EC440: Project 3 - Thread Synchronization

Project Goals

- To enable numerous threads to safely interact
- To extend our threading library with basic synchronization primitives

Collaboration policy

- You are encouraged to discuss this project with your classmates/instructors but are required to turn in your own solution.
- You must be able to fully explain your solution during oral examination.

Deadline

It is due on Monday, November 4th 16:30 ET (no deadline extensions or late submissions).

Project Description

In Project 2 we implemented a basic threading library that enables a developer to create and perform computations in multiple threads. In this Project we are going to extend the library to facilitate communication between threads.

To achieve this, we will need to:

1. Implement a basic locking mechanism that switches the interactivity of a thread on/off. At certain stages of the execution of a thread, it may be desirable to prevent the thread from being interrupted (for example, when it manipulates global data structures such as the list of running threads). For this, you will need to create two functions: ***`void lock()`*** and ***`void unlock()`***. Whenever a thread calls ***`lock`***, it can no longer be interrupted by any other thread. Once it calls ***`unlock`***, the thread resumes its normal status and will be scheduled whenever an alarm signal is received. Note that a thread is supposed to call ***`unlock`*** only when it has previously performed a ***`lock`*** operation. Moreover, the result of calling ***`lock`*** twice without invoking an intermediate ***`unlock`*** is undefined.
2. Implement pthread_join POSIX thread function:

   ```
   int pthread_join(pthread_t thread, void **value_ptr);
   ```

   This function will postpone the execution of the thread that initiated the call until the target thread terminates, unless the target thread has already terminated. There is now the need to correctly handle the exit code of threads that terminate. On return from a successful ***`pthread_join`*** call with a non-NULL value_ptr argument, the value passed to ***`pthread_exit`*** by the terminating thread shall be made available in the location referenced by value_ptr

   The results of multiple simultaneous calls to ***`pthread_join`*** specifying the same target

thread are undefined.

3. Add semaphore support to your library. As discussed in class, semaphores can be used to coordinate interactions between multiple threads. You have to implement the following functions (for which you should include *semaphore.h*):

   a. ***int sem_init(sem_t *sem, int pshared, unsigned value);***

      This function will initialize an unnamed semaphore referred to by *sem*. The *pshared* argument always equals to 0, which means that the semaphore pointed to by *sem* is shared between threads of the process. Attempting to initialize an already initialized semaphore results in undefined behavior

   b. ***int sem_wait(sem_t *sem);***

      sem_wait decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement (i.e., the semaphore value rises above zero). For this Project, the value of the semaphore **never** falls below zero.

   c. ***int sem_post(sem_t *sem);***

      sem_post increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another thread blocked in a ***sem_wait*** call will be woken up and proceeds to lock the semaphore. Note that when a thread is woken up and takes the lock as part of *sem_post*, the value of the semaphore will remain zero.

   d. ***int sem_destroy(sem_t *sem);***

      sem_destroy destroys the semaphore specified at the address pointed to by *sem* which means that only a semaphore that has been initialized by ***sem_init*** should be destroyed using ***sem_destroy***. Destroying a semaphore that other threads are currently blocked on (in ***sem_wait)*** produces undefined behavior. Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using ***sem_init***.

Details/ Implementation Hints:

1. In Project 3 we extend the functionality of the threading library you built in the previous project. If you implement ***pthread_join***, ***lock*** and ***unlock*** before Monday October 28th 16:30ET, you will get 5% extra credit for the assignment.
2. Adding the ***lock*** and the ***unlock*** functions is straightforward. To lock, you can make use of the ***sigprocmask*** function to ensure that the current thread can no longer be interrupted by an alarm signal. To unlock, simply re-enable (unblock) the alarm signal, again using ***sigprocmask***. Use ***lock*** and ***unlock*** functions to protect all accesses to global data structures.
3. When implementing the ***pthread_join*** function, you will want to introduce a BLOCKED status for your threads. If a thread is blocked, it cannot be selected by the scheduler. A thread becomes blocked when it attempts to join a thread that is still running.
4. In addition to blocking threads that wait for (attempt to join) active threads, you might also need to modify ***pthread_exit***. More specifically, when a thread exits, its context cannot be simply cleaned up. You will need to need to retain its return value since other threads may want to get this return value by calling ***pthread_join*** later. Once a thread's exit value is collected via a call to ***pthread_join***, you can free all resources related to this thread (analogous to zombie processes).
5. One question that may arise is how you can obtain the return (exit) value of a thread that does not call ***pthread_exit*** explicitly. We know that, in this case, we have to use the return value of the thread's start function (run ***man pthread_exit***). A function normally passes its return value back to the caller (the calling function) via the *$rax* register. To access this value, some assembler code is needed. In the following, we assume that you have set up the stack of every new thread so that it returns to the function ***pthread_exit_wrapper***. **Note:** This is different from returning directly into ***pthread_exit***. You can then use the following code for this wrapper function:

```
void pthread_exit_wrapper()
{
    unsigned long int res;
    asm("movq %%rax, %0\n":"=r"(res));
    pthread_exit((void *) res);
}
```

6. You can see that this wrapper function uses inline assemby to load the return value of the thread's start function (from register *eax*) into the *res* variable. Then, it performs the desired call to ***pthread_exit***, using the return value of the thread's start function as the argument.

7. To implement semaphores include ***semaphore***.*h* that can be found in */usr/include/bits.* This header contains the definition of the type *sem_t*. You will notice that this struct/union is likely not sufficient to store all relevant information for your semaphores. Thus, you might want to create your own semaphore structure that stores the current value, a pointer to a queue for threats that are waiting, and a flag that indicates whether the semaphore is initialized. Then, you can use one of the fields of the *sem_t* struct (for example, *__align*) to hold a reference to your own semaphore.
8. Once you have your semaphore data structure, start implementing the semaphore functions as described above. Make sure that you test a scenario where a semaphore is initialized with a value of 1, and multiple threads use this semaphore to manage access to a critical code section that requires mutual exclusion (i.e., they invoke ***sem_wait*** before entering the critical section). When using your semaphore, only a single thread should be in the critical section at any time. The other threads need to wait until the first thread exits and calls ***sem_post***. At this point, the next thread can proceed.

Submission Guidelines

- The extended threading library must be implemented in C and compile with -Wall -Werror
- To facilitate grading, you must include the pthreads header file(#include<***pthread***.h>) in your source(s).
- Your makefile should compile your source files into an object threads.o file.

```
gcc -Wall -Werror -c -o threads.o threads.c
```

- In your home directory create a folder project3 and place all of your source files, makefile and README there. Switch to the project3 directory and execute ***submit3***
- A confirmation mail of your submission is sent to your account on ec440.bu.edu. You can read this mail by executing mail.
- In the README file explain what you did. If you had problems, tell us why and what you did to overcome them.
- You are allowed to resubmit your files. The latest submission before the deadline will be graded

Oral Examination

- Deadline: Friday, November 8th
- You are required to meet with one member of the course staff (excluding Prof. Egele) during office hours to explain your solution.