Fundamentos de Sistemas de Operação

MIEI 2018/2019

Homework Assignment 2

Deadline and Delivery

This assignment is to be performed by groups of **2 students** – any detected frauds will cause failing the discipline. The code has to be submitted for evaluation via the Mooshak system (`http://mooshak.di.fct.unl.pt/~mooshak/`) using group's account -- the deadline is ***23h59, December 4th, 2018 (Tuesday)***.

Description

The goal of this assignment is to implement a simplified version of the phaser synchronization mechanism. A *phaser* is a collective synchronization operation that allows multiple threads to synchronize on a set of points (phases). Upon arrival on one of such phases a thread may:

- signal that it has reached the phase and continue its execution (operation **advance**)
- signal that it has reached the phase, and wait for all other threads to reach that same phase before resuming its execution (operation **advance_and_await**)

If you want to know more about Phasers, you may take a look at Java's Phaser class: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Phaser.html

Work to Do

The `fso_phaser.h` header file defines the `fso_phaser_t` type and the operations that may be applied upon a phaser. In more detail, `fso_phaser_t` is an alias for **`struct`** `fso_phaser` defined as follows:

```
struct fso_phaser {
  pthread_mutex_t mutex;  // The mutex for exclusive access to the phaser
  pthread_cond_t cond;    // The condition variable for signaling phase reaching
                          // and waiting for other threads.
  int number_threads;     // Number of threads using the phaser for synchronization.
  int* phases;            // Array with the phase of each thread in the phaser.
};

typedef struct fso_phaser fso_phaser_t;
```

Any phaser must first be initialized for a defined number of threads with `fso_phaser_init`. From the functions that may be applied upon a phaser, you must implement the following **three** (on file `fso_phaser.c`):

`int` `fso_phaser_current(fso_phaser_t* phaser);`
Returns the current phase of the `phaser`, i.e. the phase that **all** threads have already reached.
Consider a phaser for the synchronization of 5 threads, if the `phases` array field comprises the following values:

3	5	3	3	7

then the current phase is 3, given that all values are >= 3.

Note: Assume that this function by itself does require mutual exclusive access.

`int` `fso_phaser_advance(fso_phaser_t* phaser);`
Increments the thread's phase in the `phaser`, i.e., increments the position assigned to the thread in the `phases` array. To know which is such position you may use function `pthread_to_pos`.
If successful, `fso_phaser_advance` returns zero, otherwise it returns -1 to indicate the error.

`int fso_phaser_advance_and_await(fso_phaser_t* phaser);`
Increments the thread's phase in the phaser and waits until the remainder threads have also reached the same phase, i.e. blocks the thread while the current phase of the phaser is lower than the thread's phase. If successful, `fso_phaser_advance_and_await` returns zero, otherwise it returns -1 to indicate the error

IMPORTANT: By default, POSIX thread mutexes are not recursive (or non-reentrant), hence a thread that tries to lock a mutex that already owns, will block forever (deadlock).

Testing

To test your solution, you may use file `test.c` that launches a set of threads to perform *some_computation*, while signaling on a phaser each time they complete a processing round (we can imagine the processing of several batchs of a long data stream). In turn, the main thread uses the same phaser to keep track of, and output, the overall computation's progress.