Operating System Lab-7

Dining Philosopher Problem

This assignment requires you to implement a solution to the famous Dining Philosophers problem using multithreading and concurrent programming techniques.

Task 1: Creation of simple threads

In this part, you will create a main program that takes a single command line parameter (an integer) to indicate the number of threads to be created. You can finish this part by following these steps:

(1) Process the command line parameter to retrive the value of nthreads.

(2) Print your name, followed by "Assignment 7: # of threads = " along with the value of nthreads.

(3) Your main function should invoke the function:

void creatPhilosophers(int nthreads)

where nthreads is the parameter indicating the number of threads to be created. To create each thread, the function philosopherThread(void * pVoid) will be called, and passed a pointer to an integer containing the index of the thread.

(4) The philosopherThread() function should produce the phrase "This is philosopher X" for each philosopher thread, where X is the thread's index. The thread function can then just return NULL.

(5) Using the pthread_join() method, the main thread in the function creatPhilosophers() must wait until all philosopher threads have finished. The function creatPhilosophers() will print out a message saying "N

threads have been completed/joined successfully!" before returning after all generated threads have successfully connected.

Note: - Please test with values 5 and 20 for the value of nthread.

Desired Output Format:

```
PS C:\Users\Dell\Desktop\Dining Philosopher> gcc -o dphil1 dphil1.c -lpthread
PS C:\Users\Dell\Desktop\Dining Philosopher> ./dphil1 10
Shubham Assignment 7: # of threads = 10
This is philosopher 0
This is philosopher 2
This is philosopher 3
This is philosopher 4
This is philosopher 1
This is philosopher 8
This is philosopher 6
This is philosopher 7
This is philosopher 5
This is philosopher 9
10 threads have been completed/joined successfully!
PS C:\Users\Dell\Desktop\Dining Philosopher>
```

Task 2: Use Multiple Mutexes to Manage Chopsticks.

You will simulate the behaviors of the philosophers in this section, where each philosopher goes through a cycle of "thinking," "picking up chopsticks," "eating," and "putting down chopsticks," as depicted below. To implement these four matching stages, you must create four distinct functions, each of which has the following deterministic prototypes:

- **void thinking();**
- **void pickUpChopsticks(int threadIndex);**
- **void eating();**
- **void putDownChopsticks(int threadIndex);**

The main function of this assignment is pickUpChopsticks(), which mimics the action of the "pick up chopsticks" stage. Keep in mind that each chopstick is shared by two nearby philosophers, making them a shared resource. We cannot let a philosopher to pick up a chopstick that is already in their neighbor's hand

since it would be a race condition. We may use each chopstick as a mutex lock to solve this issue. Prior to eating, each philosopher must lock both their right and left chopsticks.

Only after the acquisitions of both locks (representing the two chopsticks) are successful, may this philosopher begin to eat. After finishing eating by calling the function eating(), the philosopher will invoke the function putDownChopsticks() to release both locks (i.e., chopsticks), and exit the program (that is, for Part 2, each philosopher only goes through the above cycle once).

Both functions eating() and thinking() can be easily simulated by invoking a sleep function, such as usleep(), placed between two output statements "Philosopher #X: starts eating/thinking", and "Philosopher #X: ends eating/thinking", respectively. However, we should not use a fixed value for every invocation of the sleep function to emulate the different length of activites of the philosophers. Instead, we need to utilize a random number between 1 to 500. You can use function random() to obtain a random value, which could be initilized using the srandom() function using a proper seed value for the random value generator.

Note 1: You need to test your program with both 5 threads and 20 threads, and add the output in the REPORT.txt file.

Note 2: Moreover, please run the program 100 times, and report the number of deadlocks encountered in your REPORT.txt.

Desired Output:

```
PS C:\Users\Dell\Desktop\Dining Philosopher> gcc -o dphil2 dphil2.c -pthread
PS C:\Users\Dell\Desktop\Dining Philosopher> ./dphil2
Philosopher #0: starts thinking
Philosopher #3: starts thinking
Philosopher #1: starts thinking
Philosopher #2: starts thinking
Philosopher #4: starts thinking
Philosopher #4: starts picking up chopsticks
Philosopher #4: starts eating
Philosopher #3: starts picking up chopsticks
Philosopher #1: starts picking up chopsticks
Philosopher #1: starts eating
Philosopher #2: starts picking up chopsticks
Philosopher #0: starts picking up chopsticks
Philosopher #1: ends eating
Philosopher #4: ends eating
Philosopher #3: starts eating
Philosopher #0: starts eating
Philosopher #3: ends eating
Philosopher #0: ends eating
Philosopher #2: starts eating
Philosopher #2: ends eating
PS C:\Users\Dell\Desktop\Dining Philosopher>
```

Task 3: In this part, the deadlock problem in the solution for Part 2 will be addressed by using only one global mutex object for synchronization. In addition, the solution for this part needs to enforce the order of eating, where the philosopher with index 0 should eat first, followed by the one with index 1, and so on. The ordered eating can be achived with a conditional variable, which works together with nextIndex to indicate which philosopher should eat next. Initially, before creating all philosopher threads, nextIndex is initialized to 0.

Here, when the i^{th} philosopher thread completes "thinking," the thread should check whether it is the ith philosopher's turn to eat or not prior to obtaining the chopsticks. If nextIndex==i, the ith philosopher will grab both chopsticks and begin eating. Otherwise, if nextIndex!=i, the thread should wait on the shared conditional variable. When a philosopher thread completes "eating," it will increment the value of nextIndex and release the lock to wake up all threads that are waiting on the conditional variable. Depending on the value of nextIndex, only one of the threads can move forward based on the determined order, where other threads will have to go back to waiting again.

Note: - Test the program with 5 and 20 threads. Run this program 10 times and confirm that there is no deadlock. Please include the output for one running of the case of 5 threads in the `REPORT2.txt`.

Desired Output:

```
PS C:\Users\Dell\Desktop\Dining Philosopher> gcc -o dphil3 dphil3.c -pthread
PS C:\Users\Dell\Desktop\Dining Philosopher> ./dphil3
Philosopher #0: starts thinking
Philosopher #3: starts thinking
Philosopher #1: starts thinking
Philosopher #2: starts thinking
Philosopher #4: starts thinking
Philosopher #0: starts eating
Philosopher #0: ends eating
Philosopher #1: starts eating
Philosopher #1: ends eating
Philosopher #2: starts eating
Philosopher #2: ends eating
Philosopher #3: starts eating
Philosopher #3: ends eating
Philosopher #4: starts eating
Philosopher #4: ends eating
PS C:\Users\Dell\Desktop\Dining Philosopher>
```

NOTE:

- The function prototype for pthread_create is as follows:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const

        pthread_attr_t *attr,void *(*start_routine)

        (void *), void *arg);
```

- The function prototype for pthread_join is as follows:

```
#include <pthread.h>


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

- To run the program 100 times and report the number of deadlocks encountered, you can use a simple shell script and record the results in a text file. Here's an example of how you can do this:

1. Create a shell script (e.g., run_100_times.sh) with the following content:

```
#!/bin/bash


count=0


for ((i=0; i<100; i++))
do
   if ./assign4-part2; then
      echo "Run $i: No deadlock"
   else
      echo "Run $i: Deadlock occurred"
```

```
        ((count++))
    fi
done

echo "Number of deadlocks encountered: $count"
```

2. Make the script executable:

   ```
   chmod +x run_100_times.sh
   ```

3. Run the script:

   ```
   ./run_100_times.sh > deadlock_report.txt
   ```