CS 551 Systems Programming, Summer 2024
Programming Project 3

Out: 07/29/2024 Mon.
Due: 08/04/2024 Sun. 23:59:59

Download the base code for this project from the Brightspace website.

The task is to implement a barrier synchronization facility *mybarrier* using Pthread condition variable (and, of course, Pthread mutex). The usage of *mybarrier* is similar to (but not exactly the same as) the Pthread barrier. Your job is to implement the three mybarrier APIs, `mybarrier_init()`, `mybarrier_wait()`, and `mybarrier_destroy()`, according to the following specification.

- The *mybarrier* facility is represented by a data type defined (in `mybarrier.h`) as:

```
typedef struct _mybarrier_t{
    int count;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} mybarrier_t;
```

  The first filed `count` is to record the barrier count set by the user. The second field `mutex` is a Pthread mutex to be used with the third field, `cond`, which is a condition variable. You may add field(s) into the structure in your implementation.

- A multi-threaded application wishing to use *mybarrier* calls `mybarrier_init()` to obtain an initialized *mybarrier* structure.

```
mybarrier_t * mybarrier_init(unsigned int count);
```

  The argument `count` specifies the barrier count, i.e., the number of threads that must reach the barrier before all of the threads will be allowed to continue. This `init` function allocates memory space for a `mybarrier_t` structure, initializes it properly (i.e., initializes the barrier count, and the mutex/condition variable associated with the barrier), and returns the its address as the function's return value.

- The application installs the barrier within its threads by calling `mybarrier_wait()`.

```
int mybarrier_wait(mybarrier_t * barrier);
```

  In other words, `mybarrier_wait()` is used to indicate that a thread is done with its work and is ready to wait for all the other threads to catch up.

  More specifically, the thread calling `mybarrier_wait()` is put to sleep if the barrier count (set in the call to `mybarrier_init()`) is not yet satisfied. If the thread is the last one to call `mybarrier_wait()`, thereby satisfying the barrier count, all of the threads are awakened. More requirements about this API:

  - **After all the threads are awakened, the barrier structure cannot be used again** (this is different from the Pthread barrier). In other words, only the first `count-1` calls to `mybarrier_wait()` block; the `count`-th (counting started from 1) call to `mybarrier_wait()` allows all the `count` threads to return 0 and continue; and the (`count+1`)-th and thereafter calls to `mybarrier_wait()` will return -1.
  - If a NULL pointer is fed as the the input argument, the function returns -1.

- After the program is done using the barrier, it calls `mybarrier_destroy()` to free up the resources associated the `mybarrier` structure (i.e. destroys the mutex/condition variable associated with the barrier, frees the memory space allocated for the `mybarrier` structure).

  ```
  int mybarrier_destroy(mybarrier_t * barrier);
  ```

  It is possible that the main thread calls `mybarrier_destroy()` when there are still threads blocking at `mybarrier_wait()`. Your implementation should be able to deal with this (that is, a correct implementation should ensure the resource destroy/deallocation happen after all the threads are unblocked from the wait function).

Add your implementation to the base code. The Makefile given compiles your implementation and generates an object file, which can be linked to the test program. For testing, write your own test program based on the above spec. But do not submit your test program. We will use our own test program for grading.

Log and submit your work

Log your work: besides the files needed to build your project, you must also include a README file which minimally contains your name and B-number. Additionally, it can contain the following:

- The status of your program (especially, if not fully complete).
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

Compress the files: compress your README file, all the files in the base code folder, and any additional files you add into a ZIP file. Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be "proj3_abc.zip".

Submission: submit the ZIP file to Brightspace before the deadline.

Grading guidelines

(1) Prepare the ZIP file on a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(2) If the submitted ZIP file/source code files included in the ZIP file are not named as specified above (so that it causes problems for TA's automated grading scripts), 10 points off.

(3) If the submitted code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(4) If the code is not working as required in this spec, the TA should take points based on the assigned full points of the task and the actual problem.

(5) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private.