

## Dining philosophers

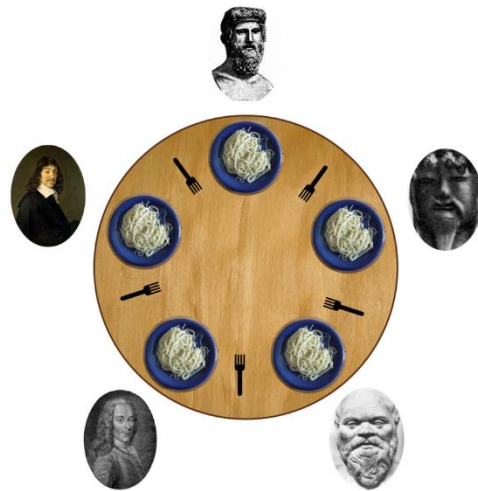
Parallel Programming-Project 2 (Mingsong Chen 2018 Autumn)

Weiwen Chen 10152510217

### Problem Description

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.



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

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

1. To make us understand more about Parallel Programming, This Project around the [Dining philosophers](#). Then we have some coding jobs.
2. Programming Language must be IEEE Posix Thread and Language C(multi-threads). The forks are shared variates, so this needs realize mutual exclusion by mutex or semaphore. Role: we use semaphore in this experiment.
3. Through Dining Philosophers problems, I want to investigate everyone's understanding of what causes Deadlock. I hope everyone is able to realize a primitive Philosopher Problem which may cause Deadlock. The input of the program is able to assign the number of philosophers. Each philosopher has 3 status {thinking, trying, eating}, which means thinking, trying to get a fork and eating spaghetti, respectively. Role that the time of thinking and eating is between  $[0.01, 0.1]$ s. Deadlock means that every one is at the status of trying. Please output the status sequence.

4. There are many methods to avoid Deadlock. Please List 2 of them and do the programming realization(output status sequence).

## Test Method(Input)

We have three modules

```
-normal # modules which may cause deadlock
-method1 # 1st method of solving
-method2 # 2nd method of solving
```

The number of philosophers can be assigned by -n

Example

```
./philosophoer -normal -n 'number'
./philosophoer -method1 -n 'number'
./philosophoer -method2 -n 'number'
```

## Abstract Design

All codes are in philosopher.c.

We have some global defines:

```
#define sleep_time 0.1

int phi_cnt; /*count of philosophers*/
#define left_fork (idx % phi_cnt)
#define right_fork ((idx+1) % phi_cnt)
char* method;
sem_t forks[55];
```

And main() function is modified from pthread Hello from the course book.

```
int main(int argc, char* argv[]){
    // get variates from command
    method = argv[1];
    sscanf(argv[3], "%d", &phi_cnt);
    // init forks (semaphore)
    for (int i = 0; i < phi_cnt; i++){
        sem_init(&forks[i], 0, 0);
        sem_post(&forks[i]);
    }
    // init philosophers (pthread), and then begin ♂
    pthread_t* philosophers = malloc(phi_cnt * sizeof(pthread_t));
    for (long i = 0; i < phi_cnt; i++){
        pthread_create(&philosophers[i], NULL, dark_deep_fantasy, (void*)i);
    }
    sem_destroy(forks);
    for (int i = 0; i < phi_cnt; i++){
```

```

        pthread_join(philosophers[i], NULL);
    }
    free(philosophers);
    return 0;
}

```

The function `deep_dark_fantasy()` choose the way how each philosopher works. As assigned, There are 3 methods(including the deadlock one). In next session I will tell how we avoid deadlock in a detail way.

- `philosopher_normal(long rnk);`
- `philosopher_method1(long rnk);`
- `philosopher_method2(long rnk);`

## How to avoid Deallock - Detail

### Method1: Arbitrator solution

One approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. In order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time until the philosopher has picked up both of their forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex. In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of their neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

```

void wait_2_forks(long idx){
    sem_trywait(&forks[left_fork]);
    sem_trywait(&forks[right_fork]);
}
void free_2_forks(long idx){...}
void philosopher_method1(long idx){
    while(1){
        printf("phisolopher: %ld: thinking.\n", idx);
        sleep(sleep_time);
        printf("phisolopher: %ld: trying.\n", idx);
        wait_2_forks(idx);
        printf("phisolopher: %ld: eating.\n", idx);
        sleep(sleep_time);
        free_2_forks(idx);
    }
}

```

Most times we donot use `sem_trywait()`. But this works whe a philosopher want to guarantee 2 forks are both available. I think I dont need extra comments, cause it read just like nature language.

## Method2: Resource hierarchy solution

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks they plan to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so they will be able to eat using two forks.

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.

