Harmony Programming Assignments

This document describes two (related) programming assignments using Harmony: building a deque (double-ended queue) and building a blocking deque. The students are to develop three Harmony programs for each:

1. A specification (using the **atomically** keyword.
2. An implementation (using a doubly-linked list).
3. A test program (to generate behaviors that can be checked).

After the students submit their programs, everything is autograded. The basic approach for autograding is as follows.

Testing the Specification. We first run some basic sequential tests using assert statements. Then, using our own specification and test program, we generate a set of behaviors. We then run the same test program against the specification of the student's program to see if it generates the same set of behaviors. We gradually increase the difficulty. A trick that we use to increase scale is to initialize the deque with some sequential operations, and then spawn the threads for concurrent tests.

Testing the Implementation. This is not much different from testing the specification. Again, we first do some simple sequential tests, and then behavioral comparisons against our own specification and test program.

Testing the Test Program. We first run the test program against the specification. If this fails, then there is no reason to try anything else. After that, we run the test program against a variety of slightly broken code, some that should even fail sequential tests.

We share, approximately, how we tested the code with the students, and also provide the output from Harmony. In theory they should be able to regenerate the results. If not, they can go to an Office Hour or file a regrade request.

1 Example Project: Building a Deque

The lectures and the Harmony book describe a concurrent queue. You are to develop a similar abstraction called a "double-ended queue," commonly known as a *deque* (pronounced "deck"). Like a queue or a stack, a deque is a dynamic list of elements. But unlike those abstractions, you can add elements to either side and remove elements from either side. A deque can double as a FIFO queue and as a stack.

Start with a specification, which must have the following methods:

`Deque`(): returns the initial value of an empty deque.

put_left(*d*, *v*): *d* points to a deque and *v* is some value. Put *v* at the left-end of the deque. Does not return a value.

put_right(*d*, *v*): *d* points to a deque and *v* is some value. Put *v* at the right-end of the deque. Does not return a value.

get_left(*d*): *d* points to a deque. If the deque is empty, the operation should return **None**. Otherwise it should remove and return the value at the left-end of the deque.

get_right(*d*): *d* points to a deque. If the deque is empty, the operation should return **None**. Otherwise it should remove and return the value at the right-end of the deque.

Place the entire specification in file `deque.hny`. Note that all operations except for `Deque`() should be atomic. Please make sure the operations are named *exactly* as above, or the autograder will not be able to find them.

The deque *implementation*, which you should put in file `deque_impl.hny`, should define the same methods but cannot use the **atomically** or **sequential** keywords and should instead rely on `Lock`, `acquire`, and `release` from the `synch` module (which the `deque_impl.hny` file should **import**). As in the queue implementation, the contents of the deque should be implemented using a linked list, with nodes allocated using `malloc` and `free` from the `alloc` module. All operations should be O(1).

You also should create a test program, in file `deque_test.hny`, for your deque. Note that the test should be a *black box* test—it cannot look inside the implementation of the deque. *We should be able to run your test program with somebody else's implementation of the deque.* So, you should not deference the deque pointer to see what's in it—you can only use the deque interface in your test program.

Testing should be based on comparing behaviors between the specification and the implementation (a form of *differential testing*). Model your test program after the `queue_btest1.hny` test program from the Harmony book. It should import **deque**, *not* **deque_impl**. It should allocate a single deque using `Deque`(). The program should define the following constants, all initialized to 1 in the program:

`N_PUT_LEFT`: the number of threads that execute a *put_left* operation.

`N_PUT_RIGHT`: the number of threads that execute a *put_right* operation.

`N_GET_LEFT`: the number of threads that execute a *get_left* operation.

`N_GET_RIGHT`: the number of threads that execute a *get_right* operation.

For example, the program should spawn `N_GET_LEFT` threads, uniquely identified through a parameter *self*, each of which executes *get_left*(*d*) on the deque *d*. Before the operation, the thread should print its intention to execute the operation. After the operation, the thread should print that the operation has finished and what the return value was (not needed for the `put` operations). We recommend that *put_left* threads put the tuple (*self*, `"left"`) on the queue, while *put_right* threads put the tuple (*self*, `"right"`) on the queue (instead of just *self*).

