form of deadlock, which will be discussed in Chapter 17.) Thus the solution violates progress.

The final naïve solution that we propose is based on a variable called turn. Each thread politely lets the other thread have a turn first. When turn = i, thread i can enter the critical section, while thread 1 − i has to wait. An invariant of this solution is that while thread i is in the critical section, turn == i. Since turn cannot be 0 and 1 at the same time, mutual exclusion is satisfied. The solution also has the nice property that the thread that has been waiting the longest to enter the critical section can go next.

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

**Exercises**

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

```python
await False
```

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

5.2 See if you can come up with some different approaches that satisfy both mutual exclusion and progress. Try them with Harmony and see if they work or not. If they don’t, try to understand
why. If you get active busy waiting or data race reports, you probably found a correct solution; you’ll learn later how to suppress those. Do not despair if you can’t figure out how to develop a solution that satisfies both mutual exclusion and progress—as we will find out, it is possible but not obvious.
Chapter 6

Peterson’s Algorithm

In 1981, Gary L. Peterson came up with a beautiful solution to the mutual exclusion problem, now known as “Peterson’s Algorithm” [Pet81]. The algorithm is an amalgam of the (incorrect) algorithms in Figure 5.5 and Figure 5.7, and is presented in Figure 6.1. (The first line specifies that the flags and turn variables should have sequential consistency—it prevents Harmony from complaining about data races involving these variables, explained in Chapter 8.)

A thread first indicates its interest in entering the critical section by setting its flag. It then politely gives way to the other thread should it also want to enter the critical section—if both do so at the same time one will win because writes to memory in Harmony are atomic. The thread continues to be polite, waiting until either the other thread is nowhere near the critical section (flag\[1 - self\] == False) or has given way (turn == self). Running the algorithm with Harmony shows that it satisfies both mutual exclusion and progress.

Why does it work? We will focus here on how one might go about proving mutual exclusion for an algorithm such as Peterson’s. It turns out that doing so is not easy. If you are interested in learning more about concurrent programming but not necessarily in how to prove concurrent programs correct, you may choose to skip the rest of this chapter. If you are still here, you have to understand a little bit more about how the Harmony virtual machine (HVM) works. In Chapter 4 we talked about the concept of state: at any point in time the HVM is in a specific state. A state is comprised of the values of the shared variables as well as the values of the thread variables of each thread, including its program counter and the contents of its stack. Each time a thread executes a HVM machine instruction, the state changes (if only because the program counter of the thread changes). We call that a step. Steps in Harmony are atomic.

The HVM starts in an initial state in which there is only one thread and its program counter is 0. A trace is a sequence of steps starting from the initial state. When making a step, there are two sources of non-determinism in Harmony. One is when there is more than one thread that can make a step. The other is when a thread executes a choose operation and there is more than one choice. Because there is non-determinism, there are multiple possible traces. We call a state reachable if it is either the initial state or it can be reached from the initial state through a finite trace. A state is final when there are no threads left to make state changes.
sequential flags, turn

```python
flags = [False, False]
turn = choose({0, 1})

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

        # critical section is here
        @cs: assert atLabel(cs) == {thread, self: 1}

        # Leave critical section
        flags[self] = False

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

Figure 6.1: [code/Peterson.lny] Peterson’s Algorithm
It is often useful to classify states. Initial, final, and reachable, and unreachable are all examples of classes of states. Figure 6.2 depicts a Venn diagram of various classes of states and a trace. One way to classify states is to define a predicate over states. All states in which \( x = 1 \), or all states where there are two or more threads executing, are examples of such predicates. For our purposes, it is useful to define a predicate that says that at most one thread is in the critical section. We shall call such states exclusive.

An invariant of a program is a predicate that holds over all states that are reachable by that program. We want to show that exclusivity is an invariant because mutual exclusion means that all reachable states are exclusive. In other words, we want to show that the set of reachable states of executing the program is a subset of the set of states where there is at most one thread in the critical section.

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

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

- \( \mathcal{I} \) holds in the initial state.
• For any state in which \( I \) holds (including unreachable ones) and for any thread in the state, if the thread takes a step, then \( I \) 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 \textit{inductive} invariant. To see why, consider the following state \( s \): let thread 0 be at label \( \text{@cs} \) and thread 1 be at the start of the \textit{await} statement. Also, in state \( s \), \textit{turn} = 1. Now let thread 1 make a step. Because \textit{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 \textit{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: \( F(i) = \text{thread}(i)@[8 \cdots 15] \Rightarrow \text{flags}[i] \), that is, if thread \( i \) is executing in lines 8 through 15, then \( \text{flags}[i] \) is set. Although it does not, by itself, imply exclusivity, we can show that \( F(i) \) 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 \( F(i) \) holds in some state, then \( F(i) \) also holds in a next state. Since only thread \( i \) ever changes \( \text{flags}[i] \), we only need to consider steps by thread \( i \). Since \( F(i) \) holds, there are two cases to consider:

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

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

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

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

Instead, we will use the following property: \( G(i) = \text{thread}(i)@\text{cs} \Rightarrow C(i) \lor \text{thread}(1 - i)@\text{gate} \). That is, if thread \( i \) is at the critical section, then either \( C(i) \) holds or thread \( 1 - i \) is about to execute line 8. Figure 6.3 formalizes \( G(i) \) in Harmony. The label \( @\text{gate} \) refers to the step that sets \textit{turn} to \( 1 - i \). You can run Figure 6.3 to determine that \( G(i) \) is an invariant for \( i = 0, 1 \). Moreover, if \( F(i) \) and \( G(i) \) 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
sequential flags, turn

flags = [False, False]
turn = choose({0, 1})

def thread(self):
    while choose({'False', 'True'}):
        # Enter critical section
        flags[self] = True
        @gate: turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        # Critical section
        @cs: assert (not flags[1 - self]) or (turn == self)
        or (atLabel(gate) == {(thread, 1 - self): 1})

        # Leave critical section
        flags[self] = False

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

We claim that $G(i)$ is an inductive invariant. First, since neither thread in in the critical section in the initial state, it is clear that $G(i)$ 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 $G(0)$ holds, then any step will result in a state in which $G(0)$ still holds.

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

1. $\text{flags}[1] = \text{False}$;
2. $\text{turn} = 0$;
3. thread 1 is at label $\text{@gate}$.

Let us consider each of these cases. We have to show that if thread 1 takes a step, then one of those cases must hold after the step. In the first case, if thread 1 takes a step, there are two possibilities: either $\text{flags}[1]$ will still be $\text{False}$ (in which case the first case continues to hold), or $\text{flags}[1]$ will be $\text{True}$ and thread 1 will be at label $\text{@gate}$ (in which case the third case will hold). We know that thread 1 never sets $\text{turn}$ to 1, so if the second case holds before the step, it will also hold after the step. Finally, if thread 1 is at label $\text{@gate}$ before the step, then after the step $\text{turn}$ 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 $G(0)$ 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 $G(0)$ continues to hold;
2. Thread 0 takes a step but still is not in the critical section. Then again $G(0)$ 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 $\text{flags}[1] = \text{False}$ or $\text{turn} == 0$ because of the $\text{await}$ condition. And hence $G(0)$ 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 $C(i)$ (i.e., $G$ without the disjunct that thread $1 - i$ is at label $\text{@gate}$). To demonstrate that this predicate is not invariant, you can remove the disjunct from Figure 6.3 and run it to get a counterexample.
This anecdote suggests the following. If you need to do a proof by induction of an algorithm, you have to come up with an inductive invariant. Before trying to prove the algorithm, you can check that the predicate is at least invariant by testing it using Harmony. Doing so could potentially avoid wasting your time on a proof that will not work because the predicate is not invariant, and therefore not an inductive invariant either. (The author fixed the Wikipedia page with the help of Fred B. Schneider.)

Exercises

6.1 Figure 6.4 presents another solution to the mutual exclusion problem. It is similar to the one in Figure 5.5, but has a thread back out and try again if it finds that the other thread is either trying to enter the critical section or already has. Compare this algorithm with Peterson’s. Why does Harmony complain about active busy waiting?

6.2 Can you find one or more inductive invariants for the algorithm in Figure 6.4 to prove it correct? Here’s a pseudo-code version of the algorithm to help you. Each line is an atomic action:

```
initially: flagX = flagY = False

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

6.3 A colleague of the author asked if the first two assignments in Peterson’s algorithm (setting flags[1-self] to True and turn to 1 - self) can be reversed. After all, they are different variables assigned independent values—in a sequential program one could surely swap the two assignments. See if you can figure out for yourself if the two assignments can be reversed. Then run the program in Figure 6.1 after reversing the two assignments and describe in English what happens.

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

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

flags = [False, False]

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

        # Critical section
        @cs: assert atLabel(cs) == {thread, self}: 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)

Figure 6.4: [code/csonebit.hny] Mutual exclusion using a flag per thread.
Chapter 7

Harmony Methods and Pointers

A method \( m \) with argument \( a \) is invoked in its most basic form as follows (assigning the result to \( r \)).

\[
r = m\ a;
\]

That’s right, no parentheses are required. In fact, if you invoke \( m(a) \), the argument is \( (a) \), which is the same as \( a \). If you invoke \( m() \), the argument is \( () \), which is the empty tuple. If you invoke \( m(a, b) \), the argument is \( (a, b) \), the tuple consisting of values \( a \) and \( b \).

You may note that all this looks familiar. Indeed, the syntax is the same as that for dictionaries (see Chapter 4). Both dictionaries and methods map Harmony values to Harmony values, and their syntax is indistinguishable. If \( f \) is either a method or a dictionary, and \( x \) is an arbitrary Harmony value, then \( f\ x, f(x), \) and \( f[x] \) are all the same expression in Harmony.

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

Harmony is not an object-oriented language like Python is. In Python you can pass a reference to an object to a method, and that method can then update the object. In Harmony, it is also sometimes convenient to have a method update a shared variable specified as an argument. For this, as mentioned in Chapter 4, each shared variable has an `address`, itself a Harmony value. If \( x \) is a shared variable, then the expression `?x` is the address of \( x \). If a variable contains an address, we call that variable a `pointer`. If \( p \) is a pointer to a shared variable, then the expression `!p` is the value of the shared variable. In particular, `!?x == x`. This is similar to how C pointers work (`*&x == x`).

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

Figure 7.1 again shows Peterson’s algorithm, but this time with methods defined to enter and exit the critical section. The name `mutex` is often used to denote a variable or value that is used for mutual exclusion. `P_mutex` is a method that returns a “mutex,” which, in this case, is a dictionary
```python
def P_enter(pm, pid):
    pm→flags[pid] = True
    pm→turn = 1−pid
    await (not pm→flags[1−pid]) or (pm→turn == pid)

def P_exit(pm, pid):
    pm→flags[pid] = False

def P_mutex():
    result = { .turn: choose({0, 1}), .flags: [ False, False ] }

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

    sequential mutex
    mutex = P_mutex()

def thread(self):
    while choose({ False, True }):
        P_enter(?mutex, self)
        @cs: assert atLabel(cs) == { (thread, self): 1 }
        P_exit(?mutex, self)
        spawn thread(0)
        spawn thread(1)
```

Figure 7.1: [code/PetersonMethod.hny] Peterson’s Algorithm accessed through methods.
const FIFO = False

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

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

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

const VALUES = { 1..5 }

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

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

that contains Peterson’s Algorithm’s shared memory state: a turn variable and two flags. Both methods $P_{\text{enter}}$ and $P_{\text{exit}}$ take two arguments: a pointer to a mutex and the thread identifier (0 or 1). $pm→turn$ is the value of the .turn key in the dictionary that $pm$ points to.

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

Finally, methods can have local variables. Different from Python, Harmony local variables must be explicitly bound within a scope. You have already seen examples: the arguments to the method, the variable result, and the bound variable (or variables) within a for statement. The let statement allows declaring additional local variables. For example: let $x = 3$: $y += x$ adds 3 to the global variable $y$. See Section A.4 for more information.

If you are ready to learn about how locks are implemented in practice, you can now skip the rest of this chapter. But if you would like to see a cool example of using the concepts introduced in this chapter, hang on for a sequential Harmony program that finds anomalies in page replacement
algorithms. In 1969, 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 7.2 will find exactly the same scenario if you define FIFO to be True. Moreover, if you define FIFO to be False, it will find a scenario for the CLOCK replacement policy [Cor69], often used in modern operating systems.

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

The program declares two caches: one with 3 entries (clock3) and one with 4 entries (clock4).

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

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

In case you want to check if you get the right results. For FIFO, the program finds (in a few minutes) the same anomaly that Bélády et al. found: 1 2 3 4 1 2 5 1 2 3 4 5. For the CLOCK algorithm the program actually finds a shorter reference string: 1 2 3 4 2 1 2 5 1 2.
Chapter 8
Spinlock

Figure 5.3 showed a faulty attempt at solving mutual exclusion using a lock. The problem with the implementation of the lock is that checking the lock and setting it if it is available is not \textit{atomic}. Thus multiple threads contending for the lock can all “grab the lock” at the same time. While Peterson’s algorithm gets around the problem, it is not efficient, especially if generalized to multiple threads. Worse, Peterson relies on load and store operations to be executed atomically, but this may not be the case. There are a variety of possible reasons for this.

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

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

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

Peterson’s algorithm relies on a \textit{sequential consistent memory model} and hence the \textbf{sequential} statement: without it Harmony will complain about data races. More precisely, the \textbf{sequential} statement says that the program relies on memory load and store instructions operating on the indicated variables to be performed sequentially, and that this order should be consistent with the order of operations invoked on each thread. The default memory models of C and Java are not
const N = 3

shared = False
private = [True,] * N

invariant len(x for x in [shared,] + private where not x) <= 1

def tas(s, p):
    atomic:
    !p = !s
    !s = True

def clear(s):
    atomic:
    assert !s
    !s = False

def thread(self):
    while choose({False, True}):
        # Enter critical section
        while private[self]:
            tas(shared, private[self])

        # Critical section
        @cs: assert (not private[self]) and (atLabel(cs) == { (thread, self): 1 })

        # Leave critical section
        private[self] = True
        clear(shared)

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

Figure 8.1: [code/spinlock.hny] Mutual Exclusion using a “spinlock” based on test-and-set.
sequentially consistent. The unfortunately named `volatile` keyword in Java has a similar function as Harmony’s `sequential` keyword. Like many constructions in Java, its `volatile` keyword was borrowed from C and C++. However, in C and C++ they do not provide sequential consistency, and one cannot implement Peterson’s algorithm in C or C++ directly.

For proper synchronization, multi-core processors provide so-called atomic instructions: special machine instructions that can read memory and then write it in an indivisible step. While the HVM does not have any specific built-in atomic instructions besides loading and storing variables, it does have support for executing multiple instructions atomically. This feature is available in the Harmony language in two ways. First, any Harmony statement can be made atomic by placing a label in front of it. Second, a group of Harmony statements can be made atomic through the `atomic` statement. We can use `atomic` statement blocks to implement a wide variety of atomic operations.

For example, we could fix the program in Figure 3.2 by constructing an atomic increment operation for a counter, like so:

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

To support implementing locks, many CPUs have an atomic “test-and-set” (TAS) operation. Method `tas` in Figure 8.1 shows its specification. Here `s` points to a shared Boolean variable and `p` to a private Boolean variable, belonging to some thread. The operation copies the value of the shared variable to the private variable (the “test”) and then sets the shared variable to `True` (“set”).

Figure 8.1 goes on to implement mutual exclusion for a set of `N` threads. The approach is called spinlock, because each thread is “spinning” (executing a tight loop) until it can acquire the lock. The program uses `N + 1` variables. Variable `shared` is initialized to `False` while `private[i]` for each thread `i` is initialized to `True`.

An important invariant, $I_1$, of the program is that at any time at most one of these variables is `False`. Another invariant, $I_2(i)$, is that if thread `i` is executing between lines 18–22 (which includes the critical section), then `private[i] == False`. Between the two (i.e., $I_1 \land \forall i : I_2(i)$), it is clear that only one thread can be in the critical section at the same time. For those who read through the previous chapter: $I_1 \land \forall i : I_2(i)$ is an inductive invariant.

To see that invariant $I_1$ is maintained, note that `!p == True` upon entry of `tas` (because of the condition on the `while` loop that the `tas` method is invoked in). There are two cases:

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

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

Invariant $I_1$ 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 $I_2(i)$ is obvious as (i) thread `i` only proceeds to the critical section if `private[i]` is `False`, and (ii) no other thread modifies `private[i]`.

