Laboratory 6

Title of the Laboratory Exercise: Solution to Dining Philosopher problem using Semaphore

1. **Introduction and Purpose of Experiment**

   In multitasking systems, simultaneous use of critical section by multiple processes leads to data inconsistency and several other concurrency issues. By solving this problem students will be able to use semaphore for synchronisation purpose in concurrent programs.

2. **Aim and Objectives**

   Aim

   - To develop concurrent programs using semaphores

   Objectives

   At the end of this lab, the student will be able to

   - Use semaphore
   - Apply appropriate semaphores in different contexts
   - Develop concurrent programs using semaphores

3. Experimental Procedure

   i. Analyse the problem statement

   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code

   iii. Implement the algorithm in C language

   iv. Compile the C program

   v. Test the implemented program

   vi. Document the Results

vii. Analyse and discuss the outcomes of your experiment

4. Question

Implement the Dining Philosopher problem using POSIX threads

5. Calculations/Computations/Algorithms

```
philosopher_thread():
1. for i = 1 to 2 // try twice
2.         hungry(philr)
3.         wait(mutex)
4.         wait(fork_left)
5.         wait(fork_right)
6.         eat(philr)
7.         post(fork_right)
8.         post(fork_left)
9.         post(mutex)
10. think(philr)
```

6. Presentation of Results

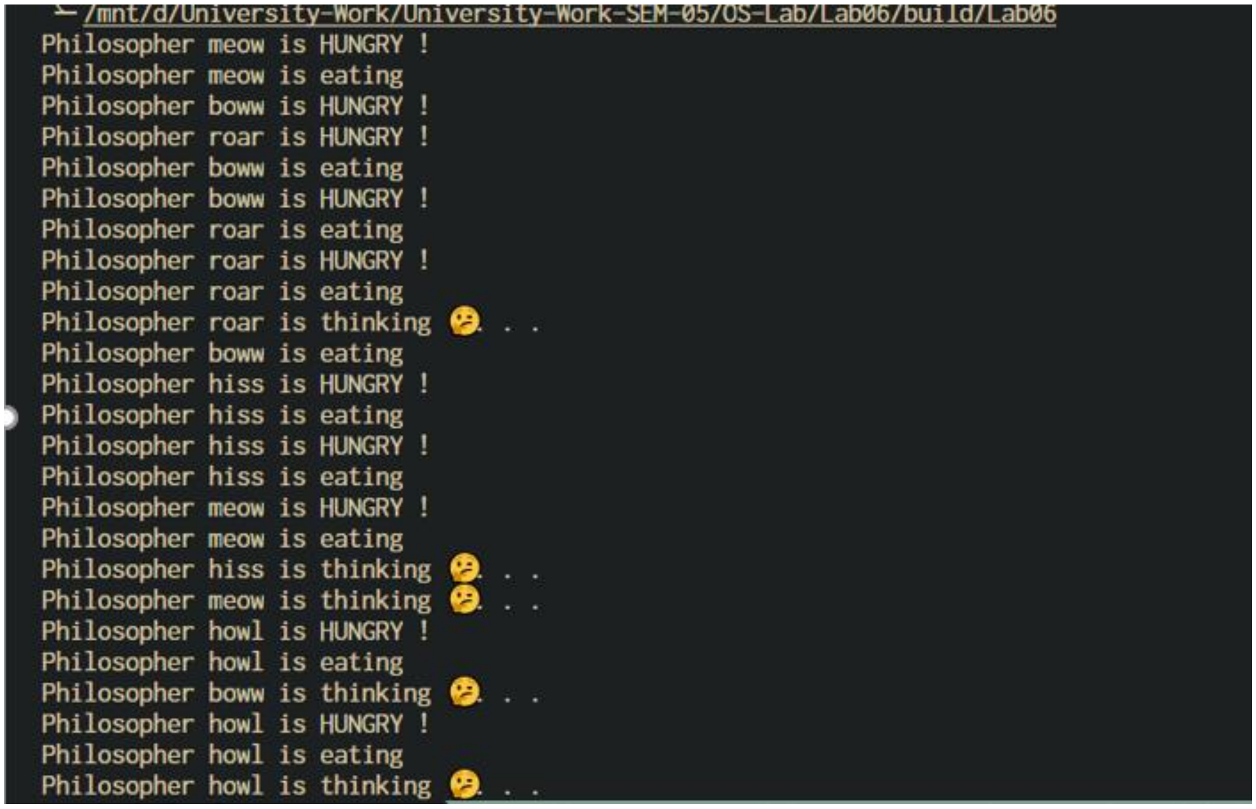
main.cpp

```cpp
#include <array>
#include <iostream>
#include <memory>
#include <string>

#include <pthread.h>
#include <semaphore.h>

struct philosopher {
   pthread_t thread;
   int position;
   sem_t *fork_left, *fork_right;
   sem_t *mutex;
   std::string name;

   philosopher() {}
   philosopher(const std::string name);
```

```
};

philosopher::philosopher(const std::string name) : name(name) {}

constexpr int const num_phil = 5;

void initialize_sem_t(std::array<sem_t, num_phil> &forks, sem_t &mutex) {
   for (auto &fork : forks) {
      sem_init(&fork, 0, 1);
   }

   sem_init(&mutex, 0, forks.size() - 1);
}

void create_philosophers(std::array<philosopher, num_phil> &philosophers, std::array<std::string, num_phil
> names, std::array<sem_t, num_phil> &forks, sem_t &mutex) {
   for (int i = 0; i < philosophers.size(); i++) {
      philosophers.at(i) = philosopher(names.at(i));
      philosophers.at(i).fork_left = &forks.at(i);
      philosophers.at(i).fork_right = &forks.at((i + 1) % forks.size());
      philosophers.at(i).position = i;
      philosophers.at(i).mutex = &mutex;
   }
}

void hungry(std::string name) {
   std::cout << ("Philosopher " + name + " is HUNGRY !\n");
}

void think(std::string name) {
   std::cout << ("Philosopher " + name + " is thinking 🤔 . . .\n");
}

void eat(std::string name) {
   std::cout << ("Philosopher " + name + " is eating\n");
}

void *philosopher_thread(void *args) {
   philosopher *curr = static_cast<philosopher *>(args);

   //  try twice for the fork
   for (int i = 0; i < 2; i++) {
      hungry(curr->name);

      sem_wait(curr->mutex);
      sem_wait(curr->fork_left);
      sem_wait(curr->fork_right);
      eat(curr->name);
      sem_post(curr->fork_left);
      sem_post(curr->fork_right);
```

```
    sem_post(curr->mutex);
  }

  think(curr->name);

  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  std::array<std::string, 5> names{"meow", "boww", "roar", "hiss", "howl"};

  std::array<sem_t, num_phil> forks;
  sem_t mutex;

  // initialize_sem_t the forks and the mutex
  initialize_sem_t(forks, mutex);

  std::array<philosopher, num_phil> philosophers;

  // create the philosophers
  create_philosophers(philosophers, names, forks, mutex);

  // run the philosophers
  for (int i = 0; i < num_phil; i++) {
    pthread_create(&philosophers.at(i).thread, NULL, philosopher_thread, &philosophers.at(i));
  }

  // wait for them to finish
  for (int i = 0; i < num_phil; i++) {
    pthread_join(philosophers.at(i).thread, NULL);
  }
}
```

