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

**GOALS**

In this project, you will:

1. Familiarize yourself with the OS/161 operating system code.
2. Familiarize yourself with C, if needed.
3. Learn about the ways in which concurrency can cause programs to behave incorrectly.

**CREATE A SORTED DOUBLY-LINKED LIST IN C FOR OS/161 (25%)**

Due Tuesday of Week 2, in class.

**GET OS/161**

Give me a copy of your rsa public key. Once I’ve used that to give you permission to our git repository, you should be able to use git to get a copy of the kernel source:

```
git clone git@antipasto:os161
```

This will create a local repository in a directory called os161 (which can be anywhere in your account on antipasto or on your own Mac or Linux computer).

You should now be able to configure the kernel using the `LINKED_LIST` config file – see “Configure a kernel” in the “Building OS/161” page linked from the course website.

Once configured, you should be able to build, install, and run the kernel. When you run it, you can enter commands to run. To run the linked list test that I’ve already provided, type `llt`.

**READ EXISTING LINKED-LIST CODE**

The existing code for a doubly-linked list is in:

- `src/kern/include/linkedlist.h` – The declarations defining types and functions that provide linked list behavior.
- `src/kern/concurrentlist/linkeddlist.c` – The implementations for a pretty minimal, sorted doubly-linked list.
- `src/kern/concurrentlist/linkeddlisttest.c` – A starting point for your tests. It forks two threads, each of which adds some items to the head of a shared list.

Read this code and make sure you understand it.

0.0.1 QUESTIONS?

1. Can you figure out how this code is run when the user types `llt`?
2. What does `splhigh()` do? Why do you have it at the beginning of each of the two threads?
3. What would `splx(splhigh())` do?
IMPLEMENT A SORTED DOUBLY-LINKED LIST

The linked-list provided only has a function to insert at the head of the list, setting the key appropriately. Add implementations for the functions specified in linkedlist.h that aren’t implemented yes.

Add code to linkedlisttest.c that uses your new methods to make sure they work.

In particular, you should make a test in which one of the threads is adding many items to the list and another thread removes them from the list.

TURNING IT IN

1. Print out and turn in your code from linkedlist.c and linkedlisttest.c. If you modified any other files (like linkedlist.h), print those out and turn them in as well.

2. Also, produce a trace of the output from running your tests. Use the script command to produce the trace:

   script testOutput.txt
   ...
   run your tests ...
   exit

BREAKING YOUR CODE WITH CONCURRENCY (75%)

Due electronically Wednesday of Week 3, at 11:00pm (or later with a penalty)

In this part, you will control the concurrency of the program to produce failures. In class, we have talked about thread synchronization primitives and how to use them to avoid problems with synchronization. In this assignment, you will get first-hand experience with the problems that are caused by unsynchronized code.

1. Try to think of interleavings that would cause problems with your code. Try to think of many different interesting categories of bugs that could be caused by bad interleavings. Don’t spend lots of time on bugs that are basically the same as each other – try to think of different ones. You must have at least five (5) substantially different bad interleavings.

2. Instrument your code so that you can cause these bugs/interleavings to happen. In a “real” multithreaded system, you don’t have control over the interleavings, but with OS/161 you do. You can call thread_yield() to yield the CPU to another ready thread, if there is one. I recommend deciding in which positions you will need to yield in order to implement all of your different interleavings. You can then write helper functions that yield at these locations.

   For example, let’s say that in case 0 you want to yield on line 45, while in case 1 you do not. Let’s call line 45 “location 5”. On line 45, you might then call yield_if_should(5); yield_if should could look up in a previously-defined boolean table based on the location number and the test number:

   if(yieldArray[testNum][location]) { thread_yield(); }

   If you have this kind of helper function, you can make new test cases by simply initializing the yieldArray array differently in linkedlisttest.c. You might also find it useful to have yield_if_should count the number of times we have reached each location, and then add a dimension to yieldArray for the count.

3. Keep a lab notebook in which you give, for each different category:
(a) A test number. I should be able to run llt <testnum> to execute this test case, where <testnum>
is the test number you provide here.

(b) Describe an interleaving that can cause the bug.

(c) Describe what behavior you expect for this interleaving.

(d) Describe what behavior you actually see when you execute this. If this is different from what
actually happened, this is fine – as long as you can explain what actually happened. You might
have to use a debugger (such as gdb) to step through the program to figure out what really
happened.

As an example, here’s my lab notebook entry for one category of bug (with line numbers removed):

**Test Number 2**

Interleaving In thread 0, we are adding an element to an empty list, while thread 1 is also adding an
element with the same key to the list.

(a) Thread 0: in linkedlist_insert() [linkedlist.c: line ??], just found that list is empty
(b) Thread 1: calls and completes linkedlist_insert(). Yields before removing element
[linkedlist.c: line ??].
(c) Thread 0: [linkedlist.c: line ??] now updates first and last to point to the new element.
This is incorrect because first should point to the element added by Thread 1.

Expected Behavior I expect that in the middle of the execution, the list will have only one element
when it should have two. Thread 1 will correctly add an element, but Thread 0 will try to add
an element to a list it thinks is empty. The element added by Thread 1 will therefore not be seen
after Thread 0 is done.

Therefore, when the threads get to removing the elements, there will be a segmentation fault
when we try to remove the second element.

Actual Behavior After running this several times and stepping through it with gdb, I realized that
I was not actually executing the interleaving I was trying to execute. I had to change the calls
to thread_yield() to take into account the number of times linkedlist_insert() and
linkedlist_remove_head() had been called so far.

Now, the run produces the following output. The second printing of the list should show two
elements, but only shows one.

```
Running test 2
*** thread 0 adding 1 elements
adding key 0 to list
yield after finding list empty
*** thread 1 adding 1 elements
adding key 0 to list
1: 0[A]
*** thread 1 removing 1 elements
yielding before actually removing anything.
0: 0[A]
*** thread 0 removing 1 elements
Item 0: A
0:
Segmentation fault
```

**What to Turn In and How**

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

- Source code for all code you have written for the assignment.
• Output generated by your test program that demonstrates that the doubly-linked list worked before you introduced concurrency.
• Your lab notebook describing the ways that you have caused your program to fail with concurrency.

You should also electronically submit your copy of the OS/161 code. To do this, do the following on antipasto (if you were working on your own computer, copy your files to antipasto before you do this):

```
/home/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.

**EVALUATION**

For the first part of the assignment, I am looking to make sure that your doubly-linked list code handles all the boundary conditions (empty list, adding to the end, adding to the beginning, etc...).

I am also looking for good programming style.

For the second part of the assignement, I am hoping you will come up with a good variety of interesting categories of buggy interleavings. I am also grading on how well you present your results in your lab notebook.

**REFERENCES**