Use the harmony flags `-o deque.hfa` and `-B deque.hfa` to compare the behaviors of the deque specification and the implementation. That is, you might run, for example:

```
harmony -c N_PUT_LEFT=2 -c N_PUT_RIGHT -o deque.hfa deque_test.hny
harmony -c N_PUT_LEFT=2 -c N_PUT_RIGHT=2 -B deque.hfa -m deque=deque_impl deque_test.hny
```

You can try different numbers of threads of each type, but beware that model checkers suffer from *state explosion*, and thus going much beyond 6 threads total may not be feasible. (You can also, if you like, pre-populate your test deque before starting the threads.)

Feedback to Students

Students receive the output from harmony as well as the following README file:

```
For grading, we did the following:

We ran 4 tests for the deque specification, the same 4 tests for the
deque implementation, and 4 tests for the deque test program.  They are
as follows:

deque0:  Here's the simple sequential test program:

    from deque import Deque, get_left, put_left

    d = Deque()
    put_left(?d, 0)
    let v = get_left(?d):
        assert v == 0, "expected 0"

    It doesn't run anything concurrently.  It just checks to see if
    put_left() and get_left work as specified.

deque1:  A slightly more complicated but still sequential test program:

    from deque import Deque, get_left, get_right, put_left, put_right

    d = Deque()
    put_left(?d, 0)
    put_left(?d, 1)
    put_right(?d, 2)
    let v = get_left(?d):
        assert v == 1, "4: expected 1"
    let v = get_right(?d):
        assert v == 2, "5: expected 2"
    let v = get_left(?d):
        assert v == 0, "6: expected 0"
```

```
    let v = get_right(?d):
        assert v == None, "7: expected None"
    let v = get_left(?d):
        assert v == None, "8: expected None"
    put_left(?d, 0)
    put_left(?d, 1)
    put_right(?d, 2)
    let v = get_left(?d):
        assert v == 1, "12: expected 1"
    let v = get_right(?d):
        assert v == 2, "13: expected 2"
    let v = get_left(?d):
        assert v == 0, "14: expected 0"
    let v = get_right(?d):
        assert v == None, "15: expected None"
    let v = get_left(?d):
        assert v == None, "16: expected None"

deque2: This is run against a correct specification using behaviors.  It
    uses 1 put_left, 1 put_right, 1 get_left, and 1 get_right.

deque3: This is run against a correct specification using behaviors.  It
    uses 1 put_left, 2 put_rights, 2 get_lefts, and 1 get_right.
    Moreover, it populates the deque first:

        deque.put_left(?thedeque, "L")
        deque.put_right(?thedeque, "R")

deque_impl0, deque_impl1, deque_impl2, deque_impl3: same as
    deque0, deque1, deque2, and deque3, except that they are
    run against the implementation rather than the specification.

deque_test0: we ran your test program against a correct implementation
    of a concurrent deque using 1 put_left, 1 put_right, 1 get_left,
    and 1 get_right.

deque_test1: we first ran your test program against a correct
    implementation of a concurrent deque using 1 put_left,
    1 put_right, 1 get_left, and 1 get_right to create a .dfa file
    We then ran it against a broken implementation that reverses
    put_left and put_right.  Note that deque_test1 will fail if
    the recommendations in the assignment file are not followed---it
    leads to a weaker test.

deque_test2: same as deque_test1 but does 2 put_lefts.
```

```
deque_test3: we first ran your test program against a correct
    implementation of a concurrent deque using 1 put_left,
    1 put_right, 1 get_left, and 1 get_right to create a .dfa file
    We then ran it against a broken implementation that does not
    delete the item in get_left().

The output of each of these test is available online in the zip file
whose URL is available in the comments section of your submission on
CMSX.  For each test you should find 3 files:

    x.status: "no submission", "compile error", "correct", "wrong"
    x.out:  standard output of Harmony
    x.out2: error output of Harmony

If there are two phases of running harmony, that is indicate using
_p0 and _p1 file name extensions.

The scoring is as follows:  no submission or compile error
(does not compile) is 0 points.  For the specification and
test, a correct run is worth 5 points and a faulty run is
worth 2 points.  For the implementation, a correct run is
worth 10 points and a faulty run 4 points.

One exception is the following: if deque_test0 fails, then the
remaining deque_test runs are only worth 2 points even if they
"succeed".  The reasoning there is that if the test program
fails for a correct implementation, it probably will also "fail"
for incorrect implementations and should not get full points.
```

Example Project: Blocking Deque

This project is a variant of the previous double-ended queue (deque) programming assignment but involves blocking operations. The deque should have a maximum number of elements. Trying to add elements to a full deque should wait until the deque is no longer full. This project also add new (blocking) operations to peek at the elements on both sides of the deque.

Specification.

