PROJECT 2 – IMPLEMENTING AND USING MONITORS

CSc 335, Spring 2011
Assigned: Tuesday, Week 4
Due: Monday, Week 5

This project is based primarily on a project from [1].

GOALS

In this project, you will:

1. Familiarize yourself some more with the OS/161 operating system code.
2. Implement locks and condition variables in OS/161.
3. Fix the concurrency problems you uncovered in Project 1.
4. Implement a solution to the bounded-buffer producer/consumer problem.

ACTIVITIES TO UNDERTAKE

IMPLEMENTING LOCKS AND CONDITION VARIABLES (40%)

1. Find synch.h. This defines three data types: semaphore, lock, and cv.
   semaphore provides counting semaphores, and is already implemented. lock and cv implement
   locks and condition variables – you will implement these.
2. Look at synchtest.c to see how lock and cv are used.
3. Implement the lock construct by adding appropriate fields to synch.h and implementing the ap-
   propriate functions in thread/synch.c. Use spinlocks and wait channels – you cannot disable
   interrupts. Do not change the public interface because locks are used by other parts of OS/161.
4. Implement the cv functions by adding data members to synch.h and implementing the functions
   in synch.c to provide Mesa-style condition variables. Use wait channels – You cannot disable in-
   terrupts. Do not change the public interface because condition variables are used by other parts of
   OS/161.
   Note: The interface is different from that given in your book. In the book, monitors have a mutex
   lock and conditions use the lock. In our implementation, the client must pass the lock into the cv
   operations.
   Note: While the code for condition variables need not be lengthy, it will require some thought because
   it is subtle. In particular, you must guarantee that a call to signal cannot affect a subsequent call to
   wait.
5. Make a directory to contain output of tests: src/kern/submitProj2.
6. Run the already provided tests to verify that your lock and condition variable implementations are
   working. The tests sy1, sy2, and sy3 test semaphores, locks, and condition variables, respectively.
   Use script to record the output of running these tests in src/kern/submitProj2/synchtests.txt.
Fixing Project 1 (20%)

1. Modify your `linkedlist.c` implementation of linked lists from Project 1 so that it ensures that the list is updated consistently, even when multiple threads are accessing it. Essentially, this is turning your list into a monitor.

2. Rerun your tests from Project 1 to ensure that you have fixed the problems. Use `script` to record the output of running these tests in `src/kern/submitProj2/linkedlisttests.txt`.

3. Remove calls to `splhigh()` in `linkedlisttest.c` code and rerun the tests. Use `script` to record the output of running these tests in `src/kern/submitProj2/linkedlisttests-withinterrupts.txt`.

Solving the Bounded-Buffer Problem (40%)

Implement a bounded buffer with the following public interface (you may change the `struct` definition, but you should keep the same public operations):

```c
struct bounded_buffer {
    char * buf_Array;
    int buf_size;
};

struct bounded_buffer *
bbuffer_create(int maxsize);

bbuffer_destroy(struct bounded_buffer *buf);

// Read a character from the buffer, blocking until there is a char
// in the buffer to satisfy the request. Copy the char into already
// allocated memory location c.
void bbuffer_read(struct bounded_buffer *buf, char *c);

// Write a character c into the buffer, blocking until enough space
// is available to satisfy the request.
void bbuffer_write(struct bounded_buffer *buf, char c);
```

1. Implement this data structure in two files: `bbuffer.h` will contain the declarations, while `bbuffer.c` will contain the definitions (implementations). The implementation should use locks and conditions to ensure thread safety. You should not use semaphores directly.

2. Add a test case, in `bbuffertest.c`, to test your bounded buffer. This test should create multiple threads that call `bbuffer_read` and `bbuffer_write` on a shared `bounded_buffer`.

3. Make a copy of the `LINKED_LIST` config file and modify the copy so that your kernel will include `bbuffer.c` and `bbuffertest.c`.

4. Modify `menu.c` to call your test when the user types `bbt`.

5. Using this test case, test and debug your bounded buffer implementation. When it is working, use `script` to show the output of running this test in `src/kern/submitProj2/bbuffertest.txt`.

What to Turn In and How

You should turn in a hard-copy of a written report, which contains:

18 April 2011
• Source code for all code you have written for the assignment.

You should also electronically submit your copy of the OS/161 code, including the directory containing:

• Output, generated by script, that shows the results of all of your working tests:
  - synchtests.txt
  - linkedlisttests.txt
  - linkedlisttests-withinterrupts.txt
  - bbuffertest.txt

Submit the directory with:

```
/export/faculty/csc335pub/submit csc335 <dirname>/kern
```

where `<dirname>` is the path to your directory that is a copy of the `src` directory of OS/161. The submit script will make a copy of your directory and make sure that nobody can read or modify this submitted copy.

**REFERENCES**