Operating Systems

Homework Assignment #3

Due 8.1.2017, 23:50

In this assignment, we learn some aspects of concurrent programming, how to write code that can be safely executed by multiple threads. The objective is to apply these methods to implement a thread safe doubly-linked list data structure.

In addition, you are required to develop code that simulates multiple writers and readers pushing and popping items concurrently to/from the list, with automatic garbage collector. The garbage collector thread is activated when the list gets too large.

Submission:

- Carefully follow all submission guidelines.
- Make sure filenames are correct, case-sensitive, zip structure is correct.
- Submit an actual zip file created using the *zip* command **only!**
- **Submit 1 file:** `hw3.c` (in a zip file, of course, per the submission guidelines!)

Linked List

First, implement a concurrent doubly-linked list.

The list is a list of integers, and the implementation contains 7 methods: *init, destroy, push_head, pop_tail, remove_last_k, length, get_mutex*.
You may define the list as you wish (i.e., a struct, typedef, etc.). Precede each method name with "*intlist_*", for ease of reading.

- *init* - initialize the list. You may assume the argument is not a previously initialized or destroyed list.
- *destroy* – frees all memory used by the list, including any of its items.
- *push_head* – receives an int, and adds it to the head of the list.
- *pop_tail* – removes an item from the tail, and returns its value.
- *remove_last_k* – removes *k* items from the tail, without returning any value.
- *size* – returns the number of items currently in the list.
- *get_mutex* – returns the mutex used by this list.

You may assume *init()* and *destroy()* are called once for each list and not concurrently with any other calls for that list, i.e., these methods do not need to be thread-safe.

For the other calls – hold the lock for as little time as possible, i.e., do the most you can **outside** of the critical section. Use condition variables where appropriate.

Note that several instances of your list implementation may be created and should work *correctly*.

Further clarifications are provided in Appendix A.

Simulator

Second, implement a simulator which tests the list data structure you'd create.

Implement the simulator in the same C file as the linked list; use only the 7 functions specified in the previous section.

Command-line arguments:

1. WNUM – number of writers (assume a positive integer)
2. RNUM – number of readers (assume a positive integer)
3. MAX – max number of items in the list (assume a positive integer)
4. TIME – number of seconds to run (assume a positive integer)

The flow:

1. Define and initialize a global doubly-linked list of integers.
2. Create a condition variable for the garbage collector.
   (different than the condition variable used internally by the list's *pop_tail* operation)
3. Create a thread for the garbage collector.
4. Create WNUM threads for the writers.
5. Create RNUM threads for the readers.
6. Sleep for TIME seconds.
7. Stop all running threads (safely, avoid deadlocks!)
8. Print the size of the list as well as all items within it.
9. Cleanup. Exit gracefully

Threads

- **Writers** - writer threads push random integers to the list, in an infinite loop.
- **Readers** – reader threads pop integers from the list, in an infinite loop.
- **Garbage Collector** – the garbage collector waits until the list has more than MAX items. Once it has, the garbage collector removes half of the elements in the list (from the tail, rounded up). In addition, the garbage collector prints the number of items removed from the list. Output a message like the following: "*GC – 7 items removed from the list*".

The reader and writer threads **can and should** help the garbage collector to wake up correctly.

Appendix A

Removing Items

The operation *pop_tail()* is blocking, i.e., if the list is empty – wait until an item is available, and then pop it.

When *remove_last_k()* is called with a *k* larger than the list size, it removes whatever items are in the list and finishes.

Function Signatures

Following are the recommended signatures for each function:

```
void intlist_init(intlist* list)

void intlist_destroy(intlist* list)

pthread_mutex_t* intlist_get_mutex(intlist* list)

void intlist_push_head(intlist* list, int value)

int intlist_pop_tail(intlist* list)

void intlist_remove_last_k(intlist* list, int k)

void intlist_size(intlist* list)
```

Performance

You are measured and graded for performance of both your general code and its multicore execution.

Make sure locks are held only when necessary, there are a minimal number of lock and unlock operations, condition variables are used to avoid busy-wait wherever appropriate, etc. For example, not all methods require a lock, so you should avoid locking whenever possible.