Harmony can check these invariants as well. $I_2(i)$ is checked by the `assert` statement. But how would one go about checking an invariant like $I_1$? Invariants must hold for every state. For
we only need an assertion at label $\texttt{cs}$ because the premise is that there is a thread at that label. However, we would like to check $I_3$ in every state (after the variables have been initialized). Harmony supports checking such invariants using the \texttt{invariant} keyword. The expression counts the number of \texttt{False} values and checks that the result is less than or equal to 1. Harmony checks the expression in every reachable state.

\textbf{Exercises}

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

\textbf{8.2} Implement a spinlock using the atomic swap operation.

\textbf{8.3} For the solution to Exercise 8.2, write out the invariants that need to hold and check them using Harmony.
Chapter 9

Blocking

In Figure 8.1 we have shown a solution based on a shared variable and a private variable for each thread. The private variables themselves are actually implemented as shared variables, but they are accessed only by their respective threads. A thread usually does not keep explicit track of whether it has a lock or not, because it is usually implied by the control flow of the program—a thread implicitly knows that when it is executing in a critical section it has the lock. There is no need to keep private as a shared variable—we only did so to be able to show and check the invariants. Figure 9.1 shows a more straightforward implementation of spinlock. The solution is similar to the naïve solution of Figure 5.3, but uses test-and-set to check and set the lock variable atomically. The lock is also cleared in an atomic block to prevent a data race. This approach is general for any number of threads.

It is important to appreciate the difference between an atomic section (the statements executed within an atomic statement) and a critical section (protected by a lock of some sort). The former ensures that while the atomic statement is executing no other thread can execute. The latter allows multiple threads to run concurrently, just not within the critical section. The former is rarely available to a programmer (e.g., none of Python, C, or Java support it), while the latter is very common.

In Harmony, atomic statements allow you to implement your own atomic primitives such as test-and-set. Other common examples include compare-and swap and fetch-and-add. atomic statements are not intended to replace locks or other synchronization primitives. When solving synchronization problems you should not directly use atomic statements but use the synchronization primitives that are available to you. But if you want to design a new synchronization primitive, then use atomic by all means. You can also use atomic statements in your test code. In fact, as mentioned before, assert statements are executed atomically.

Locks are probably the most prevalent and basic form of synchronization in concurrent programs. Typically, whenever you have a shared data structure, you want to protect the data structure with a lock and acquire the lock before access and release it immediately afterward. In other words, you want the access to the data structure to be a critical section. That way, when a thread makes modifications to the data structure that take multiple steps, other threads will not see the intermediate inconsistent states of the data structure.

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

def tas(s):
    atomic:
        result = !s
        !s = True

def clear(s):
    atomic:
        assert !s
        !s = False

def thread():
    while choose({False, True}):
        await not tas(lock)
        @cs: assert atLabel/cs == {thread, (): 1}
        clear(lock)
    for i in {1..10}:
        spawn thread()
def tas(lk):
    atomic:
        result = lk
        lk = True

def BinSema(acquired):
    result = acquired

def Lock():
    result = BinSema(False)

def acquire(binsema):
    await not tas(binsema)

def release(binsema):
    atomic:
        assert binsema
        binsema = False

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

def tas(lk):
    atomic:
        result = lk
        lk = True

def BinSema(acquired):
    result = acquired

def Lock():
    result = BinSema(False)

def acquire(binsema):
    await not tas(binsema)

def release(binsema):
    atomic:
        assert binsema
        binsema = False

from synch import Lock, acquire, release

sequential done

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

def thread(self):
    acquire(countlock)
    count = count + 1
    release(countlock)
    done[self] = True
    await done[1 - self]
    assert count == 2

spawn thread(0)
spawn thread(1)

Figure 9.3: [code/UpLock.hny] Program of Figure 3.2 fixed with a lock.
there is a state in which multiple threads are trying to access the same variable, and at least one of those accesses updates the variable. In many environments, including C and Java programs, the behavior of concurrent load and store operations have tricky or even undefined semantics. One should therefore avoid data races, which is why Harmony reports them even though Harmony has sequentially consistent memory.

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

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

Harmony has a module called synch that includes support for binary semaphores and locks. Figure 9.2 shows how they are implemented, and Figure 9.3 gives an example of how they may be used, in this case to fix the program of Figure 3.2. Notice that the module hides the implementation of binary semaphores and locks. The synch module includes a variety of other useful synchronization primitives, which will be discussed in later chapters. Also notice the use of the sequentially consistent done variable for testing purposes.

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

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

\[
\text{stop } s
\]

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

\[
\text{go } C R;
\]

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

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

def BinSema(acquired):
    result = { .acquired: acquired, .suspended: [ ] }

def Lock():
    result = BinSema(False)

def acquire(binsema):
    atomic:
        if binsema→acquired:
            stop binsema→suspended[len binsema→suspended]
            assert binsema→acquired
        else:
            binsema→acquired = True

def release(binsema):
    atomic:
        assert binsema→acquired
        if binsema→suspended == [ ]:
            binsema→acquired = False
        else:
            go (list.head(binsema→suspended)) ()
            binsema→suspended = list.tail(binsema→suspended)

Figure 9.4: [modules/synchS.hny] The binary semaphore interface in the synchS module uses suspension.
A binary semaphore maintains both a boolean indicating whether the binary semaphore is currently acquired and a list of contexts of threads that want to acquire the binary semaphore.

acquire() acquires the binary semaphore if available and suspends the invoking thread if not. In the latter case, the context of the thread is added to the end of the list of contexts. Note that stop is called within an atomic statement—this is the only exception to an atomic statement running to completion. While the thread is running no other threads can run, but when the thread suspends itself other threads can run.

release() checks to see if any threads are waiting to acquire the binary semaphore. If so, it uses the head and tail methods from the list module (see Appendix B) to resume the first thread that got suspended and to remove its context from the list.

Selecting the first thread is a design choice. Another implementation could have picked the last one, and yet another implementation could have used choose to pick an arbitrary one. Selecting the first is a common choice in binary semaphore implementations as it prevents starvation: every thread gets a chance to acquire the binary semaphore (assuming every thread eventually releases it). Chapter 15 will talk more about starvation and how to prevent it.

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

| harmony -m synch=synchS x.hny |

runs the file x.hny using the suspension version of the synch module. You will find that using the synchS module often leads to the model checker searching a significantly larger state space than using the synch module. Part of the reason is that the synchS module keeps track of the order in which threads wait to acquire a binary semaphore, while the synch module does not.

Exercises

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

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

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

9.4 People who use an ATM often first check their balance and then withdraw a certain amount of money not exceeding their balance. A negative balance is not allowed. Figure 9.6 shows two operations on bank accounts: one to check the balance and one to withdraw money. Note that all operations on accounts are carefully protected by a lock (i.e., there are no data races). The customer method models going to a particular ATM and withdrawing money not exceeding the balance. Running the code through Harmony reveals that there is a bug. It is a common type
of concurrency bug known as *Time Of Check Time Of Execution* (TOCTOE). In this case, by the time the withdraw operation is performed, the balance can have changed. Fix the code in Figure 9.6. Note, you should leave the customer code the same. You are only allowed to change the `atm` methods, and you cannot use the `atomic` statement.
from synch import Lock, acquire, release

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

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

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

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

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

def customer(atm, acct, amount):
    let bal = atm_check_balance(acct):
        if amount <= bal:
            atm_withdraw(acct, amount)
    for i in {1..N_ATMS}:
        spawn customer(i, choose({0..N_ACCOUNTS–1}),
                              choose({0..MAX_BALANCE}))

Figure 9.6: [code/atm.hny] Withdrawing money from an ATM.
Chapter 10

Concurrent Data Structures

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

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

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

We will first implement the queue as a linked list. The implementation in Figure 10.2 uses the alloc module for dynamic allocation of nodes in the list using \text{malloc()} and \text{free()}. \text{malloc}(v) returns a new memory location initialized to \( v \), which should be released with \text{free()} when it is no longer in use. The queue maintains a head pointer to the first element in the list and a tail pointer to the last element in the list. The head pointer is \text{None} if and only if the queue is empty. (\text{None} is a special address value that is not the address of any memory location.)

\text{queue.Queue()} returns a new queue object consisting of a \text{None} head and tail pointer and a lock. \text{queue.put}(q, v) and \text{queue.get}(q) both take a pointer \( q \) to the queue object because both may modify the queue. Before they access the value of the head or tail of the queue they first obtain the lock. When they are done, they release the lock.

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

\[
\text{assert queue.get(q) in \{ None, 1, 2 \}}
\]

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

Figure 10.3 shows another concurrent queue implementation [MS96]. It is well-known, but what is not often realized is that it requires sequentially consistent memory, which is not said explicitly in the paper. As a result, the algorithm is not guaranteed to work correctly with most
import queue

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

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

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

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

modern programming languages and computer hardware. But it is still useful to study it. The
implementation uses separate locks for the head and the tail, allowing a put and a get operation to
proceed concurrently. To avoid contention between the head and the tail, the queue uses a dummy
node at the head of the linked list. Except initially, the dummy node is the last node that was
dequeued. Note that neither the head nor tail pointer are ever None. The problem is when the
queue is empty and there are concurrent get and put operations. They obtain separate locks and
then concurrently access the next field in the dummy node—a data race with undefined semantics
in most environments.

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

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

The list has two dummy “book-end” nodes with values -inf and inf (similar to the Python
math.inf constant). An invariant of the algorithm is that at any point in time the list is “valid,”
starting with a -inf node and ending with a inf node.

Each node has a lock, a value, and next, a pointer to the next node (which is None for the
final inf node). The find(lst, v) helper method first finds and locks two consecutive nodes
before and after such that before->data.value < v ≤ after->data.value. It does so by performing
something called hand-over-hand locking. It first locks the first node, which is the -inf node. Then,
iteratively, it obtains a lock on the next node and release the lock on the last one, and so on, similar
to climbing a tree hand-over-hand. Using find the insert, remove, and contains methods are
fairly straightforward.
from synch import Lock, acquire, release
from alloc import malloc, free

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

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

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

Figure 10.2: [code/queue.hny] A basic concurrent queue data structure.
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    let dummy = malloc({ .value: (), .next: None });
    result = { .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
    acquire(?q→tllock)
    q→tail→next = node
    q→tail = node
    release(?q→tllock)

    def get(q):
        acquire(?q→hdlock)
        let dummy = q→head
        let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
        free(dummy)

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):  # allocate and initialize a new list node
    result = malloc({.lock: Lock(), .value: v, .next: n})

def _find(lst, v):
    let before = lst:
    acquire(?before→lock)
    let after = before→next:
    acquire(?after→lock)
    while after→value < v:
        release(?before→lock)
        before = after
        after = before→next
    acquire(?after→lock)
    result = (before, after)

def LinkedList():
    result = _node(-inf, _node(inf, None))

def insert(lst, v):
    let before, after = _find(lst, v):
    if after→value != v:
        before→next = _node(v, after)
    release(?after→lock)
    release(?before→lock)

def remove(lst, v):
    let before, after = _find(lst, v):
    if after→value == v:
        before→next = after→next
        release(?after→lock)
        free(after)
    else:
        release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
    result = after→value == v
    release(?after→lock)
    release(?before→lock)

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

mylist = LinkedList()

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

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

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

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

Like the queue in Figure 10.2, the implementation of the list is linearizable [HW90], a strong notion of consistency that makes it appear as if each of the operations executes atomically at some point between their invocation and return. Determining if an implementation of a concurrent data structure is linearizable involves finding what are known as the linearization points of the operations in an execution. These are the unique points in time at which an operation appears to execute atomically. The linearization point for the insert operation coincides exactly with the update of the before.next pointer. The linearization point of a contains method execution depends on whether the value is found or not. If found, it coincides with retrieving the pointer to the node that has the value. If not found, it coincides with retrieving the pointer to the inf node.

Exercises

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

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

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

- If the list is empty, $q$→tail should be None. Otherwise, the last element in the linked list starting from $q$→head should equal $q$→head. Moreover, $q$→tail→next should be None;
- The length field that you added in Exercise 10.3 should equal the length of the list.
Method check\((q)\) should not obtain a lock; instead add the following line just before releasing the lock in put and get:

\[
\text{assert check()}
\]

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

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

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

10.7 Add methods to the data structure in Figure 10.4 that report the size of the list, the minimum value in the list, the maximum value in the list, and the sum of the values in the list. Are they linearizable? If so, what are their linearization points?

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

10.9 Create a binary tree that uses, instead of a single lock per tree, a lock for each node in the tree.
Chapter 11

Testing

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

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

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

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

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

implq: the queue of the implementation;
import synch

lock = synch.Lock()

count = 0

invariant 0 <= count <= 1

def thread():
    synch.acquire(?lock)

    atomic: count += 1

    # critical section is here
    assert count == 1

    atomic: count -= 1

    synch.release(?lock)

for i in {1..100}:
    spawn thread()

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

import list

def Queue():
    result = []

def put(q, v):
    !q = list.append(!q, v)

def get(q):
    if !q == []:
        result = None
    else:
        result = list.head(!q)
    !q = list.tail(!q)

Figure 11.2: [code/queuespec.hny] Sequential queue specification.
```python
import queue, queuespec

const NOPS = 4
const VALUES = { 1..NOPS }

implq = queue.Queue()
specq = queuespec.Queue()

for i in {1..NOPS}:
    let op = choose({ .get, .put }):
        if op == .put:
            let v = choose(VALUES):
                queue.put(?implq, v)
                queuespec.put(?specq, v)
        else:
            let v = queue.get(?implq)
            let w = queuespec.get(?specq):
                assert v == w
```

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

`specq`: the queue of the specification

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

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

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

We briefly mentioned the notion of `linearizability` in Chapter 10. Basically, we want a concurrent queue to behave consistently with a sequential queue in that all `put` and `get` operations should
```python
from synch import Lock, acquire, release
from alloc import malloc, free

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

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

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

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

Figure 11.4: [code/queuelin.hmy](#) Queue implementation with linearization points.
import queuelin, queuespec

const NOPS = 4
const VALUES = {1..NOPS}

sequential qtime
qtime = 0

implq = queuelin.Queue()
specq = queuespec.Queue()

def thread():
    let op = choose({.get, .put}):
        if op == .put:
            let v = choose(VALUES):
                queuelin.put(?implq, v)
                await qtime == this.qtime
                queuespec.put(?specq, v)
        else:
            let v = queuelin.get(?implq):
                await qtime == this.qtime
                let w = queuespec.get(?specq):
                assert v == w
    atomic:
        qtime += 1

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

Figure 11.5: [code/qtestconc.hny] Concurrent queue test.
appear to happen in a total order. Moreover, we want to make sure that if some put or get operation $o_1$ finished before another operation $o_2$ started, then $o_1$ should appear to happen before $o_2$ in the total order. If these two conditions are met, then we say that the concurrent queue implementation is linearizable.

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

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

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

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

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

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

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

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

const N = 3

sequential putcount

testq = queue.Queue()
putcount = 0

def putter(v):
    queue.put(testq, v)
    atomic:
        putcount += 1

def main():
    await putcount == N
    let gotten = {}:
        while gotten != {0..N-1}:
            let v = queue.get(testq):
                assert v not in gotten
            gotten |= {v}
            let v = queue.get(testq):
                assert v == None

    for i in {0..N-1}:
        spawn putter(i)
    spawn main()

Figure 11.6: Concurrent put().
import queue

const N = 3

sequential gotten

testq = queue.Queue()

for i in {0..N–1}:
    queue.put(?testq, i)

gotten = {}

def getter():
    let v = queue.get(?testq):
        atomic:
            assert v not in gotten
            assert v != None
            gotten |= {v}

def main():
    await gotten == {0..N–1}
    let v = queue.get(?testq):
        assert v == None

    for i in {0..N–1}:
        spawn getter()
    spawn main()
import queue

const N = 3

sequential putcount, getcount

testq = queue.Queue()
gotten = {}
putcount = getcount = 0

def putter(v):
    queue.put(?testq, v)
    atomic:
        putcount += 1

def getter():
    let v = queue.get(?testq):
    atomic:
        assert v not in gotten
        if v != None:
            gotten |= {v}
        getcount += 1

def main():
    await (getcount == N) and (putcount == N)
    while gotten != {0..N–1}:
        let v = queue.get(?testq):
        assert v not in gotten
        gotten |= {v}
        let v = queue.get(?testq):
        assert v == None

    for i in {0..N–1}:
        spawn putter(i)
        spawn getter()
    spawn main()

Figure 11.8: [code/qtest3.hny] Mix of concurrent get() and put() operations.
```python
import queue
q1 = q2 = queue.Queue()
queue.put(q1, 1)
queue.put(q2, 2)

def getter(q, v):
    let x = queue.get(q):
    assert x == v

spawn getter(q1, 1)
spawn getter(q2, 2)
```

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

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

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

**Exercises**

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

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

11.3 Check if the extended queue implementation of Exercise 10.4 is linearizable.
```python
def seq_test():
    pass

def conc_test1():
    pass

def conc_test2():
    pass

def conc_test3():
    pass

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

Figure 11.10: [code/multitest.lny] Running a sequential test followed by several concurrent tests.
Chapter 12

Debugging

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

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

Let us run the code through some of the test programs in the last chapter. Harmony does not detect any issues with the sequential test in Figure 11.3. (Run this using the -m flag like this: `harmony -m queue=queuebug code/qtestseq.hny`) The concurrent put test of Figure 11.6 as well as the concurrent get test of Figure 11.7 also find no problems. However, when we run the new queue code through the test in Figure 11.8, Harmony finds a problem.

