Lab #3: Synchronization

Lab 3 is due Monday, Nov. 11. Your lab report and source code must be submitted by **1:25PM** before class. The late policy applies to this lab project.

When you measure the performance of the programs you write for this lab, make sure the measurement is stable and consistent. A simple way to do that is to repeat the execution multiple times and use the average.

Problem 1: Synchronization

Assume we have a *two-way linked list* with the following data structure declaration. Every node has an integer key "v". We want to use this data structure to implement a FIFO queue, i.e., new elements will always be added to the tail and existing elements will be only removed from the head.

struct p {
 	
int v;
 	
struct p * pre;
 	
struct p * next;
 	
}
	struct p * head = nullptr;
	
	struct p * tail = nullptr;
	
	void add (int v);
	
	int remove();
	
	int size();

a. Implement three methods based on the linked list. The three methods are (1) "add", that adds a key to tail of the list, and update "tail" or potentially "head" accordingly. Remember the list might be empty. (2) "remove", will remove and return the key pointed by "head" from the list, and update "head" and "tail" properly. If the list is empty, wait until a new element become available. And (3) "size" return the size of the list.

Also when a new node is needed, just malloc the memory for that node.

b. Implement the following workload. Spawn 8 threads, each of which executes the workload. The goal of this task is to minimize the overall execution time of the 8 threads.

```
For(i=0; i<1K; i++){
   Add(i*thread_id);
}
For(i=0; i<100K; i++){
   Add(i);
   remove();
}
Print size().
```

(1) First make sure your implementation of the methods in a. is thread-safe, i.e., multiple threads can call them concurrently. The simplest way is to use one lock to protect the whole list. Assume this is the baseline version for this task. Measure the execution time of the 8 threads based on this baseline version.

(2) Design and implement a better synchronization schemes at method level (i.e, not protecting the whole loop)., in pthread, for the "add" and "remove" methods, specifically for the example workload. Very importantly, the scheme should provide the best tradeoff between maximum concurrency and lock overhead. The maximum concurrency means that as long as the invocations of the methods from multiple threads change different locations of the list, the multiple invocation should run concurrently, i.e., not waiting for each other. For example, if one thread is adding a key to a list of 100 elements, and another thread is removing a key, then the two threads should not block each other. The lock overhead is incurred when a lot of locks are used in a synchronization scheme.

Measure the execution time of the 8 threads based on the new synchronization scheme. Very importantly, you should verify that your implementation is correct at the end of the program: the list is properly formed, and list size = 8K. ***Explain why you think your scheme works better***.

What to submit:

The package you submit for this lab should include your source code, performance results and analysis, and brief description that you think might help the instructor understand your code, e.g., about any design choices you make in your program. The instructor will compile, run and measure the performance of your code. Therefore, you should also describe how to compile and run your code in your submitted documentation.

How to Submit:

Copy your lab report, which is a .pdf, a .doc, or a .html file, and all your source code into an empty directory. Assuming the directory is "submission", make a tar ball of the directory using the following command:

tar czvf [your_first_name]_[your_last_name]_lab3.tar.gz submission.

Replace [your_first_name] and [your_last_name] with your first name and your last name.

Submit the tar ball. The submission time will be used as the time-stamp of your submission.