Start with a specification, which must have the following methods:

> `Bdeque`(n): returns the initial value of an empty bdeque with at most n elements.
>
> *put_left*(d, v): d points to a bdeque and v is some value. Put v at the left-end of the bdeque. Does not return a value. Should block if the bdeque is full.
>
> *put_right*(d, v): d points to a bdeque and v is some value. Put v at the right-end of the bdeque. Does not return a value. Should block if the bdeque is full.

get_left(d): d points to a bdeque. If the bdeque is empty, the operation should wait. Otherwise it should remove and return the value at the left-end of the bdeque.

get_right(d): d points to a bdeque. If the bdeque is empty, the operation should wait. Otherwise it should remove and return the value at the right-end of the bdeque.

peek_left(d): d points to a bdeque. If the bdeque is empty, the operation should wait. Otherwise it should return the value at the left-end of the bdeque.

peek_right(d): d points to a bdeque. If the bdeque is empty, the operation should wait. Otherwise it should return the value at the right-end of the bdeque.

Place the entire specification in file `bdeque.hny`. Note that all operations except for `Bdeque`(n) should be atomic. Please make sure the operations are named *exactly* as above, or the autograder will not be able to find them.

Implementation.

The bdeque *implementation*, which you should put in file `bdeque_impl.hny`, should define the same methods but cannot use the **atomically** or **sequential** keywords and should instead rely on `Lock`, `acquire`, `release`, `Condition`, `wait`, `notify`, and `notifyAll` from the `synch` module (which the `bdeque_impl.hny` file should **import**).

The implementation should *not* use a linked list, and should *not* import the `malloc` module. Instead, the elements of the deque should be a fixed size array, and you should keep track of the left and right indexes into this array. If the size of the array is n (which is the argument to `Bdeque`), the elements of the array can be initialized as [**None**,] * n .

You should use a lock and condition variables for the implementation. Ideally, no threads should be notified (woken up) unless they can complete the operation. So while it is possible to implement this assignment with a single condition variable, it would be better to use a condition variable per specific waiting condition.

Testing.

You also should create a test program, in file `bdeque_test.hny`, for your bdeque. Note that the test should be a *black box* test—it cannot look inside the implementation of the bdeque. *We should be able to run your test program with somebody else's implementation of the bdeque.* So, you should not deference the bdeque pointer to see what's in it—you can only use the bdeque interface in your test program.

Testing should be based on comparing behaviors between the specification and the implementation (a form of *differential testing*). Model your test program after the `queue_btest1.hny` test program from the Harmony book. It should import **bdeque**, *not* **bdeque_impl**. It should allocate a single bdeque using `Bdeque(SIZE)`. The program should define the following constants. in the program:

`SIZE` = 2: the size of the bounded buffer.

`N_PUT_RIGHT` = 2: the number of threads that execute a *put_right* operation.

`N_PUT_LEFT` = 1: the number of threads that execute a *put_left* operation.

`N_GET_LEFT` = 1: the number of threads that execute a *get_left* operation.

`N_GET_RIGHT` = 1: the number of threads that execute a *get_right* operation.

`N_PEEK_LEFT` = 0: the number of threads that execute a *peek_left* operation.

`N_PEEK_RIGHT` = 1: the number of threads that execute a *peek_right* operation.

(If there are peek operations, you'll want to total number of put operations to exceed the total number of get operations so the peek operations do not get stuck.) For example, the program should spawn `N_GET_LEFT` threads, uniquely identified through a parameter *self*, each of which executes *get_left*(*d*) on the bdeque *d*. Before the operation, the thread should print its intention to execute the operation. After the operation, the thread should print that the operation has finished and what the return value was (not needed for the `put` operations). We recommend that *put_left* threads put the tuple (*self*, "left") on the queue, while *put_right* threads put the tuple (*self*, "right") on the queue (instead of just *self*).

Use the harmony flags `-o bdeque.hfa` and `-B bdeque.hfa` to compare the behaviors of the bdeque specification and the implementation. That is, you might run, for example:

```
harmony -c N_PUT_LEFT=1 -c N_PUT_RIGHT=2 -o bdeque.hfa deque_test.hny
harmony -c N_PUT_LEFT=1 -c N_PUT_RIGHT=2 -B bdeque.hfa -m bdeque=deque_impl deque_test.hny
```

You can try different numbers of threads of each type, but beware that model checkers suffer from *state explosion*, and thus going much beyond 6 threads total may not be feasible. (You can also, if you like, pre-populate your test bdeque before starting the threads.)