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

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

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

def Queue():
    result = { .next: None, .value: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .next: None, .value: v, .lock: Lock() }):
        while q != None:
            acquire(?q→lock)
            let n = q→next:
                if n == None:
                    q→next = node
                    release(?q→lock)
            q = n

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

Figure 12.1: [code/queuebug.hny] A buggy queue implementation.
Figure 12.2: Harmony output of running Figure 11.8 against Figure 12.1.
Next putter(1) (thread T2) takes a turn. It gets to line 10 in Figure 12.1 where it’s trying to acquire $q \rightarrow \text{lock}$. If we look at the bottom right, we see that $q$ points to $pool[0]$, the node that has value 0 in it. Again, so far so good.

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

To fix the code without changing the data structure, we can use hand-over-hand locking (Chapter 10). Figure 12.3 shows an implementation that uses hand-over-hand locking both for put() and for get(). It passes all tests.
from sync import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .next: None, .value: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .next: None, .value: v, .lock: Lock() }):
        acquire(?q→lock)
        let n = q→next:
            while n != None:
                acquire(?n→lock)
                release(?q→lock)
                q = n
                n = n→next
            q→next = node
        release(?q→lock)

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

Figure 12.3: [code/queuefix.hny](code/queuefix.hny) Queue implementation with hand-over-hand locking.
Chapter 13

Conditional Waiting

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

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

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

13.1 Reader/Writer Locks

Locks are useful when accessing a shared data structure. By preventing more than one thread from accessing the data structure at the same time, conflicting accesses are avoided. However, not all concurrent accesses conflict, and opportunities for concurrency may be lost, hurting performance. One important case is when multiple threads are simply reading the data structure. In many applications, reads are the majority of all accesses, and read operations do not conflict with one another. Allowing reads to proceed concurrently can significantly improve performance.

What we want is a special kind of lock that allows either (i) one writer or (ii) one or more readers to acquire the lock. This is called a reader/writer lock [CHP71]. A reader/writer lock is an object whose abstract (and opaque) state contains two integer counters:
import RW

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            RW.read_acquire(rw)
            @rcs: assert atLabel(wcs) == ()
            RW.read_release(rw)
        else:
            # .write
            RW.write_acquire(rw)
            @wcs: assert (atLabel(wcs) == { (thread, (): 1 }) and
                       (atLabel(rcs) == ()))
            RW.write_release(rw)

    for i in {1..3}:
        spawn thread()

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

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

satisfying the following invariant:

\[(nreaders \geq 0 \land nwriters = 0) \lor (nreaders = 0 \land 0 \leq nwriters \leq 1)\]

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

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

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

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

The producer/consumer pattern is common. Threads may be arranged in pipelines, where each upstream thread is a producer and each downstream thread is a consumer. Or threads may be arranged in a manager/worker pattern, with a manager producing jobs and workers consuming and executing them in parallel. Or, in the client/server model, some thread may act as a server that clients can send requests to and receive responses from. In that case, there is a bounded buffer for each client/server pair. Clients produce requests and consume responses, while the server consumes requests and produces responses.
Chapter 14

Split Binary Semaphores

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

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

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

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

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

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

Note that the following invariants hold:

- At any time, at most one gate is open;
- If some gate is open, then no thread is in the critical section. Equivalently, if some thread is in the critical section, all gates are closed;
- At any time, at most one thread is in the critical section.
```python
from synch import BinSema, acquire, release

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

def release_one(rw):
    if (rw->nwriters == 0) and (rw->r_gate.count > 0):
        release(rw->r_gate.sema)
    elif ((rw->nreaders + rw->nwriters) == 0) and (rw->w_gate.count > 0):
        release(rw->w_gate.sema)
    else:
        release(rw->mutex)

def read_acquire(rw):
    acquire(rw->mutex)
    if rw->nwriters > 0:
        rw->r_gate.count += 1; release_one(rw)
        acquire(rw->r_gate.sema); rw->r_gate.count -= 1
    rw->nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(rw->mutex); rw->nreaders -= 1; release_one(rw)

def write_acquire(rw):
    acquire(rw->mutex)
    if (rw->nreaders + rw->nwriters) > 0:
        rw->w_gate.count += 1; release_one(rw)
        acquire(rw->w_gate.sema); rw->w_gate.count -= 1
    rw->nwriters += 1
    release_one(rw)

def write_release(rw):
    acquire(rw->mutex); rw->nwriters -= 1; release_one(rw)
```

Figure 14.1: [code/RWsbs.hny](#) Reader/Writer Lock using Split Binary Semaphores.
The main gate is implemented by a binary semaphore, initialized in the released state (signifying that the gate is open). The waiting gates each consist of a pair: a counter that counts how many threads are waiting behind the gate and a binary semaphore initialized in the acquired state (signifying that the gate is closed).

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

- \( n_{\text{readers}} \): the number of readers in the critical section;
- \( n_{\text{writers}} \): the number of writers in the critical section;
- \( \text{mutex} \): the main gate binary semaphore;
- \( r_{\text{gate}} \): the waiting gate used by readers, consisting of a binary semaphore and the number of readers waiting to enter;
- \( w_{\text{gate}} \): the waiting gate used by writers, similar to the readers’ gate.

Each of the \texttt{read.acquire}, \texttt{read.release}, \texttt{write.acquire}, and \texttt{write.release} methods must maintain this state. First they have to acquire the \texttt{mutex} (i.e., enter the main gate). After acquiring the \texttt{mutex}, \texttt{read.acquire} and \texttt{write.acquire} each must check to see if it has to wait. If so, it increments the count associated with its respective gate, opens a gate (using method \texttt{release.one}), and then blocks until its waiting gate opens up.

\texttt{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, \texttt{release.one()} first checks the readers’ gate and then the writers’ gate, but the other way around would have worked as well. If neither waiting gate qualifies, then \texttt{release.one()} has to open the main gate (i.e., release \texttt{mutex}).

Let us examine \texttt{read.acquire} more carefully. First the method acquires the \texttt{mutex}. Then, in the case that the thread finds that there is a writer in the critical section (\( n_{\text{writers}} > 0 \)), it increments the counter associated with the readers’ gate, leaves the critical section (\texttt{release.one}), and then tries to acquire the semaphore associated with the waiting gate. This will cause it to block until some thread opens that gate.

Now consider the case where there is a writer in the critical section and there are two readers waiting. Let us see what happens when the writer calls \texttt{write.release}:

1. After acquiring \texttt{mutex}, the writer decrements \( n_{\text{writers}} \), which must be 1 at this time, and thus becomes 0.

2. It then calls \texttt{release.one().} \texttt{release.one()} finds that there are no writers in the critical section and there are two readers waiting. It therefore releases not \texttt{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 \( n_{\text{readers}} \) (from 0 to 1). The reader finally calls \texttt{release.one().}
4. Again, release_one() finds that there are no writers and that there are readers waiting, so again it releases the readers’ semaphore.

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

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

Exercises

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

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

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

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

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

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

14.7 Implement a thread-safe GPU allocator by modifying Figure 14.2. There are N GPUs identified by the numbers 1 through N. Method gpuAlloc() returns the identifier of an available GPU, blocking if there is currently no available GPU. Method gpuRelease(gpu) releases the given GPU. It never needs to block.

14.8 With reader/writer locks, concurrency can be improved if a thread downgrades its write lock to a read lock when its done writing but not done reading. Add a downgrade method to the code in Figure 14.1. (Similarly, you may want to try to implement an upgrade of a read lock to a write lock. Why is this problematic?)

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

¹A bound lock is a restricted version of a counting semaphore.
const N = 10

availGPUs = {1..N}

def gpuAlloc():
    result = choose(availGPUs)
    availGPUs -= { result }

def gpuRelease(gpu):
    availGPUs |= { gpu }

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

14.10 Extend the solution to Exercise 14.9 by implementing the requirement that at most \( n \) cars are allowed on the bridge. Add \( n \) as an argument to `OLBlock`. 
Chapter 15

Starvation

A property is a set of traces. If a program has a certain property, that means that the traces that that program allows are a subset of the traces in the property. So far, we have pursued two properties: mutual exclusion and progress. The former is an example of a safety property—it prevents something “bad” from happening, like a reader and writer thread both acquiring a reader/writer lock. The progress property is an example of a liveness property—guaranteeing that something good eventually happens. Informally (and inexacty), progress states that if no threads are in the critical section, then some thread that wants to enter can.

Progress is a weak form of liveness. It says that some thread can enter, but it does not prevent a scenario such as the following. There are three threads repeatedly trying to enter a critical section using a spinlock. Two of the threads successfully keep entering, alternating, but the third thread never gets a turn. This is an example of starvation. With a spinlock, this scenario could even happen with two threads. Initially both threads try to acquire the spinlock. One of the threads is successful and enters. After the thread leaves, it immediately tries to re-enter. This state is identical to the initial state, and there is nothing that prevents the same thread from acquiring the lock yet again. (It is worth noting that Peterson’s Algorithm (Chapter 6) does not suffer from starvation, thanks to the turn variable that alternates between 0 and 1 when two threads are contending for the critical section.)

While spinlocks suffer from starvation, it is a uniform random process and each thread has an equal chance of entering the critical section. Thus the probability of starvation is exponentially vanishing. We shall call such a solution fair (although it does not quite match the usual formal nor vernacular concepts of fairness).

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

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

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

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

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

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

Figure 15.1: [code/RWfair.hny] Reader/Writer Lock SBS implementation addressing fairness.
SBSs allow much control over which type of thread runs next and is therefore a good starting point for developing fair synchronization algorithms. Figure 15.1 is based on Figure 14.1, but there are two important differences:

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

2. Instead of a one-size-fits-all release_one method, each method has a custom way of selecting which gate to open. In particular, read_release prefers the write gate, while write_release prefers the read gate.

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

**Exercises**

15.1 Write a fair solution to the one-lane bridge problem of Exercise 14.9.
Chapter 16
Monitors

Tony Hoare, who came up with the concept of split binary semaphores, devised an abstraction of the concept in a programming language paradigm called monitors [Hoa74]. (A similar construct was independently invented by Per Brinch Hansen [BH73].) A monitor is a special version of an object-oriented class, comprising a set of variables and methods that operate on those variables. A monitor also has special variables called condition variables, one per waiting condition. There are two operations on condition variables: wait and signal.

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

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

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

def Monitor():
    result = synch.Lock()

def enter(mon):
    synch.acquire(mon)

def exit(mon):
    synch.release(mon)

def Condition():
    result = { .sema: synch.BinSema(True), .count: 0 }

def wait(cond, mon):
    cond→count += 1
    exit(mon)
    synch.acquire(?cond→sema)
    cond→count -= 1

def signal(cond, mon):
    if cond→count > 0:
        synch.release(?cond→sema)
        enter(mon)

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

def BB(size):
    result = {
        .mon: hoare.Monitor(),
        .prod: hoare.Condition(), .cons: hoare.Condition(),
        .buf: { x(): for x in {1..size} },
        .head: 1, .tail: 1,
        .count: 0, .size: size
    }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)

def get(bb):
    hoare.enter(?bb→mon)
    if bb→count == 0:
        hoare.wait(?bb→cons, ?bb→mon)
    result = bb→buf[bb→head]
    bb→head = (bb→head % bb→size) + 1
    bb→count -= 1
    hoare.signal(?bb→prod, ?bb→mon)
    hoare.exit(?bb→mon)

Figure 16.2: [code/BBhoare.hny] Bounded Buffer implemented using a Hoare monitor.
In the late 70s, researchers at Xerox PARC, where among others the desktop and Ethernet were invented, developed a new programming language called Mesa [LR80]. Mesa introduced various important concepts to programming languages, including software exceptions and incremental compilation. Mesa also incorporated a version of monitors. However, there are some subtle but important differences with Hoare monitors that make Mesa monitors quite unlike split binary semaphores and mostly easier to use in practice.

As in Hoare monitors, there is a hidden mutex associated with each Mesa monitor, and the mutex must be acquired upon entry to a method and released upon exit. Mesa monitors also have condition variables that a thread can wait on. Like in Hoare monitors, the *wait* operation releases the mutex. The most important difference is in what *signal* does. To make the distinction more clear, we shall call the corresponding Mesa operation *notify* rather than *signal*. When a thread \( p \) invokes *notify*, it does not immediately pass control to a thread that is waiting on the corresponding condition (if there is such a thread). Instead, \( p \) remains in the critical section until it leaves the monitor explicitly (by calling either *release* or *wait*). At that point, any thread that was notified will have a chance to enter the critical section, but they compete with other threads trying to enter the critical section.

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

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

The so-called “Mesa monitor semantics” or “Mesa condition variable semantics” have become more popular than Hoare monitor semantics and have been adopted by all major programming languages. That said, few programming languages provide full syntactical support for monitors, instead opting to support monitor semantics through library calls. In Java, each object has a hidden lock and a hidden condition variable associated with it. Methods declared with the *synchronized* keyword automatically obtain the lock. Java objects also support *wait*, *notify*, and *notifyAll*. In addition, Java supports explicit allocations of locks and condition variables. In Python, locks and condition variables must be explicitly declared. The *with* statement makes it easy to acquire and release a lock for a section of code. In C and C++, support for locks and condition variables is entirely through libraries.

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

def wait(c, lk):
    let blocked, cnt, ctx = True, 0, get_context():
        atomic:
            cnt = bag.count(!c, ctx)
            !c = bag.add(!c, ctx)
            !lk = False
        while blocked:
            atomic:
                if (not !lk) and (bag.count(!c, ctx) <= cnt):
                    !lk = True
                    blocked = False

def notify(c):
    atomic:
        if !c != bag.empty():
            !c = bag.remove(!c, bag.bchoose(!c))

def notifyAll(c):
    !c = bag.empty()
from synch import *

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

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

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

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

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

Figure 16.4: [code/RWcv.lny] Reader/Writer Lock using Mesa-style condition variables.
wait adds the context of the thread to the bag. This increments the number of threads in the bag with the same context. wait then loops until that count is restored to the value that it had upon entry to wait. notify removes an arbitrary context from the bag, allowing one of the threads with that context to resume and re-acquire the lock associated with the monitor. notifyAll empties out the entire bag, allowing all threads in the bag to resume.

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

Note that wait is always invoked within a while loop that checks for the condition that the thread is waiting for. It is imperative that there is always a while loop around any invocation of wait containing the negation of the condition that the thread is waiting for. Many implementation of Mesa condition variables depend on this, and optimized implementations of condition variables often allow so-called “spurious wakeups,” where wait may sometimes return even if the condition variable has not been notified. As a rule of thumb, one should always be able to replace wait by unlock followed by lock. This turns the solution into a busy-waiting one, inefficient but still correct.