In this method I made 2 defines:

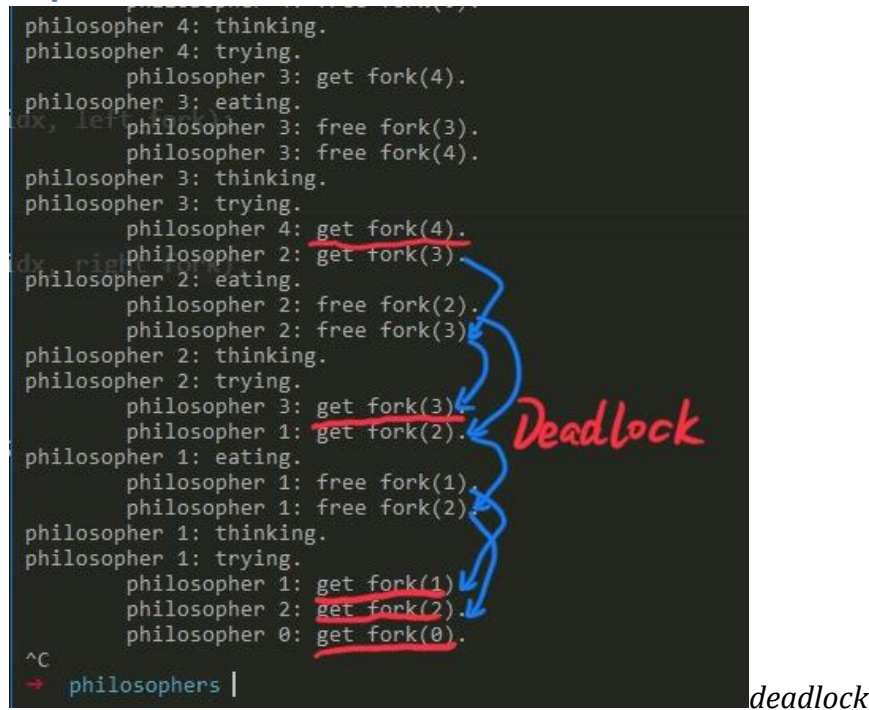
```
#define lower_fork (idx+1==phi_cnt ? 0 : idx)
#define upper_fork (idx+1==phi_cnt ? idx : idx+1)

void wait_lower_fork(long idx){...}
void wait_upper_fork(long idx){...}
void free_lower_fork(long idx){...}
void free_upper_fork(long idx){...}

void philosopher_method2(long idx){
    while(1){
        printf("philosopher %ld: thinking.\n", idx);
        sleep(sleep_time); /* thinking */
        printf("philosopher %ld: trying.\n", idx);
        wait_lower_fork(idx);
        wait_upper_fork(idx);
        printf("philosopher %ld: eating.\n", idx);
        sleep(sleep_time); /* eating */
        free_lower_fork(idx);
        free_upper_fork(idx);
    }
}
```

## Experimental Result

```
philosopher 4: thinking.
philosopher 4: trying.
  philosopher 3: get fork(4).
philosopher 3: eating.
  philosopher 3: free fork(3).
  philosopher 3: free fork(4).
philosopher 3: thinking.
philosopher 3: trying.
  philosopher 4: get fork(4).
  philosopher 2: get fork(3).
philosopher 2: eating.
  philosopher 2: free fork(2).
  philosopher 2: free fork(3).
philosopher 2: thinking.
philosopher 2: trying.
  philosopher 3: get fork(3).
  philosopher 1: get fork(2).
philosopher 1: eating.
  philosopher 1: free fork(1).
  philosopher 1: free fork(2).
philosopher 1: thinking.
philosopher 1: trying.
  philosopher 1: get fork(1).
  philosopher 2: get fork(2).
  philosopher 0: get fork(0).
^C
→ philosophers |
```



This is the part result of `./philosophoer -normal -n 5`. Obiviously we can observe a deadlock.

For the other two methods, the program kepted running for minutes. There is a `out.txt` from `./philosophoer -method2 -n 5 >>out.txt`.

I ran this for 2s, `out.txt` grow to 20 MB. Wow, if it does not cause deadlock, it will cause deadloop. This file is in the attatchment. It means nothing to pause the deadloop output here.

## Analysis

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real

computer programming when multiple programs need exclusive access to shared resources. These issues are studied in concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Complex systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, and data corruption are to be avoided.

This Experiment is a good practise for semaphore and pthread. The method which can avoid Deadlock is assigned at [wiki](#). It didnt cost me a lot of time to undertand the problem, either coding. Most time is costed at learning how makefile works and install unbuntu in win10. After the environment is down and the problem is clear, Coding performed pretty fast.

## Aknowledge

Thanks for Teaching of Professor [Mingsong Chen](#) this semester. I love his talk show on class.

## Reference

- [Dining philosophers](#)
- [Philosopher Starving Problem by Semaphore](#)
- [WayKwin/LittleExercise](#)
- [semaphore.h - semaphores \(REALTIME\)](#)

## Appendix

Submit: homeworkecnu@163.com

- source code(comment key codes)
- makefile
- .zip

Opensource: [github](#)

Science Building

2018.12.26