**CS162, Spring 2004** **Discussion #5** **Amir Kamil**
**UC Berkeley** **2/19/04**

**Topics: Testing**

# 1 Announcements

- Design reviews next week. All group members must attend! You all will lose points if anyone is missing. Sign up for design reviews on the main CS162 website.
- Project 1 code due Thursday, March 4th, 2004, 11:59pm (only 2 weeks! start now)
- Midterm on Thursday, March 18th, 2004, 5:30-7:30pm, 155 Dwinelle
- Clarification on "unspecified behavior":
  When the behavior of a method call is unspecified, you can do whatever you want. For example, in `join()`, if thread B joins thread C after A has already done so, the behavior for B is unspecified. So you can implement it such that B returns immediately, after C finishes, or never. It is up to you.

# 2 Testing

Testing is one of the most important but overlooked components of software engineering. Microsoft has two testers for each coder. Writing code is easy, making sure it works correctly is hard. For Nachos, testing your code thoroughly is essential. Your code will be graded automatically, and only a small subset of the tests will be released in advance, so you will have to write most of your tests. In order to help you do so, we will review the basics of testing.

## 2.1 Types of Tests

Recall that there are two main types of tests:

1. A *glass box test* is one which is written with some knowledge of the implementation of a piece of code. Glass box tests should cover all possible branches of execution. For example, suppose we have a function `foo(int)`, and that this function is called most often on small inputs. Suppose we implement it such that memoizes the results for inputs less than 1000, but computes the results for larger inputs. For glass box testing, we would test inputs larger than and smaller than 1000. Note that glass box testing can include tests and sanity checks internal to the code (e.g. `repOk()` for those of you who took CS61B with Clancy). For example, if an alarm implementation uses a sorted linked list, then the list should be tested or printed out on each call to make sure it remains ordered. These internal tests should be turned off when you are convinced your code works. Use a `DEBUG` flag to turn on and off these tests.
2. A *black box test* is one which only tests the given interface and specifications. Black box tests also check for corner cases. For example, a corner case while testing the priority scheduler might be with one or no threads running. Another corner case might be when all the thread have the same priority. These cases tend to be ones that you overlook while coding, so they are often great to test. You should attempt to cover all possibilities in your black box tests. First specify all possible inputs to your code, group them into similar cases, and a write a test for each case.

As a simple, concrete example, let's write tests for the producer/consumer code in figure 1.

1. Glass box tests:
   There aren't too many glass box tests possible, since there aren't very many execution paths. But here are a few:
   - test different initial values for `numCokes`, especially `numCokes == 0` and `numCokes == MAXCOKES`

```
Lock lock;
Condition wantToAdd = new Condition(lock);
Condition wantToTake = new Condition(lock);
int numCokes;
static final int MAXCOKES;

Producer() {
  lock.acquire();
  while (numCokes == MAXCOKES) {
    wantToAdd.sleep();
  }
  numCokes++;
  wantToTake.wake();
  lock.release();
}

Consumer() {
  lock.acquire();
  while (numCokes == 0) {
    wantToTake.sleep();
  }
  numCokes--;
  wantToAdd.wake();
  lock.release();
}
```

Figure 1: The producer/consumer code.

  - set the hardware timer interrupt to different values, to test different interleaving of operations

2. Black box tests:
    - test equal number of producers and consumers
    - test different number of producers and consumers
    - test different orders of calls to `produce()` and `consume()` (e.g. all producers call `produce()` and then all consumers call `consume()`, vice versa, and producers and consumers exchange calls)
    - test the corner cases - no consumers and multiple producers, no producers and multiple consumers

As a more involved example, let's take a look at a square matrix multiplication function. Though it is not completely relevant, we will first go over the algorithm used in the function in order to provide some context when looking at it. It is not at all important to understand the algorithm.

Recall the basics of matrix multiplication: in $C = A \times B$, cell $C(i, j)$ is equal to the product of row $A(i, :)$ and column $B(:, j)$, as illustrated in figure 2.

One of the nice things about matrix multiplication is that each matrix can be divided into sub-matrices, or *blocks*. The blocked matrices can then be multiplied as before, treating each block as a single entry but doing a recursive matrix multiply when multiplying blocks. Figure 3 illustrates this procedure.

A complication arises when blocking, however. What if the matrix size is not a multiple of the block sizes? There are leftover pieces of the matrix that do not compose any blocks. We can avoid this problem, however, by *padding* the matrices, adding rows and/or columns of zeros until the size is a multiple of the block sizes as illustrated in figure 4.

Now let's examine the actual matrix multiply function. We will only look at the top-level function for multiplying, which does the padding, and assume the function that actually does the multiplication works correctly. The relevant code is given in figure 5. What glass and block box tests can we write for this

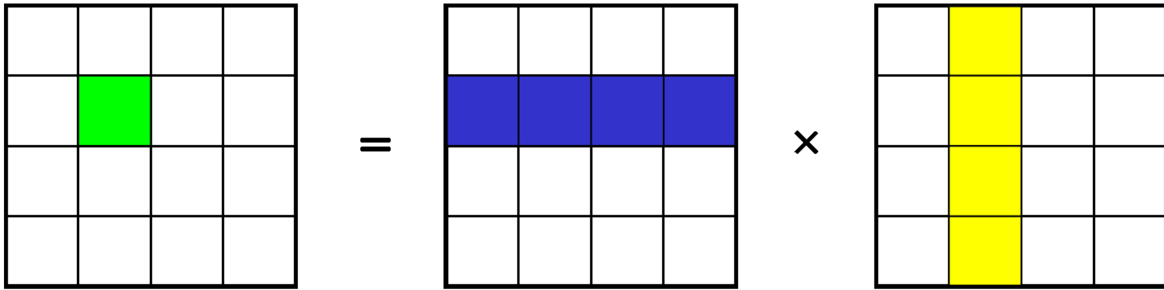

Figure 2: Simple square matrix multiply.