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

write_release executes notifyAll(?r_cond) as well as notify(?w_cond). Because we do not keep track of the number of waiting readers or writers, we have to conservatively assume that all waiting readers can enter, or, alternatively, up to one waiting writer can enter. So write_release wakes up all potential candidates. There are two things to note here. First, unlike split binary semaphores or Hoare monitors, where multiple waiting readers would have to be signaled one at a time in a baton-passing fashion (see Figure 14.1), with Mesa monitors all readers are awakened in one fell swoop using notifyAll. Second, both readers and writers are awakened—this is ok because both execute wait within a while loop, re-checking the condition that they are waiting for. So, if both type of threads are waiting, either all the readers get to enter next or one of the writers gets to enter next. (If you want to prevent waking up both readers and a writer, then you can keep track of how many threads are waiting in the code.)

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

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

Exercises

16.1 Implement a solution to the bounded buffer problem using Mesa condition variables.
16.2 Implement a “try lock” module using Mesa condition variables (see also Exercise 9.3). It should have the following API:

1. \( tl = \text{TryLock}() \) # create a try lock
2. \( \text{acquire}(\text{?}tl) \) # acquire a try lock
3. \( \text{tryAcquire}(\text{?}tl) \) # attempt to acquire a try lock
4. \( \text{release}(\text{?}tl) \) # release a try lock

\text{tryAcquire} should not wait. Instead it should return \textbf{True} if the lock was successfully acquired and \textbf{False} if the lock was not available.

16.3 Write a new version of the GPU allocator in Exercise 14.7 using Mesa condition variables. In this version, a thread is allowed to allocate a set of GPUs and release a set of GPUs that it has allocated. Method \( \text{gpuAllocSet}(n) \) should block until \( n \) GPUs are available, but it should grant them as soon as they are available. It returns a set of \( n \) GPU identifiers. Method \( \text{gpuReleaseSet}(s) \) takes a set of GPU identifiers as argument. A thread does not have to return all the GPUs it allocated at once. (You may want to try implementing this with Split Binary Semaphores. It is not as easy.)

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

16.5 Bonus problem: Figure 16.5 shows an iterative implementation of the Qsort algorithm, and Figure 16.6 an accompanying test program. The array to be sorted is stored in shared variable \texttt{textqs.arr}. Another shared variable, \texttt{testqs.todo}, contains the ranges of the array that need to be sorted (initially the entire array). Re-using as much of this code as you can, implement a parallel version of this. You should not have to change the methods \texttt{swap}, \texttt{partition}, or \texttt{sortrange} for this. Create \texttt{NWORKERS} “worker threads” that should replace the \texttt{qsort} code. Each worker loops until \texttt{todo} is empty and sorts the ranges that it finds until then. The main thread needs to wait until all workers are done.
```python
def Qsort(arr):
    result = { .arr: arr, .todo: { (0, len(arr) - 1) } }

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

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

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

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

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

```
Chapter 17

Deadlock

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

Imagine five 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 leave the table when they are not using any forks.

Figure 17.1 implements the dining philosophers in Harmony, using a thread for each philosopher and a lock for each fork. If you run it, Harmony complains that the execution may not be able to terminate, with all five threads being blocked trying to acquire the lock.

- Do you see what the problem is?
- Does it depend on N, the number of philosophers?
- Does it matter in what order the philosophers lay down their forks?

There are four conditions that must hold for deadlock to occur [CES71]:

1. Mutual Exclusion: each resource can only be used by one thread at a time:

2. Hold and Wait: each thread holds resources it already allocated while it waits for other resources that it needs;

3. No Preemption: resources cannot be forcibly taken away from threads that allocated them;

4. Circular Wait: there exists a directed circular chain of threads, each waiting to allocate a resource held by the next.

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

```
const N = 5

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

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

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

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

To implement a **No Hold and Wait** solution, a philosopher would need a way to lock both the left and right forks at the same time. Locks do not have such an ability, and neither do semaphores. so we re-implement the Dining Philosophers using condition variables that allow one to wait for arbitrary application-specific conditions. **Figure 17.2** demonstrates how this might be done. We use a single mutex for the diners, and, for each fork, a boolean and a condition variable. The boolean indicates if the fork has been taken. Each diner waits if either the left or right fork is already taken. But which condition variable to wait on? The code demonstrates an important technique to use when waiting for multiple conditions. The condition in the **while** statement is the negation of the condition that the diner is waiting and consists of two disjuncts. Within the **while** statement, there is an **if** statement for each disjunct. The code waits for either or both forks if necessary. After that, it goes back to the top of the **while** loop.

A common mistake is to write the following code instead:
import synch

const N = 5

mutex = synch.Lock()
forks = [False] * N
conds = [synch.Condition()] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think

for i in {0..N–1}:
    spawn diner(i)

Figure 17.2: [code/DinersCV.hny] Dining Philosophers that grab both forks at the same time.
while forks[left]:
    synch.wait(conds[left], mutex)
while forks[right]:
    synch.wait(conds[right], mutex)

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

The Preemption approach suggested by Havender is to allow threads to back out. While this
could be done, this invariably leads to a busy waiting solution where a thread keeps obtaining locks
and releasing them again until it finally is able to get all of them.

The No Circular Waiting approach is to prevent a cycle from forming, with each thread waiting
for the next thread on the cycle. We can do this by establishing an ordering among the resources (in
this case the forks) and, when needing more than one resource, always acquiring them in order. In
the case of the philosophers, they could prevent deadlock by always picking up the lower numbered
fork before the higher numbered fork, like so:

if left < right:
    synch.acquire(forks[left])
    synch.acquire(forks[right])
else:
    synch.acquire(forks[right])
    synch.acquire(forks[left])

or like so:

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

This completes all the Havender methods. There is, however, another approach, which is some-
times called deadlock avoidance instead of deadlock prevention. In the case of the Dining Philoso-
phers, we want to avoid the situation where each diner picks up a fork. If we can prevent more than
four diners from starting to eat at the same time, then we can avoid the conditions for deadlock from
ever happening. Figure 17.3 demonstrates this concept. It uses a counting semaphore to restrict
the number of diners at any time to four. A counting semaphore is like a binary semaphore, but
can be acquired a given number of times. It is supported by the synch module. The P or “procure”
operation acquires a counting semaphore. That is, it tries to decrement the semaphore, blocking
while the semaphore has a value of 0. The V or “vacate” operation increments the semaphore.

This avoidance technique can be generalized using something called the Banker’s Algo-
rithm [Dij64], but it is outside the scope of this book. The problem with these kinds of schemes
```python
from synch import *

const N = 5

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

def diner(which):
    left, right = (which, (which + 1) % N):
    while choose({False, True}):
        P(sema)  # procure counting semaphore
        acquire(forks[left])
        acquire(forks[right])
        # dine
        release(forks[left])
        release(forks[right])
        V(sema)  # vacate counting semaphore
        # think
    for i in {0..N–1}:
        spawn diner(i)
```

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

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

**Exercises**

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

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

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

17.4 Add an invariant that checks that the total of the balances never changes. Note that the invariant only holds if none of the locks are held.
```python
from synch import Lock, acquire, release

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

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

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

def thread():
    let a1 = choose({0..N_ACCOUNTS–1})
    let a2 = choose({0..N_ACCOUNTS–1} – { a1 }):
        transfer(a1, a2, choose({1..MAX_BALANCE}))
    for i in {1..N_THREADS}:
        spawn thread()
```

Figure 17.4: [code/bank.hny] Bank accounts.
Chapter 18

Actors and Message Passing

Some programming languages favor a different way of implementing synchronization using so-called actors [HBS73, Agh86]. Actors are threads that have only private memory and communicate through message passing. See Figure 18.1 for an illustration. Given that there is no shared memory in the actor model (other than the message queues, which have built-in synchronization), there is no need for critical sections. Instead, some sequential thread owns a particular piece of data and other threads access it by sending request messages to the thread and optionally waiting for response messages. Each thread handles one message at a time, serializing all access to the data it owns. As message queues are FIFO (First-In-First-Out), starvation is prevented.

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

![Figure 18.1: Depiction of three actors. The producer does not receive messages.](image)

104
```python
import synch

const NCLIENTS = 3

server_queue = synch.Queue()

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

spawn eternal server()

sequential done
done = [False,] * NCLIENTS

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

alice_queue = synch.Queue()
spawn client(alice_queue)

bob_queue = synch.Queue()
spawn client(bob_queue)

charlie_queue = synch.Queue()
spawn client(charlie_queue)
```

Figure 18.2: [code/counter.hny] An illustration of the actor approach.
Queue() returns a new message queue;
put(q, item) adds item to the queue pointed to by q;
get(q) waits for and returns an item on the queue pointed to by q.

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

Figure 18.2 illustrates the actor approach. There are three client threads that each want to be assigned a unique identifier from the set \{0, 1, 2\}. Normally one would use a shared 0-initialized counter and a lock. Each client would acquire the lock, get the value of the counter and increment it, and release the lock. Instead, in the actor approach the counter is managed by a separate server thread. 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. The code is tested using the done array.

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

Exercises

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

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

Configure two versions of the pipeline, one that first checks if a number is prime and then if it is one less than a power of two, the other in the opposite order. Which do you think is better?
Chapter 19

Barrier Synchronization

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

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

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

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

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

Notice there are two different waiting conditions:

1. waiting for the car to empty out;
2. waiting for the car to fill up.
from synch import Queue, put, get

const NTHREADS = 3
const NROUNDS = 4

sequential round
round = [0,] * NTHREADS

q = [Queue(),] * NTHREADS

def thread(self):
    for r in {1..NROUNDS}:
        for i in {0..NTHREADS–1} where i != self:
            put(?q[i], None)
        for i in {0..NTHREADS–1} where i != self:
            get(?q[self])
        round[self] += 1
    assert (max(round) – min(round)) <= 1

for i in {0..NTHREADS–1}:
    spawn thread(i)

Figure 19.1: [code/qbarrier.hny] Barrier synchronization with queues.
from synch import *

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

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

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

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

cost NROUNDS = 3
cost NTHREADS = 3

barr = barrier.Barrier(NTHREADS)

sequential round
round = [None,] * NTHREADS

def thread(self):
    for r in {0..NROUNDS–1}:
        barrier.enter(?barr)
        round[self] = r
        assert { x for x in round where x != None } == { r }
        round[self] = None
        barrier.exit(?barr)

    for i in {0..NTHREADS–1}:
        spawn thread(i)
```

Figure 19.3: [code/barriertest.hny](code/barriertest.hny) Test program for Figure 19.2.
But this poses a complication. Suppose, for example, that there are two seats in the car, and there is one person in the car. Does that mean that the car is not yet full, or not yet empty? We have to distinguish those situations. To this end, it is useful to think of the car as going through $2 \cdot n$ stages:

- Stage 0: the car is empty;
- Stage 1...$n - 1$: the car is filling up;
- Stage $n$: the car is full;
- Stage $n + 1$..$2n - 1$: the car is emptying out.

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

**Exercises**

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

19.2 Imagine a pool hall with $N$ tables. A table is full from the time there are two players until both players have left. When someone arrives, they can join a table that is not full, preferably one that has a player ready to start playing. Implement a simulation of such a pool hall.
Chapter 20

Interrupts

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

Harmony allows modeling interrupts using the \texttt{trap} statement:

\begin{verbatim}
trap handler argument
\end{verbatim}

invokes \texttt{handler argument} at some later, unspecified time. Thus you can think of \texttt{trap} as setting a timer. Only one of these asynchronous events can be outstanding at a time; a new call to \texttt{trap} overwrites any outstanding one. Figure 20.1 gives an example of how \texttt{trap} might be used. Here, the \texttt{main()} thread loops until the interrupt has occurred and the \texttt{done} flag has been set.

But now consider Figure 20.2. The difference with Figure 20.1 is that both the \texttt{main()} and \texttt{handler()} methods increment \texttt{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 \texttt{main()} reads \texttt{count} (and thus still has value 0) but before \texttt{main()} writes the updated value 1, then the interrupt handler will also read value 0 and write value 1. We say that the code in Figure 20.2 is not interrupt-safe (as opposed to not being thread-safe).

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

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

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

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

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

Figure 20.2: [code/trap2.hny] A race condition with interrupts.
from synch import Lock, acquire, release

sequential done
countlock = Lock()
count = 0
done = False
def handler():
    acquire(\texttt{countlock})
    count += 1
    release(\texttt{countlock})
    done = True
def main():
    trap handler()
    acquire(\texttt{countlock})
    count += 1
    release(\texttt{countlock})
    await done
    assert count == 2
spawn main()

Figure 20.3: [code/trap3.hny] Locks do not work with interrupts.
def handler():
    count += 1
    done = True

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

spawn main()

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

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

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

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

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

(Another option is to use synchronization techniques that do not use locks. See Chapter 21 for more information.)

There is another important rule to keep in mind. Just like locks should never be held for long, interrupts should never be disabled for long. With locks the issue is to maximize concurrent performance. For interrupts the issue is fast response to asynchronous events. Because interrupts may be disabled only briefly, interrupt handlers must run quickly and cannot wait for other events. It is ok to invoke non-blocking synchronization calls such as notify, but calls such as acquire and
def increment():
    let prior = setintlevel(True):
    count += 1
    setintlevel(prior)

def handler():
    increment()
    done = True

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

spawn main()

Figure 20.5: [code/trap5.hny] Example of an interrupt-safe method.
from synch import Lock, acquire, release

sequential done

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

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

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

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

spawn thread(0)
spawn thread(1)

Figure 20.6: [code/trap6.hny] Code that is both interrupt-safe and thread-safe.
wait should only be used if it is certain that they will not block for long. Informally, interrupt handlers can be producers but not consumers of synchronization events.

Exercises

20.1 The put method you implemented in Exercise 16.1 cannot be used in interrupt handlers for two reasons: (1) it is not interrupt-safe, and (2) it may block for a long time if the buffer is full. Yet, it would be useful if, say, a keyboard interrupt handler could place an event on a shared queue. Implement a new method \texttt{i.put(item)} that does not block. Instead, it should return \texttt{False} if the buffer is full and \texttt{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. \( \text{items} \) is an unbounded array with each entry initialized to \( \text{None} \). \( \text{back} \) is an index into the array and points to the next slot where a new value is inserted. The code uses two atomic operations:

\[ \text{inc} (p) \]: atomically increments \( p \) and returns the old value;
\[ \text{exch} (p) \]: sets \( p \) to \( \text{None} \) and returns the old value.

Method \text{produce}(\text{item})\ uses \text{inc}(\text{?back})\ to allocate the next available slot in the \text{items} array. It stores the item as a singleton tuple. Method \text{consume}()\ repeatedly scans the array, up to the \text{back} index, trying to find an item to return. To check an entry, it uses \text{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 \text{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 \text{inc} and \text{exch} atomic operations are not universally available on existing processors.
const MAX_ITEMS = 3

sequential back, items
back = 0
items = [None] * MAX_ITEMS

def inc(pcnt):
    atomic:
    result = !pcnt
    !pcnt += 1

def exch(pv):
    atomic:
    result = !pv
    !pv = None

def produce(item):
    items[inc(?back)] = item

def consume():
    result = None
    while result == None:
        let range, i = back, 0:
        while (i < range) and (result == None):
            result = exch(?items[i])
            i += 1

for i in {1..MAX_ITEMS}:
    spawn produce(i)
for i in {1..choose({0..MAX_ITEMS})}:
    spawn consume()

Figure 21.1: [code/hw.hny] Non-blocking queue.
However, in the literature you can find examples of practical non-blocking (aka *wait-free*) synchronization algorithms.