7. Analysis and Discussions



```
└─ /mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab06/build/Lab06
Philosopher meow is HUNGRY !
Philosopher meow is eating
Philosopher boww is HUNGRY !
Philosopher roar is HUNGRY !
Philosopher boww is eating
Philosopher boww is HUNGRY !
Philosopher roar is eating
Philosopher roar is HUNGRY !
Philosopher roar is eating
Philosopher roar is thinking 🤔 . . .
Philosopher boww is eating
Philosopher hiss is HUNGRY !
Philosopher hiss is eating
Philosopher hiss is HUNGRY !
Philosopher hiss is eating
Philosopher meow is HUNGRY !
Philosopher meow is eating
Philosopher hiss is thinking 🤔 . . .
Philosopher meow is thinking 🤔 . . .
Philosopher howl is HUNGRY !
Philosopher howl is eating
Philosopher boww is thinking 🤔 . . .
Philosopher howl is HUNGRY !
Philosopher howl is eating
Philosopher howl is thinking 🤔 . . .
```

Figure 0-1 OUTPUT

The dining philosopher's problem illustrates non-composability of low-level synchronization primitives like semaphores. It is a modification of a problem posed by Edsger Dijkstra.

Five meow, boww, roar, hiss, and howl spend their time thinking and eating spaghetti. They eat at a round table with five individual seats. For eating each philosopher needs two forks (the resources). There are five forks on the table, one left and one right of each seat. When a philosopher cannot grab both forks it sits and waits. Eating takes random time, then the philosopher puts the forks down and leaves the dining room. After spending some random time thinking about the nature of the universe, he again becomes hungry, and the circle repeats itself.

It can be observed that a straightforward solution, when forks are implemented by semaphores, is exposed to deadlock. There exist two deadlock states when all five philosophers are sitting at the table holding one fork each. One deadlock state is when each philosopher has grabbed the fork left of him, and another is when each has the fork on his right.

8. Conclusions

Things to Consider:

We need to ensure that access to the forks is limited. The basic idea is that a philosopher will think while waiting to get a hold of the fork to his left and right. Then he'll eat and release the forks allowing the philosophers to his left and right to eat.

Since the day is shared by all philosophers, we'll model each philosopher as a thread so they can run at "the same" time.

To do this, we'll need to ensure we don't get caught in a deadlock. This can happen if each philosopher grabs the fork to their left, preventing anyone from grabbing the fork to their right. If that happens each philosopher thread would block waiting for the fork on the right.

9. Comments

1. Limitations of Experiments

Semaphores is one of the solutions of the problem, due to its implementation it is slow, hence a better algorithm or interface library should be used, such as monitor mechanism.

2. Limitations of Results

Semaphores are impractical for large scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.

3. Learning happened

We learnt how to solve the dining philosopher's problem using semaphores in the POSIX thread library.

4. Recommendations

Simulate the program for higher number of iterations to get a stable concurrency. For better performance use a Monitor mechanism to solve the problem.