# CSC 335: Project 2 – Fix that Code
# Due Monday, May 6, 2024 at start of class

**Objectives:**

- To practice with concurrency controls: *semaphores* and *monitors*
- To understand *locks* and *condition variables*
- To implement a safe bounded buffer
- To continue learning the Nachos thread system
- To fix your broken Project 1...

**Your Mission**
Now that you have broken your code, it's time to implement concurrency controls to fix the problems. Locks, semaphores, and semaphore-based condition variables have already been implemented for you in NACHOS, but there are no monitors yet. In this project, you will remedy that situation by turning your doubly-linked-list into a monitor, which will fix the concurrency problems from Project 1. You will also implement a bounded buffer monitor and then implement your own condition variables.

**Make a backup first**
Just like last project, copy your entire finished Project 1 into a separate place before starting. Then use the copy as the starting point for this project. That way, if something goes horribly wrong, you have backup files from a known starting point.

**Part 1: Repair Project 1**
In the **threads** directory in NACHOS, you will find already-working versions of **Semaphore**, **Lock**, and **Condition** (variables). Study them to see how they do the logic we've discussed in class. Then, use the Lock and Condition classes to turn your DLList class into a monitor. That is, modify the methods so that (1) multiple threads can access a shared list without causing concurrency errors and (2) underflow is prevented (i.e. removeHead will now always remove something instead of ever returning null.) You will do both of these by adding locking, waiting, and signaling calls at the right places. NACHOS was nice enough to provide a protected list data structure called **SynchList** so you can see how the **Lock** and **Condition** classes are used. Look at it to help you write the code.

Rerun the two failing interleavings from Project 1 to make sure you have fixed all of your concurrency problems. Document your fixes in your write-up by summarizing each original problem and then showing how your code now prevents the errors (pasting in the corrected output for this part is fine).

**Part 2: Bounded Buffer**
Implement a bounded buffer monitor based on slides 87-88 from class. Here are the required methods:

```
// non-default constructor with a fixed size
public BoundedBuffer(int maxsize)

// Read a character from the buffer, blocking until there is a char
// in the buffer to satisfy the request. Return the char read.
public char read()

// Write the given character c into the buffer, blocking until
// enough space is available to satisfy the request.
public void write(char c)

// Prints the contents of the buffer; for debugging only
public void print();
```

To test your protected bounded buffer, add a few test cases to KThread (where all your Project 1 tests are). Be sure to test cases with multiple threads that would have caused problems if the monitor was not in place. Document these new tests (and what the errors would have been) in your write-up. *Be sure to test your wait and signal calls by forcing what would normally be underflow and overflow without the protection!*

**Part 3: Your own condition variables**
Finally, implement your own condition variables by finishing the `sleep`, `wake`, and `wakeAll` methods in the **Condition2** class. Unlike the Condition class, which is semaphore-based, this new implementation will use the enabling/disabling of interrupts to provide atomicity. Do not use semaphores. To get you started, here's the pseudocode for `sleep`:

- release the lock
- disable interrupts
- add the current thread to a queue
- block the current thread
- enable interrupts (when the thread gets woken up)
- reacquire the lock

You'll need a queue to get this done. Luckily, the **SynchList** class you looked at in Part 1 is exactly what you need.

**PITFALL ALERT**: Remember, both `wake` and `wakeAll` will only wake up a thread if there is one on the queue to awaken. Otherwise, they do nothing.

Once implemented, test your code by changing all of your tests from Parts 1 and 2 of this project to use **Condition2** objects instead of **Condition** objects. The behavior should be identical to the original. Document these in your writeup by pasting in the output of your tests.

**Turning it in**

Place (1) your writeup and (2) all your **modified** code into a **single** pdf file. Put the file into your "nachos" project folder and zip it up. Upload the zip file to Nexus.

**Grading**

This project is worth 50 points.

- 20 points for the correct implementation of the Condition2 version of condition variables
- 10 points for a correct fix of your doubly-linked list
- 10 points for a correct implementation of bounded buffer
- 10 points for the writeup. The harder it is for me to understand what your test cases are doing, the fewer points you will receive.

**Gentle Reminder**

Programming projects are *individual* projects. I encourage you to talk to others about the general nature of the project and ideas about how to pursue it. However, the technical work, the writing, and the inspiration behind these must be substantially your own. You must cite anyone else who contributes in any way to the project by adding appropriate comments to the code. Similarly, if you include information that you have gleaned from other sources, you must cite them as references. Looking at, and/or copying, other people's code is inappropriate, and will be considered an honor code violation.