**Exercises**

21.1 A *seqlock* consists of a lock and a version number. An update operation acquires the lock, increments the version number, makes the changes to the data structure, and then releases the lock. A read-only operation does not use the lock. Instead, it retrieves the version number, reads the data structure, and then checks if the version number has changed. If so, the read-only operation is retried. Use a seqlock to implement a bank much like Exercise 17.2, with one seqlock for the entire bank (i.e., no locks on individual accounts). Method *transfer* is an update operation; method *total* is a read-only operation. Explain how a seqlock can lead to starvation.
Chapter 22

Alternating Bit Protocol

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

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

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

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

We can model each channel as a variable that either contains a network message or nothing (we use $\texttt{}$ in the model). Let $s_{chan}$ be the channel from Alice to Bob and $r_{chan}$ the channel from Bob to Alice. $\texttt{net\_send(pchan, m, reliable)}$ models sending a message $m$ to !pchan, where pchan is either ?s_{chan} or ?r_{chan}. The method places either $m$ (to model a successful send) or $\texttt{()}$ (to
```python
sequential s_chan, r_chan

s_chan = r_chan = ()
s_seq = r_seq = 0

def net_send(pchan, m, reliable):
    pchan = m if (reliable or choose({False, True})) else ()

def net_recv(pchan):
    result = !pchan

def app_send(payload):
    s_seq = 1 - s_seq
    let m, blocked = {seq: s_seq, payload: payload}, True:
        while blocked:
            net_send(s_chan, m, False)
            let response = net_recv(r_chan):
                blocked = (response == ()) or (response.ack != s_seq)

def app_recv(reliable):
    r_seq = 1 - r_seq
    let blocked = True:
        while blocked:
            let m = net_recv(s_chan):
                if m != ():
                    net_send(r_chan, {ack: m.seq}, reliable)
                if m.seq == r_seq:
                    result = m.payload
                    blocked = False
```

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

const NMSGS = 5

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

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

spawn sender()
spawn receiver()

Figure 22.2: test/abptest.hny Test code for alternating bit protocol.

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

Method app_send(m) retransmits m until an acknowledgment is received. Method app_recv(reliable) returns the next successfully received message. Figure 22.2 shows how the methods may be used to send and receive a stream of NMSGS messages reliably. It has to be bounded, because model checking requires a finite model.

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

Exercises

22.1 Chapter 18 explored the client/server model. It is popular in distributed systems as well. Develop a protocol for a single client and server using the same network model as for the ABP protocol. Hint: the response to a request can contain the same sequence number as the request.

22.2 Generalize the solution in the previous exercise to multiple clients. Each client is uniquely identified. You may either use separate channel pairs for each client, or solve the problem using a single pair of channels.
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 forward that 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. It 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 select statement. This statement takes the form select 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 select statement are as follows:

- it waits until s is non-empty;
- it is executed atomically;
const NIDS = 5       # number of identifiers

network = {}        # the network is a set of messages
leader = 0          # used for checking correctness

def send(m):
    atomic: network |= { m }

def receive(self):
    result = { (id, found) for (dst, id, found) in network where dst == self }

def processor(self, succ):
    send(succ, self, False)

    let working = True:
       while working:
           select (id, found) in receive(self):
               if id == self:
                   assert self == leader
                   send(succ, id, True)
               elif id > self:
                   assert self != leader
                   send(succ, id, found)
               if found:
                   working = False

let ids, nprocs, procs = { 1 .. NIDS }, choose({ 1 .. NIDS }), []:

for i in { 0 .. nprocs - 1 }:
    let next = choose(ids):
        ids -= { next }
        procs += [ next, ]
        if next > leader:
            leader = next

for i in { 0 .. nprocs - 1 }:
    spawn processor(procs[i], procs[(i + 1) % nprocs])

Figure 23.1: [code/leader.hny] A leader election protocol on a ring.
• \( v \) 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 select 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 atomic 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.) 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 at least the following two properties:

- *all-or-nothing*: a transaction should either complete, or it should be a no-op. It 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 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 17.2) 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.
```python
network = {}

def send(m):
    atomic: network |= { m }

def bank(self, balance):
    let status, received = (), {}:
    while True:
        select req in { req for req in network - received where req.dst == self }:
            received |= { req }
        if req.request == .withdraw:
            if (status != ()) or (req.amount > balance):
                send({ .dst: req.src, .src: self, .response: .no })
            else:
                status = balance
                balance -= req.amount
                send({ .dst: req.src, .src: self, .response: .yes, .funds: balance })
        elif req.request == .deposit:
            if status != ():
                send({ .dst: req.src, .src: self, .response: .no })
            else:
                status = balance
                balance += req.amount
                send({ .dst: req.src, .src: self, .response: .yes, .funds: balance })
        elif req.request == .commit:
            assert status != ()
            status = ()
        else:
            assert (status != ()) and (req.request == .abort)
            balance, status = status, ()
```

Figure 24.1: [code/2pc.hny] Two Phase Commit protocol: code for banks.
import list
const NBANKS = 3
const NCOORDS = 2
const MAX_BALANCE = 1

def receive(self, sources):
    forme = { m for m in network where m.dst == self }:
    result = { forme } if { m.src for m in forme } == sources else {}

def transfer(self, b1, b2, amt):
    send({ .dst: b1, .src: self, .request: .withdraw, .amount: amt })
    send({ .dst: b2, .src: self, .request: .deposit, .amount: amt })
    select msgs in receive(self, { b1, b2 }):
        if all(m.response == .yes for m in msgs):
            possibly True
            for m in msgs where m.response == .yes:
                send({ .dst: m.src, .src: self, .request: .commit })
        else:
            for m in msgs where m.response == .yes:
                send({ .dst: m.src, .src: self, .request: .abort })

def check(self, total):
    allbanks = { (.bank, i) for i in { 1 .. NBANKS } }:
    for bank in allbanks:
        send({ .dst: bank, .src: self, .request: .withdraw, .amount: 0 })
    select msgs in receive(self, allbanks):
        possibly all(m.response == .yes for m in msgs)
    assert all(m.response == .yes for m in msgs) =>
    (list.sum(m.funds for m in msgs) == total)
    for m in msgs where m.response == .yes:
        send({ .dst: m.src, .src: self, .request: .abort })

    balances = { i:choose({ 0 .. MAX_BALANCE }) for i in { 1 .. NBANKS } }:
    for i in { 1 .. NBANKS }:
        spawn eternal bank(({.bank, i}, balances[i])
    for i in { 1 .. NCOORDS }:
        if choose({ .transfer, .check }) == .transfer:
            let b1 = choose(({.bank, j} for j in { 1 .. NBANKS }))
            let b2 = choose(({.bank, j} for j in { 1 .. NBANKS })) - { b1 }:
                spawn transfer(({coord, i}, b1, b2, 1)
        else:
            spawn check(({coord, i}, list.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 have the following dictionary format: `{.dst,.src,.request,.amount}`:

- **.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 a select 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. It returns the empty set if there 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**. It then waits for responses from each. If both responses are .yes, then it commits the transaction, otherwise it aborts the transaction.

131
Note the use of the `possibly` statement in this code. It is similar to `assert`, but where `assert` checks that the predicate holds in all executions, `possibly` checks that the predicate holds in at least one execution. The purpose is to check that the code doesn’t simply always abort—there exist executions where the code commits. In practice, hopefully most executions commit, but Harmony is not capable of probabilistic reasoning.

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 17.2 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 \[\text{[Lam78, Sch90]}\] 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. The trick then 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 \[\text{[vRS04]}\] is such a replication protocol. In Chain Replication, the replicas are organized in a linear chain. Each replica monitors its direct predecessor in the chain. If a replica has no predecessors, we call it the head. The head may change over time, as a head may crash and replaced by another, but at any point in time there is only one head (and only one tail, possibly the same replica). We model the state machine as a history: a sequence of operations. Only the head is allowed to introduce new updates. When it does so, it advertises the new update along with its position in the history. When the direct successor receives the next update it is expecting, it appends the update to its history and likewise advertises the new update. When the update is applied to the history of the tail replica (the replica that has no live successors), then the update is considered final.

So, when a replica fails, its successors 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 eventually detected. This is called the Fail-Stop failure model \[\text{[SS83]}\], which is distinct from the often more realistic Crash
const NREPLICAS = 4    # number of replicas  
const NOPS = 5        # number of operations  
network = {}        # the network is a set of messages  
final = None       # this is used to check correctness  

def send(m):  
    atomic: network |= { m }  

def receive(predecessor, index):  
    result = { payload for (id, payload) in network where (id == predecessor)  
                and ((payload == .crash) or (payload[0] == index)) }  

def replica(self, immortal):  
    let history, predecessors = [], { 0 .. self – 1 }:
        while choose({ immortal, True }) and (len(history) < NOPS):
            if predecessors == {}:
                # I’m the head
                send(self, (len(history), self))
                history += [self]
            else:
                # I’m not the head (yet)
                select payload in receive(max(predecessors), len(history)):
                    if payload == .crash:
                        predecessors -= { max(predecessors) }
                    else:
                        send(self, payload)
                        history += [payload[1],]
                if len(history) == NOPS:   # successful completion
                    atomic:
                        assert (final == None) or (final == history)
                        possibly 0 in history, 1 in history, (NREPLICAS – 1) in history
                        final = history
                    else:
                        # replica crashed
                        send(self, .crash)
        let survivor = choose({ 0 .. NREPLICAS – 1 }):
            for i in { 0 .. NREPLICAS – 1 }:
                spawn replica(i, i == survivor)  

Figure 25.1: [code/chain.hny] Checking correctness of Chain Replication.
failure model where processes can crash but accurate detection is not available. We will consider that failure model in the upcoming chapters.

Figure 25.1 shows an implementation of chain replication. The code starts NREPLICAS replica threads numbered 0 through NREPLICAS − 1, with initially 0 being the head and NREPLICAS − 1 being the tail. At least one of the replicas must be immortal—the code selects one of them although none of the replicas is required to fail during execution. Of course, a replica does not know whether it is immortal or not in practice—it should just assume so. The immortality of one of the replicas is only used for modeling the system.

Each replica maintains its history and a set of its predecessors. It then loops until either it fails or it has applied all NOPS operations to its history. choose({immortal, True}) can only evaluate to False in case immortal is false; otherwise it will always evaluate to True. What it does within the loop depends on whether it is the head (has no alive predecessors) or not. 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.

Messages have the format (source, payload), where source is the identifier of the replica that generated the message and payload has information about the state of the replica. The payload can take one of two forms. The first is .crash, indicating that the source of the message has crashed. The code models a replica crashing by that replica broadcasting a .crash message on the network. The other form of payload is a tuple (index, operation), which means that the source placed the given operation at the given index in its history.

If a replica thinks it is the head, it adds an operation to its history. In practice this would be some operation that the head would have received from a client. Clients send their operations to the head, which adds them to its history in the order received. We are not modeling clients of the service. Instead, in the model, the head adds its identifier self to the history until the history has NOPS operations on it or until the head crashes, whichever comes first. The head also broadcasts each update, which are intended for its successor. Because of failures, it may be that at different times a replica has different successors, hence the broadcast.

Otherwise, the replica does not believe it is the head and awaits a message from its direct predecessor. If the payload is .crash, then the replica removes this predecessor from its set of predecessors. For example, if there are initially three replicas numbered 0, 1, and 2, and replica 1 crashes, then replica 2 removes 1 from its set of predecessors, making replica 0 its direct (and only) predecessor. If, instead, the payload is a tuple, then it must be the case that it contains the next operation to add to its history. The replica adds the operation to its history and broadcasts the update, again intended for its successor.

When the replica successfully adds all NOPS updates to its history without crashing, the code makes sure that any other such replica ends in the same state. Variable final is used to check the different histories against one another.
Chapter 26

Replicated Atomic Read/Write Register

A register is an object that you can read or write. In a distributed system, it can be useful to have registers that are shared. 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 \( r_1 \).

Unfortunately, however, the 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 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 11). Each server maintains not only the value of the object, but also the timestamp of its corresponding write operation.
import bag

const F = 1
const N = (2 * F) + 1

network = bag.empty()

def send(m):
    atomic:
        network = bag.add(network, m)

def server():
    let t, v, received = (0, None), None, {}:
        while True:
            select m in { m for m in keys network – received
                where m.type in {.read, .write} }:
                received |= { m }
                if (m.type == .write) and (m.value[0] > t):
                    t, v = m.value
                    send({ .type: .response, .dst: m.src, .value: (t, v) })

def receive(uid, phase):
    let msgs = { m:c for m:c in network
        where (m.type == .response) and (m.dst == (uid, phase)) }:
        result = bag.combinations(msgs, N – F)

def read(uid):
    send({ .type: .read, .src: (uid, 1) })
    select msgs in receive(uid, 1):
        let (t, v) = max(m.value for m in keys msgs):
        send({ .type: .write, .src: (uid, 2), .value: (t, v) })
    result = v
    select msgs in receive(uid, 2):
        pass

def write(uid, v):
    send({ .type: .read, .src: (uid, 1) })
    select msgs in receive(uid, 1):
        let (t,) = max(m.value for m in keys msgs)
        let nt = (t[0] + 1, uid):
        send({ .type: .write, .src: (uid, 2), .value: (nt, v) })
    select msgs in receive(uid, 2):
        pass

Figure 26.1: [code/abd.hny](code/abd.hny) An implementation of a replicated atomic read/write register.
import abd

def reader():
    let first = abd.read((.reader, 1))
    let second = abd.read((.reader, 2)):
    assert (first == .token) => (second == .token)

def writer():
    abd.write((.writer, 1), .token)

for i in {1 .. abd.N}:
    spawn eternal abd.server()
    spawn reader()
    spawn writer()

Figure 26.2: [code/abdtest.hny] Testing the code in Figure 26.1.

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 26.1 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 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 (aka multiset) of messages. The bag itself is modeled in Harmony as a dictionary that maps unique elements in the bag to their multiplicity. So, for example, \{a : 1, b : 2\} is the bag that consists of one .a and two copies of .b. The bag module (Section B.2) contains a variety of handy functions on bags.
A server initializes its timestamp $t$ to $(0, \text{None})$ and its value to $\text{None}$. 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 26.2 tests the third property of the properties listed above. The `writer()` method writes a new value `.token`. The `reader()` method reads twice and makes sure that if the first read operation returns `.token`, then so does the second. This illustrates the importance of the second phase of the `read` operation. You can comment out Lines 32 and 33 in Figure 26.1 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 27

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. In this chapter, we will present a simple consensus algorithm that can tolerate fewer than $1/3$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. 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 two 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.

The consensus problem is surprisingly tricky to solve in the face of processor failures and without making assumptions about how long it takes to send and receive a message.

Figure 27.1 presents our algorithm. Besides the network variable, it uses a shared list of proposals and a shared set of decisions. To reduce the state space to explore, not all permutations of zeroes and ones are explored. With 5 processors ($F = 2$), say, we only explore the cases where no processors propose 1, where exactly one processor proposes 1, and where 2 processors propose 1. In this particular algorithm, all messages are broadcast to all processors, so they do not require a destination address. The invariants we want to maintain on those variables is that $\text{decisions} \subseteq \text{proposals}$ and $|\text{decisions}| \leq 1$. Since the initial value of $\text{decisions}$ is $\emptyset$, the invariants clearly hold initially.

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 and proposal. Therefore, messages contain a round number and a proposal. To start things, each processor first broadcasts its initial
import bag

const F = 2
const N = (3 * F) + 1
const NROUNDS = 3

network = bag.empty()

let n_zeroes = choose({ 0 .. N / 2 }):
    proposals = [ 0 if i <= n_zeroes else 1 for i in { 1 .. N } ]

decisions = {}

def broadcast(msg):
    atomic: network = bag.add(network, msg)

def receive(round, k):
    let msgs = { e:c for (r,e):c in network where r == round }:
        result = bag.combinations(msgs, k)

def processor(proposal):
    broadcast(0, proposal)
    for round in {0..NROUNDS–1}:
        select quorum in receive(round, N – F):
            let count = [ bag.count(quorum, i) for i in { 0..1 } ]:
                assert count[0] != count[1]
                proposal = 0 if count[0] > count[1] else 1
                if count[proposal] == (N – F):
                    assert proposal in proposals  # validity
                    possibly proposal == 0, proposal == 1  # can decide either value
                    decisions |= { proposal }
                    assert len(decisions) <= 1  # agreement
                    broadcast(round + 1, proposal)
    for i in {0..N–1}:
        spawn processor(proposals[i])

Figure 27.1: [code/bosco.hny] A crash-tolerant consensus protocol.
round number (0) and proposal. 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 \texttt{NROUNDS} that limits the number of rounds. It may be that no decisions are made, but that does not violate either Validity or Agreement. We only check that if decisions are made, they satisfy Validity and Agreement.

In Line 22, a processor waits for $N - F$ messages using the Harmony \texttt{select} statement. Since Harmony has to check all possible executions of the protocol, the \texttt{receive}$(\text{round}, k)$ method returns all \textit{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 \texttt{bag.combinations} method to find all combinations of size $k$. The \texttt{select} statement waits until there is at least one such combination and then chooses an element, which is bound to the \textit{quorum} variable. The body of the \texttt{select} 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 \texttt{select} 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 \texttt{count[0]} stores the number of zeroes and \texttt{count[1]} stores the number of ones received in the round. The rules of the algorithm are simple:

- update \textit{proposal} to be the majority value;
- if the quorum is unanimous, decide the value.

