KING ABDULAZIZ UNIVERSITY
THE COLLEGE OF ENGINEERING

OPERATING SYSTEMS

EE463

LAB PROJECT – THE DINING PHILOSOPHERS

STUDENT NAME: SAMER SHOUKAT KHAN

ID: 1306219

SUBMISSION DATE: 29/05/2015

SUBMITTED TO

ENG. TURKI GARI

THE DINING PHILOSOPHER PROBLEM

The dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

PROBLEM STATEMENT

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. (An alternative problem formulation uses rice and chopsticks instead of spaghetti and forks).

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has 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 he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply is assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher will not starve; i.e., can forever continue to alternate between eating and thinking, assuming that any philosopher cannot know when others may want to eat or think.

THINGS TO CONSIDER

We need to ensure that access to the chopsticks is limited. The basic idea is that a philosopher will think while waiting to get a hold of the chopsticks 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 chopstick to their left, preventing anyone from grabbing the chopstick to their right. If that happens each philosopher thread would block waiting for the chopstick on the right.

THE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

typedef struct {
  int position;
  int count;
  sem_t *forks;
  sem_t *lock;
} params_t;

void initialize_semaphores(sem_t *lock, sem_t *forks, int
num_forks);
void run_all_threads(pthread_t *threads, sem_t *forks,
sem_t *lock, int num_philosophers);

void *philosopher(void *params);
void think(int position);
void eat(int position);

int main(int argc, char *args[])
{
  int num_philosophers = 5;

  sem_t lock;
```

```
  sem_t forks[num_philosophers];
  pthread_t philosophers[num_philosophers];

  initialize_semaphores(&lock, forks, num_philosophers);
  run_all_threads(philosophers,        forks,        &lock,
num_philosophers);
  pthread_exit(NULL);
}

void initialize_semaphores(sem_t *lock, sem_t *forks, int
num_forks)
{
  int i;
  for(i = 0; i < num_forks; i++) {
    sem_init(&forks[i], 0, 1);
  }

  sem_init(lock, 0, num_forks - 1);
}

void  run_all_threads(pthread_t  *threads,  sem_t  *forks,
sem_t *lock, int num_philosophers)
{
  int i;
  for(i = 0; i < num_philosophers; i++) {
    params_t *arg = malloc(sizeof(params_t));
    arg->position = i;
    arg->count = num_philosophers;
    arg->lock = lock;
    arg->forks = forks;

    pthread_create(&threads[i],  NULL,  philosopher,  (void
*)arg);
  }
}
```

```
void *philosopher(void *params)
{
  int i;
  params_t self = *(params_t *)params;

  for(i = 0; i < 3; i++) {
    think(self.position);

    sem_wait(self.lock);
    sem_wait(&self.forks[self.position]);
    sem_wait(&self.forks[(self.position     +     1)     %
self.count]);
    eat(self.position);
    sem_post(&self.forks[self.position]);
    sem_post(&self.forks[(self.position     +     1)     %
self.count]);
    sem_post(self.lock);
  }

  think(self.position);
  pthread_exit(NULL);
}

void think(int position)
{
  printf("Philosopher %d thinking...\n", position);
}

void eat(int position)
{
  printf("Philosopher %d eating...\n", position);
}
```

❖ The params_t Struct

POSIX threads take a single `void *` parameter. However, we need to pass in four values to each of the threads. We need to pass in the position of the current philosopher at the table, the total number of philosophers, the semaphore for the critical region and the semaphores for the forks. To make this possible, we create a simple struct called `params_t` to wrap these values.

❖ The **main** Function

We begin by defining a few semaphores. We need one semaphore for each fork and one for controlling the critical region. Then we setup and run all of the philosopher threads. Finally, we call `pthread_exit` to wait for all threads to complete.

❖ The initialize_semaphores Function

We initialize the forks by setting their initial value to 1. We could have used mutexes for these, but since they're really just binary semaphores I thought I'd stick to one type of lock throughout.

Next, we initialize the lock semaphore. This will be used to wrap the calls to`eat` in a critical region. We know that only two philosophers can eat at the same time (since there are 5 forks and 2 are required to eat). We could initialize the lock to 2 and it would technically be correct. However, I like to do the simplest thing to avoid deadlock. The simplest thing is to not let ALL philosophers grab forks. So instead of 2, I initialized it to 4; one less than the number of philosophers.

❖ The run_all_threads Function

This function spawns the philosopher threads. It creates an instance of`params_t` to pass to each thread and calls `pthread_create` for each of them. Each thread will call the philosopher function passing it's instance of`params_t` as the argument.

❖ The philosopher Function

This is the meat and potatoes of the app. Here is where each philosopher lives out their day. The interesting part is surrounding the call to eat.

First, we wait on the lock. This is the start of the critical region and the part that ensures that no more than four philosophers are attempting to pick up a fork at any given time. Next we wait on the left and right forks. Since these semaphores are initialized to 1, we can be sure that we are the only one that's using either of them at the moment. Knowing that, we eat.

Finally, we post to the forks indicating we're done eating and post to the lock to exit the critical region to allow other philosophers to eat.

SUMMARY

With a little effort, we've got a working solution to the dining philosophers problem. There are many proposed solutions for this particular problem as attempted by programmers through many years. The code and solution presented are one of the simplest yet most effective.

REFERENCES

1. "Dining Philosophers Problem." *Wikipedia*. Wikimedia Foundation, n.d. Web. 29 May 2015. http://en.wikipedia.org/wiki/Dining_philosophers_problem
2. "Pseudomuto.com." *Dining Philosophers in C ·*. N.p., n.d. Web. 29 May 2015. http://pseudomuto.com/development/2014/03/01/dining-philosophers-in-c/