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

   The algorithm is as follows:

   - `take_fork(int)`

   This function represents the steps that need to be taken to obtain a resource. For the dining philosopher problem, the resource that needs to be obtained is the fork.

   ```
   Step 1: Start

   Step 2: wait on semaphore forks[fork], which decreases the
   value of the semaphore, indicating that the resource is
   consumed

   Step 3: Stop
   ```

   - `put_fork(int)`

   This function represents the steps that need to be taken to release a resource. For the dining philosopher problem, the resource that needs to be released is the fork

   ```
   Step 1: Start

   Step 2: post semaphore fork[fork], which increases the value
   of the semaphore, indicating that the resource is released

   Step 3: Stop
   ```

   - `eat(int)`

   This function represents the steps that need to be taken after the required resources are obtained. In case of the dining philosopher problem, the resources that need to be obtained are the left and right forks.

   ```
   Step 1: Start

   Step 2: display message

   Step 3: Stop
   ```

   - `act(arg)`

   This function drives the actions of the philosophers.

   ```
   Step 1: Start
   ```

Step 2: type cast arg to int to get the int representing the philosopher

Step 3: while true, repeat

3.1: if philosopher is the last philosopher, then

3.1.1: take right fork using function take_fork((philosopher + 1) % MAX_PHILOSOPHERS)

3.1.2: take left fork using function take_fork(philosopher)

3.2: else

3.2.1: take left fork using function take_fork(philosopher)

3.2.2: take right fork using function take_fork((philosopher + 1) % MAX_PHILOSOPHERS)

3.2: eat(philosopher)

3.3: put down left fork using put_fork(philosopher)

3.4: put down right fork using put_fork((philosopher + 1) % MAX_PHILOSOPHERS)

Step 4: Stop

- main()

  The main function.

  Step 1: Start

  Step 2: define the threads for each philosopher

  Step 3: initialize the semaphores for each fork

  Step 4: create and spawn thread, each thread executes the act function and the parameter is i

  Step 5: join all threads

  Step 6: Stop

The program is:

```c
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>

#define MAX_PHILOSOPHERS 5

void take_fork(int);
void put_fork(int);
void eat(int);

// binary semaphores
sem_t forks[MAX_PHILOSOPHERS];

/**
 * Function which drives the philosophers eating.
 *
 * @param arg The int identifying the philosopher
 */
void* act(void* arg) {
    int philosopher = *(int*) arg;

    while (true) {
        printf("Philosopher %d is thinking\n", philosopher);

        // first take forks

        // this is key part to avoid deadlock. For one of the philosopher's
        // change the order of taking forks.
        if (philosopher == MAX_PHILOSOPHERS - 1) {
            // take right fork first
            take_fork((philosopher + 1) % MAX_PHILOSOPHERS);
            // take left fork
            take_fork(philosopher);
        } else {
            // take left fork first
            take_fork(philosopher);
            // take right fork
            take_fork((philosopher + 1) % MAX_PHILOSOPHERS);
        }

        // once forks are taken, eat
        eat(philosopher);
        sleep(2);

        // putdown forks
        put_fork(philosopher);
        put_fork((philosopher + 1) % MAX_PHILOSOPHERS);
    }

}

/**
 * Function that represents steps taken to obtain a resource,
 * in this case, a fork.
 *
 * @param fork The fork to take.
 */
void take_fork(int fork) {
    sem_wait(&forks[fork]);
}

/**
 * Function that represents steps taken to release a resource,
 * in this case, a fork.
 *
 * @param fork The fork to release.
```

```
 */
void put_fork(int fork) {
    sem_post(&forks[fork]);
}

/**
 * Function that represents the steps taken once all the
 * required resources (in this case, forks) are obtained.
 * For the dining philosopher problem, it is to eat.
 *
 * @param philosopher The philosopher that obtained both
 * the forks.
 */
void eat(int philosopher) {
    printf("Philosopher %d is eating\n", philosopher);
}

int main() {
    // creating threads
    pthread_t threads[MAX_PHILOSOPHERS];

    // initializing semaphores
    for (int i = 0; i < MAX_PHILOSOPHERS; i++)
        sem_init(&forks[i], 0, 1);

    // creating and spawning threads
    for (int i = 0; i < MAX_PHILOSOPHERS; i++)
        pthread_create(&threads[i], NULL, act, &i);

    // joining threads
    for(int i = 0; i < MAX_PHILOSOPHERS; i++)
        pthread_join(threads[i], NULL);
}
```

Figure 1: Source code

6. Presentation of Results

```
/mnt/d/RUAS-sem-05/OS/lab6 main*
❯ gcc -pthread lab6.c

/mnt/d/RUAS-sem-05/OS/lab6 main*
❯ ./a.out
Philosopher 1 is thinking
Philosopher 1 is eating
Philosopher 3 is thinking
Philosopher 3 is eating
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is thinking
Philosopher 3 is thinking
Philosopher 4 is eating
Philosopher 2 is eating
Philosopher 4 is thinking
Philosopher 4 is eating
Philosopher 2 is thinking
Philosopher 1 is eating
Philosopher 4 is thinking
Philosopher 4 is eating
Philosopher 1 is thinking
Philosopher 1 is eating
^C

/mnt/d/RUAS-sem-05/OS/lab6 main* 7s
❯
```

Figure 2: Compiling and running

7. Analysis and Discussions

The dining philosopher's problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. The following is the problem statement:

"Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks"

The amount of spaghetti is assumed to infinite and the philosophers never feel sated. A simple algorithm for this problem is:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

But this approach can lead to a deadlock if all the philosophers pick up their left forks at the same time. This can be solved by changing the order of taking the forks for any one of the philosophers. This approach has been implemented in the program.

8. Conclusions

The dining and drinking philosopher's problem are a very old and important problem in the distributed computing field. It was first introduced by Dijkstra and then used by many other researches as a general problem for illustrating mutual exclusion and resource sharing and allocation problem. A lot of algorithms have been introduced to resolve this problem with many options and assumptions which makes each proposed algorithm suitable for specific applications. It was introduced the problem with some of the fundamental and very old solutions for it in the Introduction section. The problem is implemented in C. It was executed and verified.

9. Comments

1. Limitations of Experiments

The perfectness of the implemented solution cannot be experimentally verified.

2. Limitations of Results

The results do not show the current state of the other philosophers in code.

3. Learning happened

The method to develop the Dining Philosopher problem using Semaphore and POSIX threads was learned. The applications and contexts of semaphores to develop concurrent programs was also learned.