After that, proceed with the next round.

The \texttt{possibly} statement checks if there exist executions where 0 is decided and if there exist executions where 1 is decided. Doing so is useful because the assertions would certainly not fail if no decision is ever made. Also, it is nice to know that both 0 and 1 can be decided.

While one can run this code within little time for $F = 2$, for $F = 3$ 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 \textit{all} $N$ messages in a round instead of $N - F$. It 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.

Figure 27.2 shows the code for this model. Running this with $F = 3$ does not take very long and this approach is a good blueprint for testing other round-based protocols (of which there are many).

**Exercises**

27.1 The algorithm as given works in the face of crash failures. A more challenging class to tolerate are \textit{arbitrary failures} in which up to $F$ processors may send arbitrary messages, including
import bag
const F = 2
const N = (3 * F) + 1
const NROUNDS = 3
network = bag.empty()
let n_zeroes = choose({ 0 .. N / 2 }):
  proposals = [ 0 if i <= n_zeroes else 1 for i in { 1 .. N } ]
decisions = {}
def broadcast(msg):
  atomic: network = bag.add(network, msg)
def receive(round):
  let msgs = { e:c for (r,c) in network where r == round }:
    result = {} if bag.size(msgs) < N else { msgs }
def processor(proposal):
  broadcast(0, proposal)
  for round in {0..NROUNDS–1}:
    select msgs in receive(round):
      let choices = bag.combinations(msgs, N – F)
      let quorum = choose(choices)
      let count = [ bag.count(quorum, i) for i in { 0..1 } ]:
        assert count[0] != count[1]
      proposal = 0 if count[0] > count[1] else 1
      if count[proposal] == (N – F):
        assert proposal in proposals  # validity
        possibly proposal == 0, proposal == 1  # can decide either value
        decisions |= { proposal }
        assert len(decisions) <= 1  # agreement
        broadcast(round + 1, proposal)
    for i in {0..N–1}:
      spawn processor(proposals[i])

Figure 27.2: [code/bosco2.hny] Reducing the state space.
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.

27.2 In 1983, Michael Ben-Or presented a randomized algorithm that can tolerate crash failures with just $N = 2F - 1$ processors [BO83]. Implement this algorithm.
Chapter 28

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 27, Paxos uses rounds of messaging. The communication pattern, however, is different. Similar to the atomic read/write register protocol in Chapter 26, 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 28.1 and Figure 28.2 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
import bag

const F = 1
const NACCEPTORS = (2 * F) + 1
const NLEADERS = F + 1
const NBALLOTS = 2

network = bag.empty()

let nfalse = choose({0..NLEADERS/2}):
  proposals = [ i <= nfalse for i in {1..NLEADERS} ]
decisions = {}

def send(m):
  atomic: network = bag.add(network, m)

def receive(ballot, phase):
  let msgs = { c:e for (b,p,t,e):c in network
              where (b,p,t) == (ballot, phase, .B) }
  result = bag.combinations(msgs, NACCEPTORS - F)

  for i in {0..NLEADERS - 1}:
    spawn leader(i + 1, proposals[i])
  for i in {1..NACCEPTORS}:
    spawn eternal acceptor()
import bag

const F = 1
const NACCEPTORS = (2 * F) + 1
const NLEADERS = F + 1
const NBALLOTS = 2

def leader(ballot, proposal):
    send(ballot, 1, .A, None)
    while ballot <= NBALLOTS:
        select quorum in receive(ballot, 1):
            let accepted = { e for e in quorum where e != None }:
                if accepted != {}:
                    _, proposal = max(accepted)
                send(ballot, 2, .A, proposal)
        select quorum in receive(ballot, 2):
            if bag.count(quorum, (ballot, proposal)) == (NACCEPTORS – F):
                assert proposal in proposals  # validity
                possibly proposal, not proposal # can decide either False or True
                decisions = { proposal }
                assert len(decisions) <= 1  # agreement
                ballot += NLEADERS
                if ballot <= NBALLOTS:
                    send(ballot, 1, .A, None)

def acceptor():
    let ballot, last_accepted, received = 0, None, {}:
    while True:
        select b,p,e in { (b,p,e) for b,p,t,e in network
            where ((b,p) not in received) and (t == .A) }:
            received |= { (b, p) }
            if b > ballot:
                ballot = b
            if p == 2:
                last_accepted = (ballot, e)
            send(b, p, .B, last_accepted)

Figure 28.2: [code/paxos.hny] A version of the Paxos protocol, Part 2.
2. A message to the acceptors and awaits \(N - F\) of their \textit{last accepted} values. If all those acceptors responded with \texttt{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.

In the initialization, the \texttt{proposals} variable is initialized to either \([\texttt{False, False}]\) or to \([\texttt{True, False}]\) (in case \texttt{NBALLOTS} = 2). The other combinations of \texttt{True} and \texttt{False} are symmetric, so there is no need to run the model checker through those scenarios as well, halving the running time.

\textbf{Exercises}

\textbf{28.1} Perhaps the trickiest detail of the algorithm is that, in Line 14 of Figure 28.2, the leader selects the proposal with the highest ballot number it receives. Replace the \texttt{max} operator in that statement with \texttt{choose} and see if it finds a problem. First try with \texttt{NBALLOTS} = 2 and then with \texttt{NBALLOTS} = 3. (Warning, the latter may take a long time.) If it finds a problem, analyze the output and see what went wrong.

\textbf{28.2} [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 29

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

When Bob receives \( \langle 1, A, N_A \rangle_{Bob} \), 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 \( \langle 2, N_A, N_B \rangle_{Alice} \), 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 \( \langle 3, N_B \rangle_{Bob} \), 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 \( N_B \).
network = {}

dest = choose({ None, .bob, .corey })

def send(m):
    atomic: network |= { m }

def alice():
    if dest != None:
        send({ .dst: dest, .contents: { .type: 1, .nonce: .nonceA, .initiator: .alice } })
        select m in network where (m.dst == .alice) and (m.contents.type == 2) and (m.contents.nonce == .nonceA):
            send({ .dst: dest, .contents: { .type: 3, .nonce: m.contents.nonce2 } })

def bob():
    select m in network where (m.dst == .bob) and (m.contents.type == 1) and (m.contents.initiator == .alice):
        send({ .dst: alice, .contents: { .type: 2, .nonce: m.contents.nonce, .nonce2: .nonceB } })
    select m in network where (m.dst == .bob) and (m.contents.type == 3) and (m.contents.nonce == .nonceB):
        assert dest == .bob

def corey():
    let received, nonces, msgs = {}, { .nonceC }, {}:
        while True:
            select m in network - received where m.dst == .corey:
                received |= { m }
                nonces |= { m.contents.nonce }
            if m.contents.type == 2:
                nonces |= { m.contents.nonce2 }
            for dst in { .alice, .bob } for n in nonces:
                msgs |= {{ .dst: dst, .contents: { .type: 1, .nonce: n, .initiator: ini } }}
                for ini in { .alice, .bob }
                msgs |= {{ .dst: dst, .contents: { .type: 2, .nonce: n, .nonce2: n2 } }}
                for n2 in nonces }
                msgs |= {{ .dst: dst, .contents: { .type: 3, .nonce: n } }}
                send(choose(msgs - network))

        spawn alice(); spawn bob()
        spawn eternal corey()

Figure 29.1: [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 29.1 shows the protocol implemented in Harmony. A message $\langle m \rangle_p$ is encoded in Harmony as a dictionary \{\texttt{dst} : $p$, \texttt{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 29.1 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 29.1 quickly finds a viable attack. The attack goes like this:

1. Alice creates a new nonce $N_A$ and sends $\langle 1, A, N_A \rangle_{Corey}$ to Corey;
2. Upon receipt, Corey sends $\langle 1, A, N_A \rangle_{Bob}$ to Bob;
3. Upon receipt, Bob, believing it is engaging in the protocol with Alice, creates a new nonce $N_B$ and sends $\langle 2, N_A, N_B \rangle_{Alice}$ 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 $\langle 3, N_B \rangle_{Corey}$ to Corey.
5. Corey learns $N_B$ and sends $\langle 3, N_B \rangle_{Bob}$ to Bob.
6. Bob receiving $\langle 3, N_B \rangle_{Bob}$ 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**

29.1 Figure out how to fix the protocol.

29.2 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


Appendix A

Harmony Language Details

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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>False, True</td>
</tr>
<tr>
<td>Integer</td>
<td>-inf, ..., -2, -1, 0, 1, 2, ..., inf</td>
</tr>
<tr>
<td>Atom</td>
<td>.example, .test1, .0x4A</td>
</tr>
<tr>
<td>Program Counter</td>
<td>(method names are program counter constants)</td>
</tr>
<tr>
<td>Dictionary</td>
<td>{ .account: 12345, .valid: False }</td>
</tr>
<tr>
<td>Set</td>
<td>{}, { 1, 2, 3 }, { False, .id, 3 }</td>
</tr>
<tr>
<td>Address</td>
<td>?lock, ?flags[2], None</td>
</tr>
<tr>
<td>Context</td>
<td>(generated by stop expression)</td>
</tr>
</tbody>
</table>

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

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

- False < True;
- integers are ordered in the natural way;
- atoms are lexicographically ordered;
- program counters are ordered by their integer values;
- dictionaries are first converted into a list of ordered (key, value) pairs. Then two dictionaries are lexicographically ordered by this representation;
• a set is first converted into an ordered list, then lexicographically ordered;
• an address is a list of atoms. None is the empty list of atoms. Addresses are lexicographically ordered accordingly;
• contexts (Section C.2) are ordered first by name, then by program counter, then by the remaining content.

Generic operators on Harmony values include:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e == e$</td>
<td>equivalence</td>
</tr>
<tr>
<td>$e != e$</td>
<td>inequivalence</td>
</tr>
<tr>
<td>$e &lt; e$, $e &lt;= e$, $e &gt; e$, $e &gt;= e$</td>
<td>comparison</td>
</tr>
</tbody>
</table>

**Boolean**

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

Operations on booleans include:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$ and $e$ and ...</td>
<td>conjunction</td>
</tr>
<tr>
<td>$e$ or $e$ or ...</td>
<td>disjunction</td>
</tr>
<tr>
<td>$e =&gt; e$, $e$ not =&gt; e</td>
<td>implication</td>
</tr>
<tr>
<td>not $e$</td>
<td>negation</td>
</tr>
<tr>
<td>$v$ if $e$ else $v'$</td>
<td>$v$ or $v'$ depending on $e$</td>
</tr>
<tr>
<td>any $s$, all $s$</td>
<td>disjunction / conjunction for set or list $s$</td>
</tr>
</tbody>
</table>

**Integer**

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

Operations on integers include:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( -e )</td>
<td>negation</td>
</tr>
<tr>
<td>abs(e)</td>
<td>absolute value</td>
</tr>
<tr>
<td>( e + e + \ldots )</td>
<td>sum</td>
</tr>
<tr>
<td>( e - e )</td>
<td>difference</td>
</tr>
<tr>
<td>( e \times e \times e \ldots )</td>
<td>product</td>
</tr>
<tr>
<td>( e \div e, e \div e )</td>
<td>integer division</td>
</tr>
<tr>
<td>( e % e, e \mod e )</td>
<td>integer division remainder</td>
</tr>
<tr>
<td>( e \times e )</td>
<td>power</td>
</tr>
<tr>
<td>( e \equiv e )</td>
<td>binary inversion</td>
</tr>
<tr>
<td>( e &amp; e &amp; \ldots )</td>
<td>binary and</td>
</tr>
<tr>
<td>( e</td>
<td>e</td>
</tr>
<tr>
<td>( e \wedge e \wedge \ldots )</td>
<td>binary exclusive or</td>
</tr>
<tr>
<td>( e \ll e )</td>
<td>binary shift left</td>
</tr>
<tr>
<td>( e \gg e )</td>
<td>binary shift right</td>
</tr>
<tr>
<td>{e..e′}</td>
<td>set of integers from e to e′ inclusive</td>
</tr>
</tbody>
</table>

**Atom**

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

**Set**

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len s</td>
<td>cardinality</td>
</tr>
<tr>
<td>s - s</td>
<td>set difference</td>
</tr>
<tr>
<td>s &amp; s &amp; \ldots</td>
<td>intersection</td>
</tr>
<tr>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>s \wedge s \wedge \ldots</td>
<td>inclusion/exclusion (elements in odd number of sets)</td>
</tr>
<tr>
<td>choose s</td>
<td>select an element (Harmony will try all)</td>
</tr>
<tr>
<td>min s</td>
<td>minimum element</td>
</tr>
<tr>
<td>max s</td>
<td>maximum element</td>
</tr>
<tr>
<td>any s</td>
<td>True if any value is True</td>
</tr>
<tr>
<td>all s</td>
<td>True if all values are True</td>
</tr>
</tbody>
</table>

Harmony also supports set comprehension. In its simplest form, \{ f(v) for v in s \} returns a set that is constructed by applying f to all elements in s (where s is a set or a list). This is known as mapping. But set comprehension is much more powerful and can include joining multiple sets (using nested for loops) and filtering (using the where keyword).
For example: \{ x + y \text{ for } x \text{ in } s \text{ for } y \text{ in } s \text{ where } (x \cdot y) == 4 \} \text{ returns a set that is constructed by summing pairs of elements from } s \text{ that, when multiplied, have the value } 4.

**Dictionary**

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

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

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

Operations on dictionaries include the following:

<table>
<thead>
<tr>
<th>d k</th>
<th>indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>len d</td>
<td>the number of keys in d</td>
</tr>
<tr>
<td>keys d</td>
<td>the set of keys in d</td>
</tr>
<tr>
<td>v [not] in keys d</td>
<td>check if v is a key in d</td>
</tr>
<tr>
<td>v [not] in d</td>
<td>check if v is a value in d</td>
</tr>
<tr>
<td>min d</td>
<td>the minimum value in d</td>
</tr>
<tr>
<td>max d</td>
<td>the maximum value in d</td>
</tr>
<tr>
<td>any d</td>
<td>True if any value is True</td>
</tr>
<tr>
<td>all d</td>
<td>True if all values are True</td>
</tr>
<tr>
<td>d &amp; d &amp; ...</td>
<td>dictionary intersection</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

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

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

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

**List or Tuple**

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

You can denote a list by a sequence of values, each value terminated by a comma. As per usual, you can use brackets or parentheses at your discretion. For Python users, the important thing to note is that a singleton list in Harmony must contain a comma. For example [1,] is a list containing the value 1, while [1] is simply the value 1.
The list module (Section B.4) contains various convenient routines that operate on lists or tuples. Native operations on lists or tuples include the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t[k]</code></td>
<td>indexing</td>
</tr>
<tr>
<td><code>t + t + ...</code></td>
<td>concatenation</td>
</tr>
<tr>
<td><code>t * n</code></td>
<td><code>n</code> copies of <code>t</code> concatenated</td>
</tr>
<tr>
<td><code>v [not] in t</code></td>
<td>check if <code>v</code> is a value in <code>t</code></td>
</tr>
<tr>
<td><code>len t</code></td>
<td>the length of <code>t</code></td>
</tr>
<tr>
<td><code>min t</code></td>
<td>the minimum value in <code>t</code></td>
</tr>
<tr>
<td><code>max t</code></td>
<td>the maximum value in <code>t</code></td>
</tr>
<tr>
<td><code>any t</code></td>
<td><code>True</code> if any value is <code>True</code></td>
</tr>
<tr>
<td><code>all t</code></td>
<td><code>True</code> if all values are <code>True</code></td>
</tr>
<tr>
<td><code>t &amp; t &amp; ...</code></td>
<td>pairwise list minimum</td>
</tr>
<tr>
<td>`t</td>
<td>t</td>
</tr>
</tbody>
</table>

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

```python
any((x%2) == 0 for x in t).
```

**String**

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

Native operations on strings include the following:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[k]</code></td>
<td>indexing</td>
</tr>
<tr>
<td><code>s + s + ...</code></td>
<td>concatenation</td>
</tr>
<tr>
<td><code>s * n</code></td>
<td><code>n</code> copies of <code>s</code> concatenated</td>
</tr>
<tr>
<td><code>c [not] in s</code></td>
<td>check if <code>c</code> (a one-character atom) is in <code>s</code></td>
</tr>
<tr>
<td><code>len s</code></td>
<td>the length of <code>s</code></td>
</tr>
</tbody>
</table>

**Bag or Multiset**

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v [not] in keys b</code></td>
<td>check if <code>v</code> is in <code>b</code></td>
</tr>
<tr>
<td><code>t &amp; t &amp; ...</code></td>
<td>bag intersection</td>
</tr>
<tr>
<td>`t</td>
<td>t</td>
</tr>
</tbody>
</table>
**Program Counter**

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

Operations on program counters include the following:

<table>
<thead>
<tr>
<th>m a</th>
<th>invoke method with program counter m and argument a</th>
</tr>
</thead>
<tbody>
<tr>
<td>atLabel l</td>
<td>return a bag of (method, argument) pairs of threads executing at label l</td>
</tr>
</tbody>
</table>

You can create lambda functions similarly to Python. For example: \( \text{lambda}(x, y): x + y. \)

**Address**

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

**Context**

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

A thread can call \texttt{get\_context()} to retrieve its current context.