Feedback to Students

Students receive the output from harmony as well as the following README file:

```
For grading, we did the following:

We ran 4 tests for the bdeque specification, the same 4 tests for the
bdeque implementation, and 4 tests for the bdeque test program.  They are
as follows:

bdeque0:  Here's the simple sequential test program:

    from bdeque import *

    d = Bdeque(2)
    put_left(?d, 0)
    let v = get_left(?d):
        assert v == 0, "expected 0"

    It doesn't run anything concurrently.  It just checks to see if
    put_left() and get_left work as specified.
```

bdeque1: A slightly more complicated but still sequential test program:

```
    from bdeque import *

    d = Bdeque(3)
    put_left(?d, 0)
    put_left(?d, 1)
    put_right(?d, 2)
    let v = get_left(?d):
        assert v == 1, "4: expected 1"
    let v = get_right(?d):
        assert v == 2, "5: expected 2"
    let v = peek_left(?d):
        assert v == 0, "6: expected 0"
    let v = peek_right(?d):
        assert v == 0, "7: expected 0"
    let v = get_left(?d):
        assert v == 0, "8: expected 0"
    put_left(?d, 0)
    put_left(?d, 1)
    put_right(?d, 2)
    let v = get_left(?d):
        assert v == 1, "12: expected 1"
    let v = get_right(?d):
        assert v == 2, "13: expected 2"
    let v = peek_left(?d):
        assert v == 0, "14: expected 0"
    let v = peek_right(?d):
        assert v == 0, "15: expected 0"
    let v = get_left(?d):
        assert v == 0, "16: expected 0"
```

bdeque2: This is run against a correct specification using behaviors. It
    uses SIZE=2, 1 put_left, 1 put_right, 1 get_left, and 1 peek_right.

bdeque3: This is run against a correct specification using behaviors. It
    uses 1 put_left, 0 put_rights, 1 get_left, 1 get_right, 1 peek_left,
    and 1 peek_right. Moreover, it populates the deque first:

```
        bdeque.put_left(?thedeque, "L")
        bdeque.put_right(?thedeque, "R")
```

bdeque_impl0, bdeque_impl1, bdeque_impl2, bdeque_impl3: same as
    bdeque0, bdeque1, bdeque2, and bdeque3, except that they are
    run against the implementation rather than the specification.

```
bdeque_test0: we ran your test program against a correct implementation
    using SIZE=2 of a concurrent bdeque using 1 put_left, 1 put_right,
    1 get_left, and 1 peek_right.

bdeque_test1: we first ran your test program against a correct
    implementation of a concurrent bdeque with SIZE=2 using 1 put_left,
    1 put_right, 1 get_right, 1 peek_left, and 1 peek_right to create a
    .dfa file.  We then ran it against an implementation that doesn't
    wait but returns None if the bdeque is empty.

bdeque_test2: we first ran your test program against a correct
    implementation of a concurrent bdeque with SIZE=1 using 1 put_left,
    1 put_right, 1 get_left, and 1 get_right to create a .dfa file.
    We then ran it against an implementation that internally uses a
    size of 2 instead of 1.

bdeque_test3: we first ran your test program against a correct
    implementation of a concurrent bdeque with SIZE=2 using 1 put_left,
    1 peek_left, and 1 peek_right to create a .dfa file.  We then ran it
    against a broken implementation that only notifies one peeker on
    a put operation.

The output of each of these test is available online in the zip file
whose URL is available in the comments section of your submission on
CMSX.  For each test you should find 3 files:

    x.status: "no submission", "compile error", "correct", "wrong"
    x.out:  standard output of Harmony
    x.out2: error output of Harmony

If there are two phases of running harmony, that is indicate using
_p0 and _p1 file name extensions.

The scoring is as follows:  no submission or compile error
(does not compile) is 0 points.  For the specification and
test, a correct run is worth 5 points and a faulty run is
worth 2 points.  For the implementation, a correct run is
worth 10 points and a faulty run 4 points.

One exception is the following: if bdeque_test0 fails, then the
remaining bdeque_test runs are only worth 2 points even if they
"succeed".  The reasoning there is that if the test program
fails for a correct implementation, it probably will also "fail"
for incorrect implementations and should not get full points.

2 bonus points each are given for functionally correct implementations
```

```
that satisfy the following two additional requirements:

    tmn: the code does not notify too many threads.  That is, a get
         should only notify one putter, a peek should not notify
         anybody, and a put should notify at most one getter.
    tfc: the code uses separate condition variables for getters, putters,
         and peekers
```