Background

The purpose of this assignment is to provide additional experience with concurrent programming and semaphores. We will use the BACI system, since it provides a relatively simple way to experiment with these techniques.

Exercise 0. Download and install the BACI system. See

```
http://inside.mines.edu/~tcamp/baci/baci_index.html#Java
```

for details. Once there, download the JavaBACI classes and follow the instructions found there. The BACI `C--` compiler guide, available at the same site, may also be of interest. Place a copy of the `javabaci` script in your `bin` directory.

Exercise 1. Examine the code found in the programs `ab.cm` and `ab-serialized.cm`. Observe how multiple processes are created and how semaphores are introduced and used. *Predict the output of each of these two programs.*

(a) Compile and execute `ab.cm` to verify your prediction. To compile `ab.cm`, issue the following command:

```
javabaci bacc ab
```

(Notice that the `.cm` file extension is not required.) The compilation will produce a listing file, `ab.lst`, and a *pcode* file suitable for interpretation, named `ab.pco`. Had there been syntax errors, the pcode file would not have been generated and error messages would have been produced instead. To interpret the pcode file, give this command:

```
javabaci bainterp ab
```

(Notice that the `.pco` extension is not required.) Run the program a few times.

(b) Repeat, with `ab-serialized.cm`. Run the compiled program several times. Make sure that you understand how the semaphore ensures the serialization you are observing here.

Exercise 2. Examine the code in `abaa.cm`. Predict the output, then compile and run it. Make a copy of this program, naming it `abaa-serialized.cm`. By introducing one or more semaphores, modify this program so the processes terminate in the order A (any copy), B, A (any copy), A. Run your program several times to see if the desired order is obtained. *Include comments near the top of your program which explains why the processes will terminate in this order.*

Exercise 3. The code in `producer-consumer.cm` provides a skeleton solution to the producer-consumer problem — also known as the *bounded buffer problem*. To illustrate the problem, we will adopt a very modest overall goal: namely, to format a collection of integers as a sequence of rows, with three values per row.

To achieve this overall effect, we will let the producer process generate the values in a given range, depositing them in a buffer. The consumer process will simply remove these values from the buffer and display them, three per row.

How can the consumer process know when all of the data values have been seen? One way is to terminate the stream of values with a known *sentinel* value, such as -1. Placing this sentinel value in the buffer will be the responsibility of the producer.

Using the solution discussed in class as a guide, complete this program so that the producer and consumer are properly synchronized. Do not change any of the constant values or introduce any additional global variables. Similarly, you should not need any additional semaphores beyond those which appear in the program. In the traditional solution to the producer-consumer problem, the processes are considered to be infinite processes. Here, of course, we want both the producer and consumer to be finite.

Exercise 4. In the solution to the producer-consumer problem, consider the effect of multiple producers. For example, suppose the processes created in the main body is now written as:

```
cobegin {
   producer(10, 19);
   producer(20, 29);
   producer(30, 39);
   producer(40, 50);
   consumer();
}
```

For the net effect, we expect the consumer process to see (in some order) the values 10 through 50. As before, the values should be written to the standard output file, three per line.

Make a copy of `producer-consumer.cm`, naming it `multi-producer-consumer.cm`. Make the necessary changes to the program so that all processes are properly synchronized.

You may find it helpful to introduce a new constant, declared like:

```
const int NR_PRODUCERS = 4;
```

What to Submit

Put copies of your programs into a folder, then drag this folder onto the EIU submit icon. Include the source code and compiled versions of each of your solutions. Be sure that your name appears in each program as a comment somewhere near the top of the file. Your programs should be generously commented.

Source Code

```
1  // MAT 4970
2  //
3  // File:    ab.cm
4  // Author: Bill Slough
5  //
6  // Purpose: This program creates two processes, A and B.
7  //
8
9  const int TRIALS = 20;
10
11 void A() {
12    cout << "A";
13 }
14
15 void B() {
16    cout << "B";
17 }
18
19 void main() {
20     int i;
21     for (i = 0; i < TRIALS; i++) {
22        // Start two concurrent processes
23        cobegin {
24            A();
25            B();
26        }
27        cout << endl;
28     }
29  }
```

Listing 1: Two processes; no serialization.

```
1  // MAT 4970
2  //
3  // File:    ab-serialized.cm
4  // Author: Bill Slough
5  //
6  // Purpose: This program creates two processes, A and B, and ensures that A runs to completion
7  //          before B begins.
8  //
9
10 const int TRIALS = 20;
11 semaphore a_finished;
12
13 void A() {
14    cout << "A";
15    signal(a_finished);
16 }
17
18 void B() {
19    wait(a_finished);
20    cout << "B";
21 }
22
23 void main() {
24     int i;
25     for (i = 0; i < TRIALS; i++) {
26        // Initialize the semaphore
27        initialsem(a_finished, 0);
28
29        // Start two concurrent processes
30        cobegin {
31            A();
32            B();
33        }
34        cout << endl;
35     }
36 }
```

Listing 2: Two processes; with serialization. This example shows how to create a semaphore, initialize it, and use the wait() and signal() functions to manipulate it.

```
1  // MAT 4970
2  //
3  // File:    abaa.cm
4  // Author: Bill Slough
5  //
6  // Purpose: This program creates four processes, but they are not serialized.
7  //
8
9  const int TRIALS = 20;
10
11 void A() {
12    cout << "A";
13 }
14
15 void B() {
16    cout << "B";
17 }
18
19 void main() {
20     int i;
21     for (i = 0; i < TRIALS; i++) {
22        // Start four concurrent processes
23        cobegin {
24            A();
25            A();
26            A();
27            B();
28        }
29        cout << endl;
30     }
31  }
```

Listing 3: Four processes; no serialization.

```
1  // MAT 4970
2  //
3  // File:    producer-consumer.cm
4  // Author:
5  //
6  // Purpose: Illustrates a semaphore-based solution to the producer-consumer problem
7  //
8
9  const int SENTINEL = -1;
10 const int BUFFER_SIZE = 5;
11
12 // Data structure for a circular buffer
13 int buffer[BUFFER_SIZE];    // slots for buffer entries
14 int in;                     // index of next available slot
15 int out;                    // index of oldest entry
16 int size;                   // number of elements in the buffer
17
18 semaphore mutex;    // allow at most one process access to the shared buffer
19 semaphore full;     // number of occupied buffer slots
20 semaphore Empty;    // number of empty buffer slots
21
22 void enqueue(int item) {
23    // Add an item to the buffer
24
25    //  ...INCOMPLETE...
26 }
27
28 int dequeue() {
29    // Removes an item from the buffer
30
31    // ... INCOMPLETE ...
32 }
33
34 void producer(int low, int high) {
35    //
36    // Generates integers in the range [low, low + 1, ..., high] and places
37    // them in the shared buffer, to be removed by the consumer.
38    //
39
40    // ... INCOMPLETE ...
41 }
42
43 void consumer() {
44    //
45    // Removes all values found in the shared queue and displays them on
46    // the standard output file, three to a line.  The appearance of the
47    // SENTINEL value in the queue indicates no further values will be
48    // generated.
49    //
50
51    // ... INCOMPLETE ...
52 }
53
54 void main() {
55    // Initialize the semaphores
56       // ... INCOMPLETE ...
57
58    // Create an empty buffer queue
59       // ... INCOMPLETE ...
60
61    // Start the producer and consumer processes
62    cobegin {
63       producer(10, 50);
64       consumer();
65    }
66 }
```

Listing 4: Skeleton solution to the producer/consumer problem.