**A.2 Statements**

Harmony currently supports the following statements (below, \( S \) is a list of statements):
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 `x = f()`.

Assignment

The statement `x = 3` changes the state by assigning 3 to variable `x` (assuming `x` was not already 3). `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 computes the address of `x[f()]`, then computes 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 `=.`. Evaluates the righthand side expression but is otherwise a no-op.

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 is at the location where the **assert** statement is, then \( b \) is always true.

If \( b \) is an expression, then it 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 well.

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.

atomic

The statement **atomic** \( S_1; S_2; \ldots \) evaluates statements \( S_1, S_2, \ldots \) atomically in that no other threads can run while this atomic block is executing. Typically an atomic block runs to completion before any other thread can run. 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.

The **atomic** statement is useful for implementing synchronization primitives such as test-and-set. It is also useful for testing. It is not a replacement for lock/unlock, and should not generally be used for synchronization otherwise. Lock/unlock does allow other threads to run concurrently—just not in the same critical section.

await

The statement **await** \( b \) is equivalent to **while** \( \neg b \): **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** \( m: a : S_1; S_2 : \ldots \) defines a new program counter constant \( m \) referring to a method that takes an argument \( a \) and executes the statements \( S_1, S_2, \ldots \). 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 variable names are assigned upon application. It is allowed, but discouraged, to updates those local variables in statements \( S_1, S_2, \ldots \). Each method has a predefined local variable \( result \), initialized to **None**, that is returned by the method. 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.

Because Harmony lists are dictionaries, deleting from lists is different from Python: `x = [a, b]; del x[0]` results in `x` having value `{1 : b}` rather than `[b]` (which is `{0 : b}`).

**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 or a dictionary. `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 values of the dictionary in the order of the keys. This makes iterating over lists intuitive and identical to Python. 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.

It is allowed, but discouraged, to assign different values to `x` within statements `S1`, `S2`, ...

If `y` is a dictionary, 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.

The statement also supports nesting and filtering. Nesting is of the form `for x1 in y1 for x2 in y2 : S1; S2; ...`, which 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 (program counter constants) within the module.

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

**go**

The statement `go c e` starts a thread with context `c` that has executed a `stop` expression. The `stop` expression returns value `e`. The same context can be started multiple times, allowing threads to fork.

**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 and error. Harmony also reports an error if the expression evaluates to a value other than False or True.

let

You can introduce new local variables in a method using the let expression. The statement let a = b : S₁; S₂, ... evaluates b, assigns the result to local 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. The variables may be updates in statements S₁, S₂, ....

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:

<table>
<thead>
<tr>
<th>Nested let statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>let a = y;</td>
</tr>
<tr>
<td>let b = z;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>let a = y</td>
</tr>
<tr>
<td>let b = z;</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

pass

The pass statement does nothing.
possibly

The statement possibly \( b_1, b_2, \ldots \) atomically evaluates all predicates \( b_i \). At completion, Harmony reports which of the predicates never held. While `assert` statements check that nothing ever goes wrong, possibly statements can check that certain things sometimes go right.

**select ... in ... [ where ... ]**

The statement `select x in y : S_1; S_2; ...` requires that \( y \) evaluates to a set value. The statement does the following three things atomically:

- it waits until \( y \) is non-empty;
- it selects one element of \( y \) non-deterministically (using a `choose` expression);
- it executes statements \( S_1, S_2, \ldots \) with the selected element assigned to local 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. If `waiting` is an unused local variable, then `select x in y : S_1; S_2; ...` is equivalent to

```plaintext
let waiting = True :
    while waiting:
        atomic:
            if y != {}:
                let x = choose(y):
                S_1
                S_2
                ...
            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.

The optional `where` clause can be used to filter the set down to the elements that satisfy the condition following the `where` keyword.

**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 doing so would lead to high overhead.
spawn
The statement `spawn m a` starts a new thread that executes method `m` with argument `a`. `m` must be a program counter constant, and `a` is typically a tuple containing zero or more parameters to be passed to the method.

The default thread-local state of the thread, called `self`, is the empty dictionary by default. It can be specified by adding a parameter: `spawn m a, e` specifies that `e` should be the initial value of the thread-local state.

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 m a` specifies that the current thread should execute method `m` with argument `a` and some future, unspecified, time. It 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.

while
The statement `while c : S_1; S_2; ...` executes statements `S_1`, `S_2`, ... 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:

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

In Python, lists are objects. Thus `x` and `y` point to the same list, and the assertion would fail if executed by Python. In Harmony, lists are values. So, when `x` is updated in Line 2, it does not affect the value of `y`. The assertion succeeds. Harmony supports references to values (Chapter 7), allowing programs to implement shared objects.

Because Harmony does not have objects, it also does not have object methods. However, Harmony methods and lambdas are program counter constants. These constants can be added to dictionaries. For example, in Figure 7.1 you can add the `P_enter` and `P_exit` methods to the `P_mutex` dictionary like so:

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

That would allow you to simulate object methods.

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

```python
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:

```python
lock = Lock()
x = y = ?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 constant, a global shared variable, a local variable, or a module. Constants are declared using `const` statements. Those constants are computed at compile-time.

Local variables all declared. They can be declared in `def` statements (i.e., arguments), `let` statements, `for` loops, and `select` statements. Also, each method has an implicitly declared `result` variable, which is initialized to `None`. Each thread has a variable called `this` that contains the thread-local state. Local 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 any threads are spawned.

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

### 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 ...`. We call this *application*, and application is left-associative. So, `a b c` is interpreted as `(a) b c` where `b` is applied to `a`, and then `c` is applied to the result. For readability, it may help to write `a(b)` for function application and `a[b]` for indexing. In case `b` is an atom, you can also write `a.b` for indexing.

There are three 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, and the remaining operators have the lowest precedence. For example, `-2 + 3` evaluates to 1, not -5.

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

In almost all expressions, subexpressions are evaluated left to right. So, \( a[b] + c \) first evaluates \( a \), then \( b \) (and then applies \( b \) to \( a \)), and then \( c \). The one exception is the expression \( a \text{ if } c \text{ else } b \), where \( c \) is evaluated first. In that expression, only \( a \) or \( b \) is evaluated depending on the value of \( c \). In the expression \( a \text{ and } b \text{ and } \ldots \), evaluation is left to right but stops once one of the subexpressions evaluates to \text{False}. Similarly for \text{or}, where evaluation stops once one of the subexpressions evaluates to \text{True}. A sequence of comparison operations, such as \( a < b < c \), is evaluated left to right but stops as soon as one of the comparisons fails.

As an aside: the expression \( a \text{ not in } b \) is equivalent to \( \text{not} (a \text{ in } b) \). Harmony generalizes this construct for any pair of a unary (except ‘−’) and a binary operator. In particular, \( a \text{ not and } b \) is the same as \( \text{not} (a \text{ and } b) \). For those familiar with logic gates, \( \text{not and} \) is the equivalent of \text{NAND}. Similarly, \( \text{not ==>} \) is non-implication.

### A.6 Tuples, Lists, and Pattern Matching

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

Because tuples and lists are dictionaries, the \texttt{del} statement is different from Python. For example, if \( x = [a, b, c] \), then \texttt{del} \( x[1] \) results in \( x = \{ 0:a, 2:c \} \), \( \text{not} x = [a, .c] \). Harmony also does not support special slicing syntax like Python. To modify lists, use the \texttt{subseq} method in the \texttt{list} module (Section B.4).

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

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

As in Python, pattern matching can also be used in \texttt{for} statements. For example:

```python
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:

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

and then call \( f[1, (2, 3)] \). Note that the more familiar: \texttt{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
```python
from stack import Stack, push, pop

teststack = Stack()
push(teststack, 1)
push(teststack, 2)
v = pop(teststack)
assert v == 2
push(teststack, 3)
v = pop(teststack)
assert v == 3
v = pop(teststack)
assert v == 1
```

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

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

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

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

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

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 `select` statements.

A.7 Dynamic Allocation

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

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

def Stack():
    result = []

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

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

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

def Stack():
    result = ()

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

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

Figure A.4: [code/stack3.hny] Stack implemented using a recursive tuple data structure.
from alloc import malloc, free

def Stack():
    result = None

def push(st, v):
    st = malloc({.value: v, .rest: st})

def pop(st):
    node = st:
    result = node->value
    st = node->rest
    free(node)

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

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

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

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

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

A.8 Comments

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

Modules

B.1 The alloc module

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

<table>
<thead>
<tr>
<th>alloc(v)</th>
<th>return a pointer to a memory location initialized to v</th>
</tr>
</thead>
<tbody>
<tr>
<td>free(p)</td>
<td>free an allocated memory location p</td>
</tr>
</tbody>
</table>

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.2 The bag module

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

<table>
<thead>
<tr>
<th>empty()</th>
<th>returns an empty bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromSet(s)</td>
<td>create a bag from set s</td>
</tr>
<tr>
<td>fromList(t)</td>
<td>convert list t into a bag</td>
</tr>
<tr>
<td>count(b, e)</td>
<td>count how many times e occurs in bag b</td>
</tr>
<tr>
<td>bchoose(b)</td>
<td>like choose(s), but applied to a bag</td>
</tr>
<tr>
<td>add(b, e)</td>
<td>add one copy of e to bag b</td>
</tr>
<tr>
<td>remove(b, e)</td>
<td>remove one copy of e from bag b</td>
</tr>
<tr>
<td>combinations(b, k)</td>
<td>return set of all subbags of size k</td>
</tr>
</tbody>
</table>

B.3 The hoare module

The hoare module implements support for Hoare-style monitors and condition variables.
Monitor()  return a monitor mutex
enter(m)  enter a monitor. m points to a monitor mutex
exit(m)  exit a monitor
Condition()  return a condition variable
wait(c, m)  wait on condition variable pointed to by c in monitor pointed to by m
signal(c, m)  signal a condition variable

B.4 The list module
The list module has various useful methods that operate on lists or tuples:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subseq(t, b, f)</td>
<td>return a slice of list t starting at index b and ending just before f</td>
</tr>
<tr>
<td>append(t, e)</td>
<td>append e to list t</td>
</tr>
<tr>
<td>head(t)</td>
<td>return the first element of list t</td>
</tr>
<tr>
<td>tail(t)</td>
<td>return all but the first element of list t</td>
</tr>
<tr>
<td>reversed(t)</td>
<td>reverse a list</td>
</tr>
<tr>
<td>sorted(t)</td>
<td>sorted set or list</td>
</tr>
<tr>
<td>set(t)</td>
<td>convert values of a dict or list into a set</td>
</tr>
<tr>
<td>list(t)</td>
<td>convert set into a list</td>
</tr>
<tr>
<td>values(t)</td>
<td>convert values of a dict into a list sorted by key</td>
</tr>
<tr>
<td>items(t)</td>
<td>convert dict into (key, value) list sorted by key</td>
</tr>
<tr>
<td>enumerate(t)</td>
<td>like Python enumerate</td>
</tr>
<tr>
<td>sum(t)</td>
<td>return the sum of all elements in t</td>
</tr>
<tr>
<td>qsort(t)</td>
<td>sort list t using quicksort</td>
</tr>
</tbody>
</table>

B.5 The set module
The set module implements the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>issubset(s, t)</td>
<td>returns whether s is a subset of t</td>
</tr>
<tr>
<td>issuperset(s, t)</td>
<td>returns whether s is a superset of t</td>
</tr>
<tr>
<td>add(s, e)</td>
<td>returns s ∪ {e}</td>
</tr>
<tr>
<td>remove(s, e)</td>
<td>returns s {e}</td>
</tr>
<tr>
<td>combinations(b, k)</td>
<td>returns set of all subsets of size k</td>
</tr>
</tbody>
</table>

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

B.6 The synch module
The synch module provides the following methods:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tas(lk)</code></td>
<td>test-and-set on lk</td>
</tr>
<tr>
<td><code>cas(ptr, old, new)</code></td>
<td>compare-and-swap on ptr</td>
</tr>
<tr>
<td><code>BinSem(v)</code></td>
<td>return a binary semaphore initialized to v</td>
</tr>
<tr>
<td><code>Lock()</code></td>
<td>return a binary semaphore initialized to <code>False</code></td>
</tr>
<tr>
<td><code>acquire(bs)</code></td>
<td>acquire binary semaphore !bs</td>
</tr>
<tr>
<td><code>release(bs)</code></td>
<td>release binary semaphore !bs</td>
</tr>
<tr>
<td><code>Condition()</code></td>
<td>return a condition variable</td>
</tr>
<tr>
<td><code>wait(c, lk)</code></td>
<td>wait on condition variable !c and lock lk</td>
</tr>
<tr>
<td><code>notify(c)</code></td>
<td>notify a thread waiting on condition variable !c</td>
</tr>
<tr>
<td><code>notifyAll(c)</code></td>
<td>notify all threads waiting on condition variable !c</td>
</tr>
<tr>
<td><code>Semaphore(cnt)</code></td>
<td>return a counting semaphore initialized to cnt</td>
</tr>
<tr>
<td><code>P(sema)</code></td>
<td>procure !sema</td>
</tr>
<tr>
<td><code>V(sema)</code></td>
<td>vacate !sema</td>
</tr>
<tr>
<td><code>Queue()</code></td>
<td>return a synchronized queue object</td>
</tr>
<tr>
<td><code>get(q)</code></td>
<td>return next element of q, blocking if empty</td>
</tr>
<tr>
<td><code>put(q, item)</code></td>
<td>add item to a</td>
</tr>
</tbody>
</table>
Appendix C

The Harmony Virtual Machine

The Harmony Virtual Machine (HVM, Chapter 4) has the following state:

<table>
<thead>
<tr>
<th>code</th>
<th>a list of HVM machine instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>labels</td>
<td>a dictionary of atoms to program counters</td>
</tr>
<tr>
<td>variables</td>
<td>a dictionary mapping atoms to values</td>
</tr>
<tr>
<td>ctxbag</td>
<td>a bag of runnable contexts</td>
</tr>
<tr>
<td>stopbag</td>
<td>a bag of stopped contexts</td>
</tr>
<tr>
<td>choosing</td>
<td>if not None, indicates a context that is choosing</td>
</tr>
</tbody>
</table>