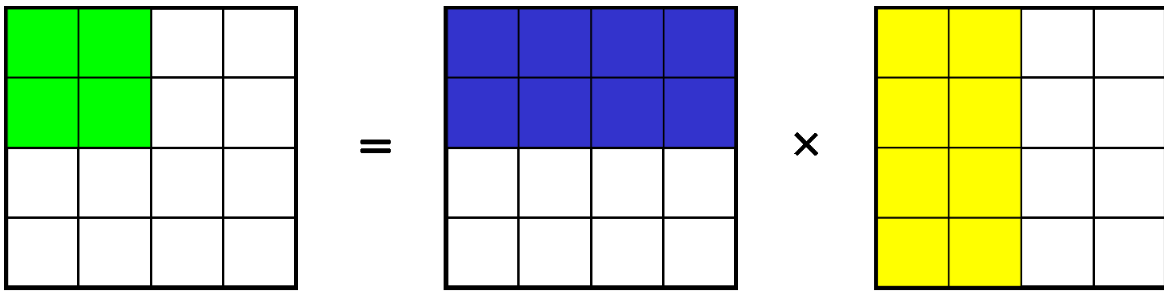

Figure 3: Blocked square matrix multiply.

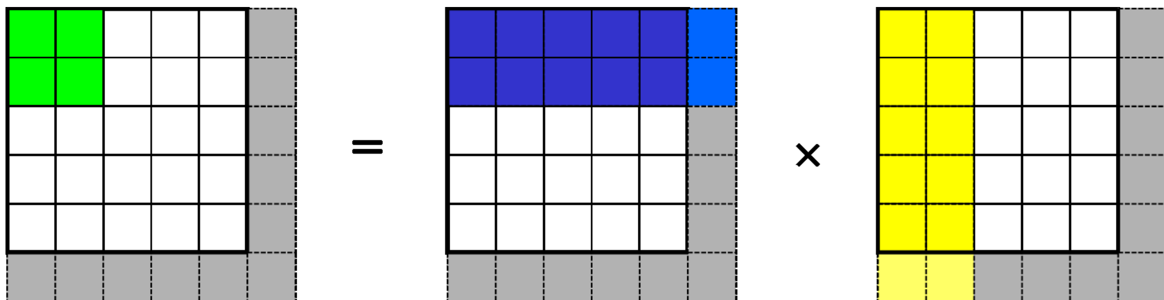

Figure 4: Padded and blocked square matrix multiply.

```
const int M, N, K; // block sizes

// actually does matrix multiplication
private void do_mult(int, double *, double *, double *);

// Multiplies matrices A and B, leaving the result in C.
// REQUIRES: A, B, and C are size * size in length
// MODIFIES: C
public void mult(const int size, const double *A, const double *B, double *C) {
  if (size % M == 0 && size % N == 0 && size % K == 0)
    do_mult(size, A, B, C); // no padding required
  else {
    // padding required
    size2 = least common multiple of M, N, K greater than size
    // allocate memory
    A2 = new double[size2^2];
    B2 = new double[size2^2];
    C2 = new double[size2^2];
    // copy to temporary arrays
    copy A to A2 with padding  // see illustration
    copy B to B2 with padding
    set C2 to all 0's
    // do multiply
    do_mult(size2, A, B, C);
    // copy answer to original array
    copy C2 to C minus padding
    // free memory
    delete A2, B2, C2
  }
}
```

Figure 5: Matrix multiply padding code.

function? Note that the actual specification for the function gives no mention of blocking or padding; all the user knows is that it multiplies matrices.

- Glass box tests:
  1. test matrices whose size is a multiple of the block sizes `M`, `N`, and `K`
  2. test matrices whose size is a non-multiple of one of, two of, and all of `M`, `N`, and `K`
  3. try different block sizes
  4. call the matrix multiply function numerous times, ensuring that there are no memory leaks
  5. try matrices smaller and largen than the block sizes `M`, `N`, and `K`
- Black box tests:
  1. try corner cases: 0x0 matrix, 1x1 matrix
  2. try small and large matrices
  3. try random matrices

## 2.2 Design Document Requirements

On you design documents, you are required to describe your test cases. You should write two to four tests per part, though you may need more for the more difficult parts. Write both black and glass box tests, and make sure that your tests cases are distinct. For example, consider the following tests for the priority scheduler:

- Run three threads, A, B, and C with priorities 1, 2, and 3. Make sure they run in order.
- Run five threads, A, B, C, D, and E with priorities 2, 3, 5, 6, and 7. Make sure they run in order.

These test cases, while they might be useful in your testing, are not very distinct. They have a different number of threads and different priorities, but almost any scheduler that passes the first test will pass the second. On your design document, identify similar cases like these and reduce them to one case. The following test description covers both of the above cases:

*Run some number of threads with distinct priorities and ensure that they run in the correct order.*

In addition to the test descriptions, give the desired results of your tests, so that the purpose of each test is clear to the reader.