Project: Multi-Threading

COSC 216.001

Due Date: Dec 12, 2021, 11:59pm

1 Overview

Use Posix threads to create concurrently executable code.

1.1 Objectives

- Learn how to write a multi-threaded program
- Learn how to build multi-thread safe data structures

2 Detailed Description

2.1 Introduction

Modern computer programming makes extensive use of concurrent execution through the use of threads. Multi-threaded programming, requires careful management of data structures to make them *multi-thread safe*. In this project, you will investigate thread-safe coding by building three simple utility libraries. The first is a simple counter that multiple threads can use. The second is a hash table, an important data structure used in many multi-threaded applications as well as in the operating system itself. Finally, the third is a data structure that maintains a set of the most recently used (MRU) objects. This structure is something like that would be needed to implement an OS cache, that keeps the MRU data, but purges the least recently used (LRU) data as necessary.

2.2 What to do

You are provided with header files for each of the three parts of this project. Further information about each part and the related functions follows below.

2.2.1 Multi-Threaded Counter

For the first C library you should implement thread-safe increment, decrement, and get value procedures. Below are the interfaces that you should implement. The header file for this library has been provided in `MTCounter.h`

`MTCounter`: The header includes a stub `typedef` for this data structure. You should provide a real definition within your implementation. If you are careful **not** to include the header file in your implementation, you can avoid a double declaration error.

`MTC_init`: Allocates and initializes your counter data structure. It should include storage for the value as well as a pthread mutex to make your counter thread-safe. Return the resulting structure.

`MTC_value`: Returns the current value of the counter. Do you need to lock the mutex during this call? Why or why not?

`MTC_increment`: Increments the counter. To make this thread-safe, you should lock and unlock the mutex before and after the increment operation.

`MTC_decrement`: Similar to `MTC_increment`, but here we decrement instead.

2.2.2 Multi-Threaded Hash Table

The second library you should implement is a thread-safe hash table. In your previous CS courses (e.g. CMSC204), you implemented a single-threaded hash table. Operating systems make use of hash tables quite a bit, and often they are accessed by more than one thread. Hence, it is good practice to see how one would implement a thread-safe hash table.

Your implementation will only store integers. Below are the interfaces that you should implement. For a benchmark, a good implementation should only be a couple hundred lines of code. The header file this library has been provided in `MTHash.h`:

`MTHash`: The header includes a stub `typedef` for this data structure. You should provide a real definition within your implementation. If you are careful **not** to include the header file in your implementation, you can avoid a double declaration error.

`MTH_init`: Allocates and initializes your hash table data structure. It should include storage for the indicated number of hash table entries. Each entry will need to point to store a pointer to linked-list of. values, as well as a pthread mutex to make your table entry thread-safe. Return the resulting table structure.

`MTH_add`: Inserts an integer x to the hash table. Return -1 if x already exists in the hash table (and do not re-insert x to the hash table). Return 0 if x has been successfully added to the hash table. The table entry that should be selected for a given x is: $index = x \bmod tableSize$. Each bucket should maintain a linked list of integers. It is up to you how you want to manage the list (e.g. it can be sorted, it can also be unsorted).

`MTH_delete`: Removes x from the hash table. Return -1 if x does not exist. Return 0 if x has been successfully removed.

`MTH_size`: Counts and returns the number of elements in the hash table.

`MTH_bucket_size`: Counts and returns the number of elements in the specified table entry.

`MTH_print`: Prints the contents of your hash table. Each entry should print as a space-delimited list of integers on a single line. Print a new line after each bucket. Empty table entries should result in blank lines.

2.2.3 Multi-Threaded Cache

Operating Systems make extensive use of data caches to improve the memory latency of the system. In today's multi-core systems, the availability of thread-safe caches is paramount to the performance of the system. The final library you should build implements a data structure to track a set of positive integers (values greater than zero) that retains the Most Recently Used (MRU) entries. When the cache is full, space should be made for new entires by removing the Least Recently Used (LRU) data. Below are the interfaces that you should implement. The header file this library has been provided in `MTHash.h`:

`MTCache`: The header includes a stub `typedef` for this data structure. You should provide a real definition within your implementation. If you are careful **not** to include the header file in your implementation, you can avoid a double declaration error.

`MTL_init`: Allocates and initializes your cache data structure. It should include storage for the indicated number of cached integer values. How you implement thread safety in your data structure is up to you. I suggest you experiment with a couple of options to identify the best performing solution.

`MTL_add`: Inserts an integer into the cache. If the cache is full (e.g., it has *size* elements in it already), you should first remove the *Least Recently Used* (**LRU**) element; this is the value that should be returned. The new element should be added to your cache as the *Most Recently Used* (**MRU**) value. If no **LRU** value is removed from the cache (e.g. the cache is not full or a duplicate value is added), the function should return 0. If the user tries to insert the value 0 or negative values, the function should return -1 as an indication of failure.

`MTL_update`: Makes the provided integer argument the **MRU** element in the cache. If the integer does not appear in the cache, the function returns -1, otherwise it returns 0.

`MTL_delete`: Removes the integer argument from the cache. Returns 0 upon success and -1 otherwise.

`MTL_size`: Returns the number of data elements in the cache.

`MTH_print`: Prints the contents of the cache as a space-delimited list terminated by a newline. Cache entries should be listed in order from *Most Recently Used* to *Least Recently Used* (**MRU->LRU**).

Further Reading

More Reading

We covered most of the necessary information during class lectures. General discussion of multi-threading and concurrent programming can be found in chapter 12 of *Computer Systems*, particularly sections 12.3-12.5. Further references for the specific functions you may need.ca be found in the following manual pages:

- `pthread_create`
- `pthread_self`
- `pthread_join`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

3 Submission

Each of your source code files must have a comment near the top that contains your name, StudentID, and M-number.

The project should be submitted via Blackboard by **December 12, 2021, 11:59pm**. Follow these instructions to turn in your project.

You should submit the following files:

- MTCounter.c, MTHash.c, MTCache.c
- *any other source files your project needs*

The following submission directions use the instructions that we will use for all projects this semester.

1. create a directory for your project:

   ```
   mkdir proj7
   ```

2. create (or copy) all of your source files in this directory. Example: To copy a file called `example.c` into your `uShell` directory:

   ```
   cp example.s proj7
   ```

3. Create a tar file named `<user_name>.tar.gz`, where `<user_name>` is your studentID and `<proj_dir>` is the directory containing the code for your project, by typing:

   ```
   tar -czf <user_name>.tar proj7
   ```

4. Finally, submit the compressed tar file to Blackboard in accordance with the class policies.

Late assignments will not be given credit.

4 Grading

Your score will be computed out of a maximum of 100 points based on the following distribution:

80 Correctness.

20 Style points. We expect you to have good comments (10 pts) and to check the return value of EVERY system call (10 pts).

For the purposes of the **Good Faith Attempt** (GFA). Only your implementation of MTCounter.c will be evaluated. Correct implementation of all functions in MTCounter.h will be sufficient to qualify for the GFA.