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

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

A stride can involve multiple steps. The following instructions start a new stride: `Load`, `Store`, `AtomicInc`, and `Continue`. The HVM interleaves 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>compute address from two components</td>
</tr>
<tr>
<td>Apply</td>
<td>pop ( m ) and ( i ) and apply ( i ) to ( m ), pushing a value</td>
</tr>
<tr>
<td>Assert, Assert2</td>
<td>pop ( b ) and check that it is True. Assert2 also pops value to print</td>
</tr>
<tr>
<td>AtomicInc/Dec</td>
<td>increment/decrement the atomic counter of this context</td>
</tr>
<tr>
<td>Continue</td>
<td>no-op (but causes a context switch)</td>
</tr>
<tr>
<td>Choose</td>
<td>choose an element from the set on top of the stack</td>
</tr>
<tr>
<td>Cut</td>
<td>cut a set into its smallest element and the remainder</td>
</tr>
<tr>
<td>Del ([ v ])</td>
<td>delete shared variable ( v )</td>
</tr>
<tr>
<td>DelVar ([ v ])</td>
<td>delete thread variable ( v )</td>
</tr>
<tr>
<td>Dup</td>
<td>duplicate the top element of the stack</td>
</tr>
<tr>
<td>Frame ( m \ a )</td>
<td>start method ( m ) with arguments ( a ), initializing variables.</td>
</tr>
<tr>
<td>Go</td>
<td>pop context and value, push value on context’s stack, and add to context bag</td>
</tr>
<tr>
<td>IncVar ( v )</td>
<td>increment thread variable ( v )</td>
</tr>
<tr>
<td>Jump ( p )</td>
<td>set program counter to ( p )</td>
</tr>
<tr>
<td>JumpCond ( e \ p )</td>
<td>pop expression and, if equal to ( e ), set program counter to ( p )</td>
</tr>
<tr>
<td>Load ([ v ])</td>
<td>push the value of a shared variable onto the stack</td>
</tr>
<tr>
<td>LoadVar ([ v ])</td>
<td>push the value of a thread variable onto the stack</td>
</tr>
<tr>
<td>Move ( i )</td>
<td>move stack element at offset ( i ) to top of the stack</td>
</tr>
<tr>
<td>( n )-ary ( op )</td>
<td>apply ( n )-ary operator ( op ) to the top ( n ) elements on the stack</td>
</tr>
<tr>
<td>Pop</td>
<td>pop a value of the stack and discard it</td>
</tr>
<tr>
<td>Possibly ( i )</td>
<td>pop a value to check the ( i^{th} ) predicate in a possibly statement</td>
</tr>
<tr>
<td>Push ( c )</td>
<td>push constant ( c ) onto the stack</td>
</tr>
<tr>
<td>ReadonlyInc/Dec</td>
<td>increment/decrement the read-only counter of this context</td>
</tr>
<tr>
<td>Return</td>
<td>pop return address, push result, and restore program counter</td>
</tr>
<tr>
<td>Sequential</td>
<td>pop an address of a variable that has sequential consistency</td>
</tr>
<tr>
<td>SetIntLevel</td>
<td>pop ( e ), set interrupt level to ( e ), and push old interrupt level</td>
</tr>
<tr>
<td>Spawn [eternal]</td>
<td>pop initial thread-local state, argument, and method and spawn a new context</td>
</tr>
<tr>
<td>Split</td>
<td>pop tuple and push its elements</td>
</tr>
<tr>
<td>Stop ([ v ])</td>
<td>save context into shared variable ( v ) and remove from context bag</td>
</tr>
<tr>
<td>Store ([ v ])</td>
<td>pop a value from the stack and store it in a shared variable</td>
</tr>
<tr>
<td>StoreVar ([ v ])</td>
<td>pop a value from the stack and store it in a thread variable</td>
</tr>
<tr>
<td>Trap</td>
<td>pop interrupt argument and method</td>
</tr>
</tbody>
</table>
Clarifications:

- The **Address** instruction expects two values on the stack. The top value must be an address value, representing a dictionary. The other value must be a key into the dictionary. The instruction then computes the address of the given key.

- Even though Harmony code does not allow taking addresses of thread variables, both shared and thread variables can have addresses.

- The **Load**, **LoadVar**, **Del**, **DelVar**, and **Stop** instructions have an optional variable name: if omitted the top of the stack must contain the address of the variable.

- **Store** and **StoreVar** instructions have an optional variable name. In both cases the value to be assigned is on the top of the stack. If the name is omitted, the address is underneath that value on the stack.

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

- The **Frame** instruction pushes the value of the thread register (i.e., the values of the thread variables) onto the stack. It initializes the result variable to **None**. The **Return** instruction restores the thread register by popping its value of the stack.

- All method calls have exactly one argument, although it sometimes appears otherwise:
  - \( m() \) invokes method \( m \) with the empty dictionary \( () \) as argument;
  - \( m(a) \) invokes method \( m \) with argument \( a \);
  - \( m(a, b, c) \) invokes method \( m \) with tuple \( (a, b, c) \) as argument.

  The **Frame** instruction unpacks the argument to the method and places them into thread variables by the given names.

- Every **Stop** instruction must immediately be followed by a **Continue** instruction.

### C.2 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:
<table>
<thead>
<tr>
<th>name</th>
<th>the name of the main method that the thread is executing</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>the argument given to the main method</td>
</tr>
<tr>
<td>program counter</td>
<td>an integer value pointing into the code</td>
</tr>
<tr>
<td>frame pointer</td>
<td>an integer value pointing into the stack</td>
</tr>
<tr>
<td>atomic</td>
<td>if non-zero, the thread is in atomic mode</td>
</tr>
<tr>
<td>readonly</td>
<td>if non-zero, the thread is in read-only mode</td>
</tr>
<tr>
<td>stack</td>
<td>a list of Harmony values</td>
</tr>
<tr>
<td>method variables</td>
<td>a dictionary mapping atoms (names of method variables) to values</td>
</tr>
<tr>
<td>thread-local variables</td>
<td>a dictionary mapping atoms (names of thread-local variables) to values</td>
</tr>
<tr>
<td>stopped</td>
<td>a boolean indicating if the context is stopped</td>
</tr>
<tr>
<td>failure</td>
<td>if not None, string that describes how the thread failed</td>
</tr>
</tbody>
</table>

Details:

- The frame pointer points to the current stack frame, which consists of the caller’s frame pointer and variables, the argument to the method, an “invocation type atom” (normal, interrupt, or thread), and the return address (in case of normal).

- A thread terminates when it reaches the Return instruction of the top-level method (when the stack frame is of type thread) or when it hits an exception. Exceptions include divide by zero, reading a non-existent key in a dictionary, accessing a non-existent variable, as well as when an assertion fails;

- The execution of a thread in atomic mode does not get interleaved with that of other threads.

- The execution of a thread in read-only mode is not allowed to update shared variables of spawn threads.

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

How Harmony Works

This appendix gives a very brief overview of how Harmony works. In a nutshell, Harmony goes through the following three 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. If there is a problematic path in this graph (see below), then it will report the shortest such path in the x.hco output file in JSON format.

3. 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 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.

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 statements. So the model checker only interleaves threads at **Load** and **Store** 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 if finds—if one fails, Harmony stops further exploration;
- An assertion failing;
- 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 of a dictionary, and so on;
- An infinite loop: a thread goes into an infinite loop without accessing shared variables.

The model checker also counts the number of times each **Possibly** instruction is invoked with the value **True**. It reports those **Possibly** instructions that either have never been reached or never have been executed with the value **True**.
D.3 Model Checker Output Analysis

The output of the model checker is a graph that is typically very large. 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 Kosaraju’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.

Finally, in the output, each thread has a unique identifier: $T_0$ is the initialization thread; $T_n$ 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

Installing and Running Harmony

There are currently three options for using Harmony:

1. You can download and install a command-line version of Harmony on your computer;
2. You can download and install HarmonyLang, a VSCode plug-in for Harmony;
3. You can run Harmony programs in the cloud using Harmony IDE.

Below you can find more information on each of these options.

E.1 Command-Line

You can get the latest released developer version of Harmony by downloading from harmony.cs.cornell.edu. It includes the latest bug fixes and features at any time. Harmony is developed and maintained by the author on both MacOSX and Linux, and so these are the preferred platforms at this time. A Windows version is also available, but may not be the latest version and is currently not as well supported.

When you download the .zip file, you will get the following files:

- README.txt: installation documentation;
- archive.xml: portable compressed archive of the code base;
- install.py: Python3 program to install and update the code base.

First place this directory (folder) where you would like it—the Downloads folder is at best a good temporary place. You can put the directory in your home directory, for example.

Installation requires Python3 and a 64-bit C compiler. The developer uses recent versions of gcc and clang for development. Run python3 install.py to install the code base. It extracts files from archive.xml and installs them in the current directory. It will try to compile the model checker using gcc. If that fails under Windows, it will install an executable that is pre-compiled on a Windows 10 machine. If these options do not work, you can compile the file charm.c by hand using a 64-bit C compile and place the output in charm.exe.
After installation, there is a harmony file for use under MacOSX and Linux, and a harmony.bat file for use under Windows. You should be able to run `harmony --help` in any of these environments.

If you would like to run `harmony` from any directory, you have to add the current directory to your search path. Under MacOSX and Linux, you will have to set the PATH environment variable. See [http://www.linux-migration.org/ch02s06.html](http://www.linux-migration.org/ch02s06.html) for more information. Under Windows, search “Edit environment variables” in the search bar. You can add the directory either to the Path associated with your account or to the system Path. If you do not install `harmony` in your search path, you may have to run `./harmony` in the installation directory instead of just `harmony`.

The installation directory will have the following subdirectories:

- `code`: contains all the code examples from this book;
- `modules`: contains the Harmony modules;
- `python`: contains the Python examples from the book.

For example, you can try `harmony code/Diners.hny` to run the Dining Philosophers code. Harmony currently produces three output files:

- `code/Diners.hvm`: the bytecode in JSON format;
- `code/Diners.hco`: the output of the model checker in JSON format;
- `code/Diners.htm`: the model checker output converted to HTML format.

You are probably only interested to see the last one, which you should be able to view in any web browser of your choice, including Safari, Chrome, Edge, or Firefox.

You can see if you are running the latest version of Harmony at any time by running `python3 install.py --check`. If you would like to update your installation, run `python3 install.puy --update`.

### E.2 HarmonyLang for VSCode

The VSCode plug-in, developed by Kevin Sun and Anthony Yang, is available from the VSCode marketplace: [https://marketplace.visualstudio.com/items?itemName=kevinsun-dev-cornell.harmonylang](https://marketplace.visualstudio.com/items?itemName=kevinsun-dev-cornell.harmonylang). This page comes with installation documentation. VSCode also sports a wonderful animator for Harmony output.

HarmonyLang is regularly updated to include the latest Harmony distribution.

### E.3 Harmony IDE for the cloud

Harmony IDE is an Integrated Development Environment for Harmony, also developed by Kevin Sun and Anthony Yang, that runs in the cloud. An important advantage of using Harmony IDE is that you do not have to install anything. You can even run it on your smartphone. However, since Harmony programs can use significant compute resources, Harmony IDE must put a limit on the amount of compute resources you can use. So while Harmony IDE is currently a great way to try out Harmony, if you become a serious user of Harmony you will probably want to install either the command-line version or HarmonyLang.
Acknowledgments

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

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

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. Later they were joined by Shi Chong Zhao, who also made significant contributions.

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

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

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

acknowledgment, 122  
acquire, 52  
actor model, 104  
adress, 17, 39  
alloc module, 172  
alternating bit protocol, 122  
atLabel operator, 25  
atom, 16  
atomic instruction, 45  
atomic statement, 45  
atomicity, 5  

bag, 17  
bag module, 172  
barrier synchronization, 107  
big lock, 56  
binary semaphore, 50  
blocked thread, 50  
blocking queue, 104  
bounded buffer, 80  
broadcast, 92  
buzzy waiting, 78  
bytecode, 16  

choose operator, 10  
circular buffer, 80  
client/server model, 80  
coarse-grained lock, 56  
constant, 10  
context, 17  
continuation, 17  
corner case, 6  
critical region, 24  
critical section, 24  

data race, 47  

deadlock, 98  
deadlock avoidance, 101  
determinism, 5, 133  
dictionary, 16  
dining philosopher, 98  
directory, 17  
distributed system, 122  
double turnstile, 111  
dynamic allocation, 55  

eexception, 112  
failure, 122  
fairness, 86  
fine-grained lock, 56  
formal verification, 6  

ggo statement, 50  
hand-over-hand locking, 56  
Harmony method, 39  
Harmony Virtual Machine, 16  
Heisenbug, 5  
HVM, 16  

import statement, 41  
inductive invariant, 33  
interleaving, 12  
interrupt, 112  
interrupt-safety, 112  
invariant, 6, 33  

label, 25  
linearizable, 60  
linearization point, 60  
list module, 172, 173  
liveness property, 25
lock, 26, 47
lock granularity, 56
logical timestamp, 67, 136

machine instruction, 12
Mesa, 92
message passing, 104
model checking, 5
module, 41, 52
monitor, 89
multiple conditions, waiting on, 99
multiset, 17
mutual exclusion, 25

network, 122
non-blocking synchronization, 119
non-determinism, 31
notify, 92
notifyAll, 92

pattern matching, 168
Peterson’s Algorithm, 31
pipeline, 80
pointer, 39
producer/consumer problem, 80
program counter, 17
progress, 25
property, 86
protocol, 122

race condition, 13
reachable state, 31
reader/writer lock, 78
register, 17
release, 52
replication, 133
retransmission, 122
ring buffer, 80

safety property, 25
select statement, 127
seqlock, 121
sequence number, 122
sequential, 5
sequential consistency, 19
set module, 173
shared variable, 5

signal, 89
single point of failure, 136
spinlock, 43, 45
split binary semaphore, 81
stack machine, 19
starvation, 52, 86
state, 31
state machine replication, 133
step, 31
stop expression, 50
stride, 175
synch module, 47, 173
synchronized queue, 104

TAS, 45
test, 5
test-and-set, 45
thread, 5, 11, 24
thread safety, 24
thread variable, 31
thread-local, 17
Time Of Check Time Of Execution, 53
TOCTOE, 53
trace, 31

virtual machine, 16

wait, 89
wait-free synchronization, 121

186
Glossary

**actor model** is a concurrency model where there are no shared variables, only threads with private variables that communicate through message passing. 104

**atomic instruction** a machine instruction that may involve multiple memory load and/or store operations and is executed atomically. 45

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

**barrier synchronization** is when a set of threads execute in rounds, waiting for one another to complete each round. 107

**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. 50

**busy waiting** (aka spin-waiting) is when a thread waits in a loop for some application-defined condition instead of blocking. 78

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

**condition variable** a variable that keeps track of which threads are waiting for a specific application-level condition. The variable can be waited on as well as signaled or notified. 89

**conditional critical section** is a critical section with, besides mutual exclusion, additional conditions on when a thread is allowed to enter the critical section. 81

**context** (aka continuation) describes the state of a running thread, including its program counter, the values of its variables (stored in its register), and the contents of its stack. 17

**critical section** (aka critical region) is a set of instructions that only one thread is allowed to execute at a time. The instructions are, however, not executed atomically, as other threads can continue to execute and access shared variables. 24
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. 47

deadlock is when there are two or more threads waiting indefinitely for one another to release a resource. 98

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

fairness is when each thread eventually can access each resource it needs to access with high probability. 86

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

linearizable is a consistency condition for concurrent access to an object, requiring that each access must appear to execute atomically sometime between the invocation of the access and its completion. 60

lock an object that can be owned by at most one thread at a time. Useful for implementing mutual exclusion. 26

machine instruction is an atomic operation on the Harmony virtual machine, executed by a thread. 12

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

monitor is a programming language paradigm that supports mutual exclusion as well as waiting for resources to become available. 89

mutual exclusion is the property that two threads never enter the same critical section. 24

non-blocking synchronization (aka wait-free synchronization) is when access to a shared resource can be guaranteed in a bounded number of steps even if other threads are not making progress. 119

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. 80

property describes a set of execution traces or histories that are allowed by a program. Safety properties are properties in which “no bad things happen,” such as violating mutual exclusion in a critical section. Liveness properties are properties where “something good eventually happens,” like threads being able to enter the critical section if they want to. 86

race condition describes when multiple threads access shared state concurrently, leading to undesirable outcomes. 13
**reader/writer lock** is a lock on a resource that can be held by multiple threads if they all only read the resource. 78

**replication** maintains multiple copies of some resource to improve availability in the face of failures. 133

**sequential consistency** is a consistency model in which shared memory accesses are executed in an order consistent with the program order. 31

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

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

**spinlock** is an implementation of a lock whereby a thread loops until the lock is available, at which point the thread atomically obtains the lock. 43

**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. 19

**starvation** is when a thread cannot make progress because it is continuously losing a competition with other threads to get access to a resource. 86

**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. 31

**state machine replication** is a replication technique in which a collection of deterministic state machines process the same inputs in the same order. 133

**step** is the execution of a machine instruction by a thread, updating its state. 31

**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. 11

**thread safety** is when the implementation of a data structure allows concurrent access with well-defined semantics. 24

**thread variable** is a variable that is private to a single thread and stored in its register. 31

**trace** is a sequence of steps, starting from an initial state. An infinite trace is also called a *behavior*. 31