Thread-safe Library

CS453: Operating systems

Overview

In this project, we will create a thread-safe version of an existing list library. We will then test it using a provided solution to the producers-consumers problem.

Setup

Run the **git pull --rebase** command in your backpack folder (on the master branch). You should see a `p4` project folder in your backpack with all the starter code. It should look like the listing below:

```
[amit@localhost p4(master) ]$ ls -A

backpack.sh  include  Item.h  Makefile  test-pc-mq.sh  wrapper-library
.gitignore   Item.c   lib     pc.c      test-pc.sh

[amit@localhost threadsafe-library(master) ]$ ls -A lib include wrapper-library
include:
.gitkeep

lib:
.gitkeep

wrapper-library:
Makefile  ThreadsafeBoundedList.h
```

You will create the `ThreadsafeBoundedList.c` file in the wrapper-library subfolder to complete the implementation of the thread-safe list library.

Specification

Wrap and existing library

First, we will create a **monitor** version of a doubly linked list library implementation (that was provided for you earlier in the `list` folder)**.** We will also add an additional feature to bound the capacity of the list. Since we don't have the source code for the list library, we will build a wrapper library that creates the monitor version around the original library. Sometimes in industry you may not have access to the original code so your only option is to patch the binary or write a wrapper for the existing code to fix any issues. Read about how Microsoft chose the patching method for their equation editor in Microsoft Word.

https://www.bleepingcomputer.com/news/microsoft/microsoft-appears-to-have-lost-the-source-code-of-an-office-component/.

The API for this library has been provided to you in your backpack repo in the `ThreadsafeBoundedList.h` file in `p4/wrapper-library` folder.

We declare a structure that contains a pointer to the underlying list as well as additional variables to implement the monitor. However, we declare it incomplete in `ThreadsafeBoundedList.h` header file as shown below.

```
struct tsb_list;
```

We will provide a complete declaration in `ThreadsafeBoundedList.c` file as shown below. This makes the internals of the structure opaque to the users and they cannot directly modify those variables.

```
struct tsb_list {
      struct list *list;
      int capacity;
      Boolean stop_requested;
      pthread_mutex_t mutex;
      pthread_cond_t listNotFull;
      pthread_cond_t listNotEmpty;
};
```

We also provide wrapped versions of all the underlying functions from the list library as well as some additional functions. See the header file for details on the functions that you are wrapping. You will be creating and implementing these functions in the `ThreadsafeBoundedList.c` file. Each function should be protected by a mutex. If the list becomes full, then adding to the list shall block on a condition variable. If the list is empty, then removing from the list shall block on another condition variable.

Relevant man pages are:

- **pthread_mutex_init**
- **pthread_mutex_lock**
- **pthread_mutex_unlock**
- **pthread_cond_init**
- **pthread_cond_wait**
- **pthread_cond_signal**
- **pthread_cond_broadcast**

Using the wrapper Library

We will simulate a size-bounded queue (implemented as a doubly-linked list with a limit on maximum size) being shared by multiple producers and multiple consumers running simultaneously. Suppose we have **m** producers and **n** consumers (where **m** and **n** are supplied as command line arguments). The initial main process creates a global queue data structure so all threads can access it. Then the main process creates **m** producer threads and **n** consumer threads.

We will fix the number of items each producer produces (as a command line argument). Then we will keep track of how many items each consumer actually consumed. At the end of the program, we print out the total number of items produced versus the total number of items consumed. These two numbers should be the same unless the program has race conditions.

The items are a structure that contain an unique id (basically number the items 1, 2, 3, ... as they are entered in the queue) and also contains the index of the producer that created it.

We have provided you with a working version of the producer-consumer test program in the file **pc.c**. You should not modify the test program **pc.c**. However, you will need to add an additional function `finishUp()` to your wrapper list class that allows the simulation to be stopped after the producers are done by signaling the consumers to clean up any remaining items in the queue.

The producer/consumer threads test program uses the monitor version of the doubly linked list to run the simulation. To test your monitor version, the producers randomly insert new items at the front or back of the queue. Similarly the consumers randomly remove items from the front or back of the queue.

Setup for libraries

We are using two libraries in this project. The original linked list library `libmylist.so` that is in the folder `../list/lib` (relative to p4) and the wrapper library `libthreadsafe-mylib.so` that is in the `./lib` directory (relative to p4). You must export both paths in `LD_LIBRARY_PATH` for everything to work and the paths must be relative to the executable (pc). For example, use the following command before running the pc executable:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:../list/lib:./lib
```

Notes on testing

-
- When testing your program limit the number of threads to less that ten and number of items produced per producer to less than ten thousand to avoid overloading *onyx*. On your system, you can test with higher values.
- Run the program several times for the same input arguments. Verify that the results do not vary.
- Comment out the output statements. See if that changes the results for the number of items consumed or produced.
- Use the testing script `backpack.sh` provided in the assignment folder. It uses the `test-pc.sh` script, which has also been provided.

Extra Credit: Multiple Queues (10 points)

Create another version of the testing program pc.c and call it pc-mq.c and adjust the `Makefile` accordingly.

Add another command line argument that allows the user to specify the number of queues (with the same size limit on each). The number of queues should be the first command line argument.

```
Usage: pc-mq <#queues> <poolsize> <#items/producer> <#producers>
             <#consumers> <sleep interval(microsecs)
```

Now modify your the testing code to create multiple queues and have the producers/consumers access the queues in round-robin fashion, which is explained below. Suppose we have **k** queues. Then if the ith queue is full, the producer moves on to the **(i+1) mod k**th queue. Similarly if the **i**th queue is empty, the consumer moves on to the **(i + 1) mod k**th queue.

Testing would be the same as before. An additional test script is `test-pc-mq.sh` You could also informally test if multiple queues allows the producers and consumers to get their work done faster although that would depend on the mix of the work being simulated.

Submission

Files committed to git (backpack)

Required files to submit through git (backpack), if you created more helper files that is fine.

1. Makefile
2. Any and all files to ensure the project builds
3. README.md (at the top-level of the p4 folder)

Push your code to a branch for grading

Run the following commands

1. make clean
2. cd p4
3. git add <file ...> (on each file!)
4. git commit -am “Finished project p4”
5. git branch thread_safe_library
6. git checkout thread_safe_library
7. git push origin thread_safe_library
8. git checkout master

Check to make sure you have pushed correctly

- Use the command **git branch -r** and you should see your branch listed (see the example below)

```
$ git branch -r
```

```
origin/HEAD -> origin/master
origin/master
origin/thread_safe_library
. . .
```

Submit to Blackboard

You must submit the sha1 hash of your final commit on the correct branch. Your instructor needs this in order to troubleshoot any problems with submission that you may have.

- git rev-parse HEAD

Grading Rubric

Provided via backpack.