GTU Department of Computer Engineering
CSE 312 - Spring 2022
Homework #1 Report

Hasan Mutlu
1801042673

IMPLEMENTATION

In the homework, I couldn't make a library-like implementation. It was because in order to implement a library, we must be able to make use of dynamic memory allocation to create and add new Tasks to TaskManager, without losing the locally created data in constructors etc. But it isn't provided for the template OS at the level we start our implementation from. (Can't even use the "new" keyword.) Because of that, I couldn't provide yielding, terminating and joining operations either. I tried terminating threads by deleting them from Task array in TaskManager but it caused errors.

Instead, I implemented a test scenario of same couple of threads, with and without Peterson's Algorithm. Firstly, two threads will work without Peterson's algorithm and produce wrong result. Secondly, two threads with same functionalities will work with Peterson's Algorith and produce correct result.

For this test scenario, I made a research about how can I simulate a race condition. I found an example video and implemented that scenario. Implementation took place in kernel.cpp file and no other files are modified

Definition of Test Scenario:

In this scenario, there are two threads that work on same variable. The shared variable is an integer and it has an initial value. First thread increments and second thread decrements this value. But they don't directly increment the shared variable. Instead, they assign this shared variable into a local variable, do incrementation/decrementation and assign it back to the shared variable.

Race condition is caused by using this local variable. After first thread increments the local variable's value, it goes to sleep. While it is sleeping, second thread starts and initializes its local variable with the initial value of the shared variable. However this initial value is wrong because it should've been incremented in the first thread. Since the first thread didn't assign this incremented value back to the shared variable, wrong value of shared variable comes to the second thread. After second thread decrements its local variable, it goes to sleep as well. First thread assigns incremented value to the shared variable and finishes its job. But that data will be lost since second thread will also assign its decremented value and overwrite it.

Design Explanations:

-> There is one mainTask() Task created in kernel.cpp's main function. mainTask function first creates two threads that work without Peterson's Algorithm, wait for them to finish and print the final value of their shared variable. Then it repeats the same process with two threads that work safely with Peterson's Algorithm as well.

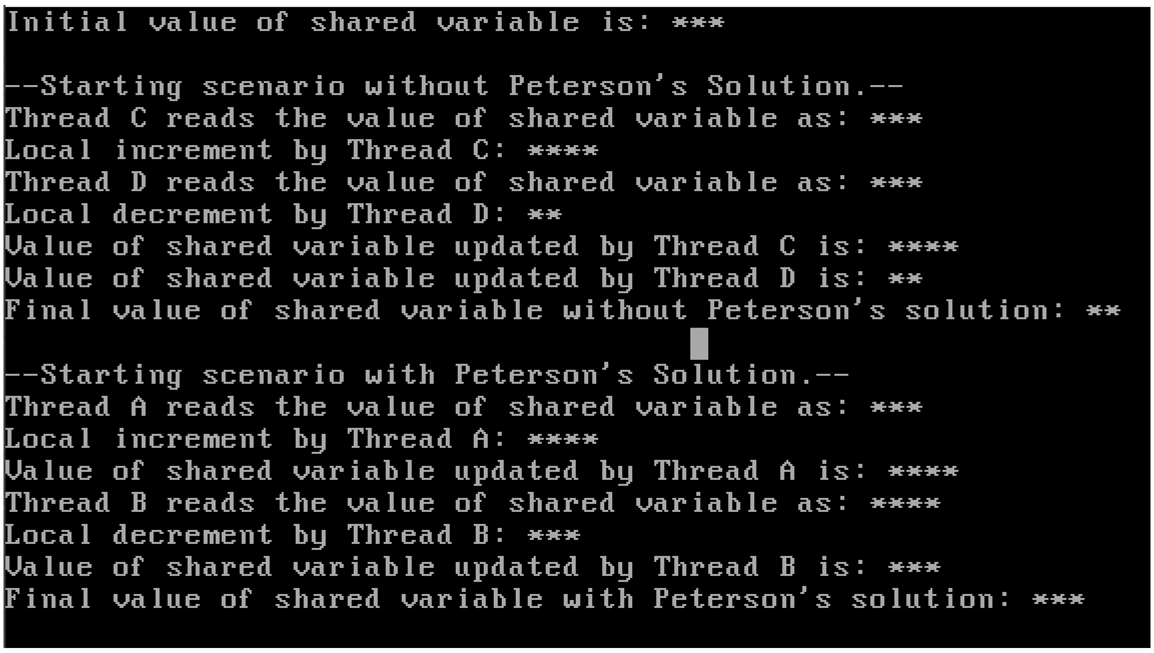
-> In order to send threads to sleeping, I used a very long empty for loop.

-> Printing variable values directly isn't provided in the template OS. So, in order to print the values of variables, I used for loop and printed '*' character as much as the values.

-> Functions coming to an end were causing error so there are one infinite while loop at the end of every function.

-> GlobalDescriptorTable and TaskManager variables are carried out from main function to global scope.

RUNNING AND RESULT



```
Initial value of shared variable is: ***

--Starting scenario without Peterson's Solution.--
Thread C reads the value of shared variable as: ***
Local increment by Thread C: ****
Thread D reads the value of shared variable as: ***
Local decrement by Thread D: **
Value of shared variable updated by Thread C is: ****
Value of shared variable updated by Thread D is: **
Final value of shared variable without Peterson's solution: **

--Starting scenario with Peterson's Solution.--
Thread A reads the value of shared variable as: ***
Local increment by Thread A: ****
Value of shared variable updated by Thread A is: ****
Thread B reads the value of shared variable as: ****
Local decrement by Thread B: ***
Value of shared variable updated by Thread B is: ***
Final value of shared variable with Peterson's solution: ***
```

-> Firstly, initial value of shared variable, which is 3, is printed.

-> Scenario without Peterson's Solution starts. Thread C reads shared variable's value and assigns it to its local variable. Then goes to sleep.

-> Thread D starts and reads shared variable's value and assigns it to its local variable. As you can see, it reads it as 3 although Thread C should've incremented it.

-> Both threads print updated values of the shared variable and final result is printed. As you can see, Thread C's incrementation made no effect and is overwritten.

-> Scenario with Peterson's Solution starts. Thread A reads shared variable's value and assigns it to its local variable. Then it goes to sleep but Thread B doesn't start during this sleep. Because Peterson's Algorithm provides safety. Coming back from sleep, Thread A assigns new value of the shared variable and ends.

-> Then, Thread B starts and reads shared variable's value. As you can see, it reads it as 4, which is incremented by Thread A. Then it assigns it to its local variable, goes to sleep for a little and assigns it back to the shared variable.

-> Finally, final result is correct. Sharec variable's value stays the same after one increment and one decrement.

Notes about running: In first part, the running orders of Thread C and D can rarely be opposite randomly. It doesn't change it giving the wrong result anyway. Also, the printed texts can rarely tangle with each other in first part. It can be considered as yet another indication of race conditions.