Lab 2 Reports

tiago_moore

Part 1 Learning Objective Summary

The output varies depending on the implementation of the code..

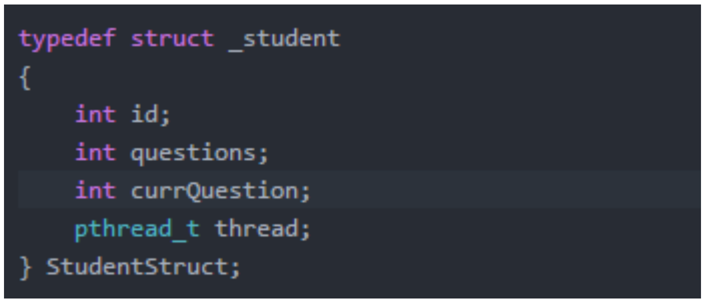
If I implement the pthread_join call.. and wait for each thread to exit in order.. the the data is access in order and simply increments by 1 each time the global variable is accessed. However, when I don't wait for each thread to wait in order, the global variable is accessed by multiple threads at the same time causing synchronization issues(A race condition). The data result we see from one thread is completely independent of the result we see from the following thread. thread 1 might show 15 and thread 2 might show 12, even though it should have added consecutively.

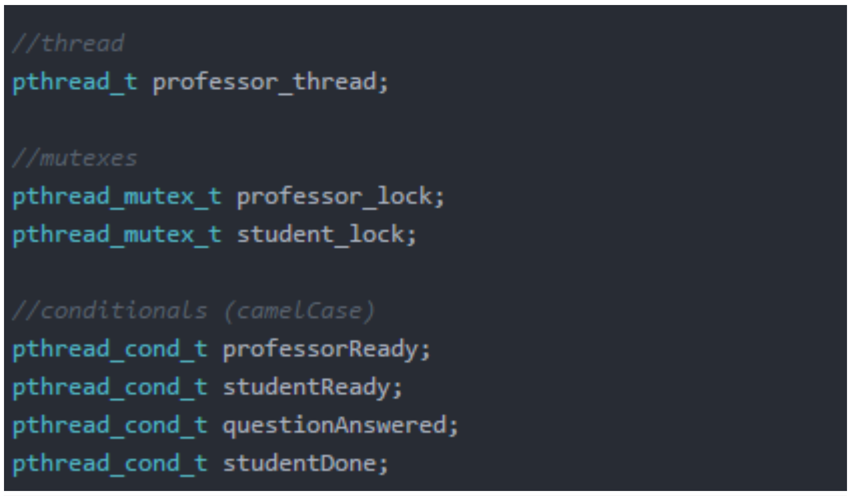
In summary, the scheduler could start threads before another thread completes, therefore without any synchronization measures including a pthread_join, the global variable could have been modified by thread 2 before thread 1 completes.... causing thread 1 to produce an invalid variable result- or the other way around contingent on how the scheduler.
With a mutex lock, a secondary thread cannot have access to the variable that the first thread is modifying, therefore no other thread will be able to change the value of a variable ensuring synchronized access of shared resources in the code.A mutex lock is one way of controlling a commonly shared variables between multiple concurrently running threads. Implementing the barrier before the pint statement allowed all threads to finish before continuing and printing the result.

lukas_borges

Part 2 Learning Objective Summary

For part two I used mutexes and locks to synchronize the interaction between the students and the professor:


```
typedef struct _student
{
    int id;
    int questions;
    int currQuestion;
    pthread_t thread;
} StudentStruct;
```


```
//thread
pthread_t professor_thread;

//mutexes
pthread_mutex_t professor_lock;
pthread_mutex_t student_lock;

//conditionals (camelCase)
pthread_cond_t professorReady;
pthread_cond_t studentReady;
pthread_cond_t questionAnswered;
pthread_cond_t studentDone;
```

How it works:

Each student created fires a thread (thread).
The professor is its own thread (professor_thread).
One mutex lock for the professor (professor_lock)
One mutex lock for the student (student_lock)

Conditionals work the following way:
Professor ready to answer question (professorReady)
Student ready to ask question (studentReady)
Question answered (questionAnswered), and,
Student satisfied with the answer for a question (studentDone).

So the way I coded is the following:

Professor code:
1. Professor fires first. Locks professor_lock.

2. It creates a thread and passes its action function which consists of:
- Signaling that the professor is ready
- Waiting for student to be ready
- Answering question
- Signaling that question was answered
- Waiting for student to be satisfied

3. Professor waits until it is ready once again (to avoid students from getting professor_lock first).

4. Professor unlocks the professor_lock.

Student code:
1. Student fires second and locks student_lock.
2. Student runs enterOffice function
3. Student unlocks student_lock.
4. Student passes a an action function to its thread which consists of:
- Locking the student_lock (so that no other student interacts with the Professor).
- Lock the professor_lock (so that the professor will interact with this student).
- Ask question (questionStart).
- Signal the professor to wake up and answer (studentReady).
- Wait for the professor to answer.
- Check to see if all questions were answered; if yes leaveOffice
- Signal that student is satisfied with answer
- Wait for the professor to be ready for the next student
- Unlock both mutexes (professor and student)

What have I learned?
After the T.A. analyzed my code he explained me that I had a busy-waiting mechanism.
My professor vs. student interaction is fine, but the way I have the students waiting to get in office to ask questions to the professor was wrong.

What I did was the following:

- Created a student queue to be populated by the amount of students the user passes in.
- While there were at least one student in queue, pass the next student to the office.
- If office capacity full, no students get in the office. While loop keeps running until there is space again and another student can be dequeued.

What I was supposed to do:

Instead of queuing up the students I was supposed to use a semaphore (mutex) to just pass the student in. If capacity full, sleep.
Once a spot is open, wake everyone up and let the computer choose which student comes in to

ask question.

By not doing so, the program is more likely to have students ask batch questions. I should have synchronized in a more efficient way, but at least my partner and I learned how to synchronize without having to put the system in a busy-waiting state.

This was a tough assignment but once again I learned a lot.
Thanks again Professor Wei.