Assignment7 Report

姓名：Yitong WANG(王奕童) 11910104@mail.sustech.edu.cn

学号：11910104

实验课时段：周五5-6节

实验课教师：Yun SHEN(沈昀) sheny@mail.sustech.edu.cn

实验课SA:

- Yining TANG(汤怡宁) 11811237@mail.sustech.edu.cn
- Yushan WANG(王宇杉) 11813002@mail.sustech.edu.cn

Q1. Deadlock

（1）Is the operating system in a safe state? Why?

This OS is in a safe state, since we can find an order of process execution to let them get the resource one by one.

Currently, the resource condition is enumerated as following:

- P1: capture 0A 2B 1C 0D，require 2A 1B 0C 0D
- P2：capture 0A 1B 0C 1D，require 0A 0B 2C 1D
- P3：capture 0A 0B 1C 0D，require 1A 0B 0C 1D
- P4：capture 1A 1B 0C 0D，require 0A 1B 1C 1D
- Free Resource：1A 0B 1C 2D

We can see the free resource can meet the requirement of P3.

Thus we execute P3 first and the free resource become:

1A 0B 2C 2D

Similarly, we can execute P2 and the free resource become:

1A 1B 2C 3D

After that, P4 can also be run and make the free resource into:

2A 2B 2C 2D

Finally, P1 can be executed.

Therefore, the execution order is (P3, P2, P4, P1), and the system is in a safe mode.

（2） If P4 requests (0,0,1,1), please run the Banker’s algorithm to determine if the request should be granted.

i. check the need and request:

request(P4) = (0, 0, 1, 1) <= (0, 1, 1, 1) = need(P4);

passed.

ii. check the available and request:

request(P4) = (0, 0, 1, 1) <= (1, 0, 1, 2) = available(P4)

passed

iii. check the deadlock.

First assume the allocation is successful.

Then the resource condition is changed into:

- P1: capture 0A 2B 1C 0D，require 2A 1B 0C 0D
- P2：capture 0A 1B 0C 1D，require 0A 0B 2C 1D
- P3：capture 0A 0B 1C 0D，require 1A 0B 0C 1D
- P4：capture 1A 1B 1C 1D，require 0A 1B 0C 0D
- Free Resource：1A 0B 0C 1D

We can first allocate free resource to P3. After P3 execution, the resource becomes: 1A 0B 1C 1D.

But it can never allocate resource for any processes among P1, P2 and P4.

Thus it is a unsafe state. The assumption failes and the allocation is not successful.

Thus the request cannot be granted.

(3) Let's assume P4's request was granted anyway (regardless of the answer to question 2). If then the processes request additional resources as follows, is the system in a deadlock state? Why? [10 pts]

The system is not in a deadlock state.

After resource allocation, the resource condition is:

Then the resource condition is changed into:

- P1: capture 0A 2B 1C 0D，require 2A 1B 0C 0D
- P2：capture 0A 1B 0C 1D，require 0A 0B 1C 0D
- P3：capture 0A 0B 1C 0D，require 1A 0B 0C 0D
- P4：capture 1A 1B 1C 1D，require 0A 1B 0C 0D
- Free Resource：1A 0B 0C 1D

We first allocate 1A resource to P3. P3 can be executed normally and the free resource becomes:

1A 0B 1C 1D

Then allocate 1C resource to P2. P2 can be executed normally and the free resource becomes:

1A 1B 1C 2D

Then allocate 1B resource to P4. P4 can be executed normally and the free resource becomes:

2A 2B 2C 3D

Finally, P1 can be executed normally.

Therefore, the execution order is (P3, P2, P4, P1), and the system is in a safe mode.

Q2. Dining philosophers problem

Two types of solutions:

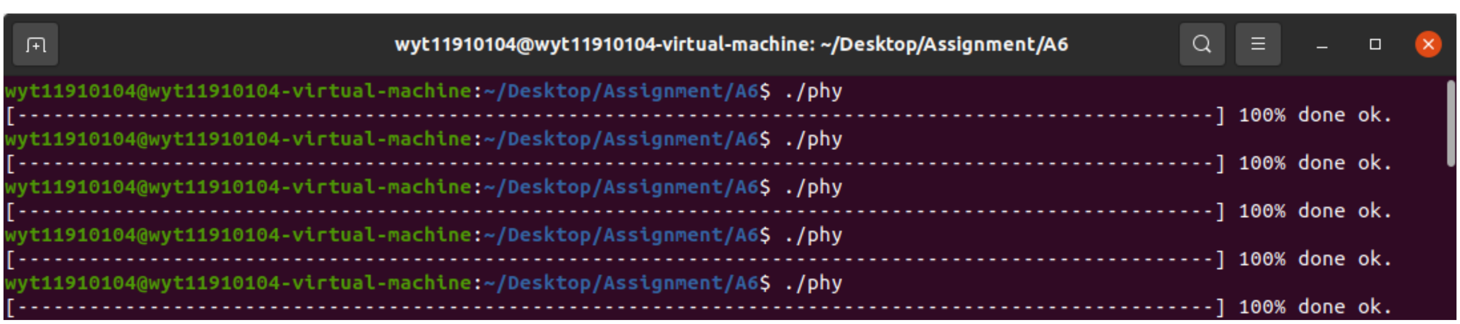
1. Use Sleep-based locks(pthread_mutex_lock)

Explain your design idea: Use one `pthread_mutex_t` to lock the eating status, so that there is only 1 philosophers can eat the spaghetti.

The modified code screenshot:

```c
80 /* Solution 1: Mutex Lock(Line 81 to 102)*/
81 pthread_mutex_t eater_lock;
82 
83 void init() {
84     // write code if you desire.
85 
86 }
87 
88 void wants_to_eat(int p_no) {
89     // fixme
90 
91     pthread_mutex_lock(&eater_lock);
92     pick_right_fork(p_no);
93     pick_left_fork(p_no);
94 
95     eat(p_no);
96 
97     put_left_fork(p_no);
98     put_right_fork(p_no);
99 
100     pthread_mutex_unlock(&eater_lock);
101 
102 }
103 //----------------------end---------------------
```

Running result screenshot:



2. Use ReentrantLock

Explain your design idea:

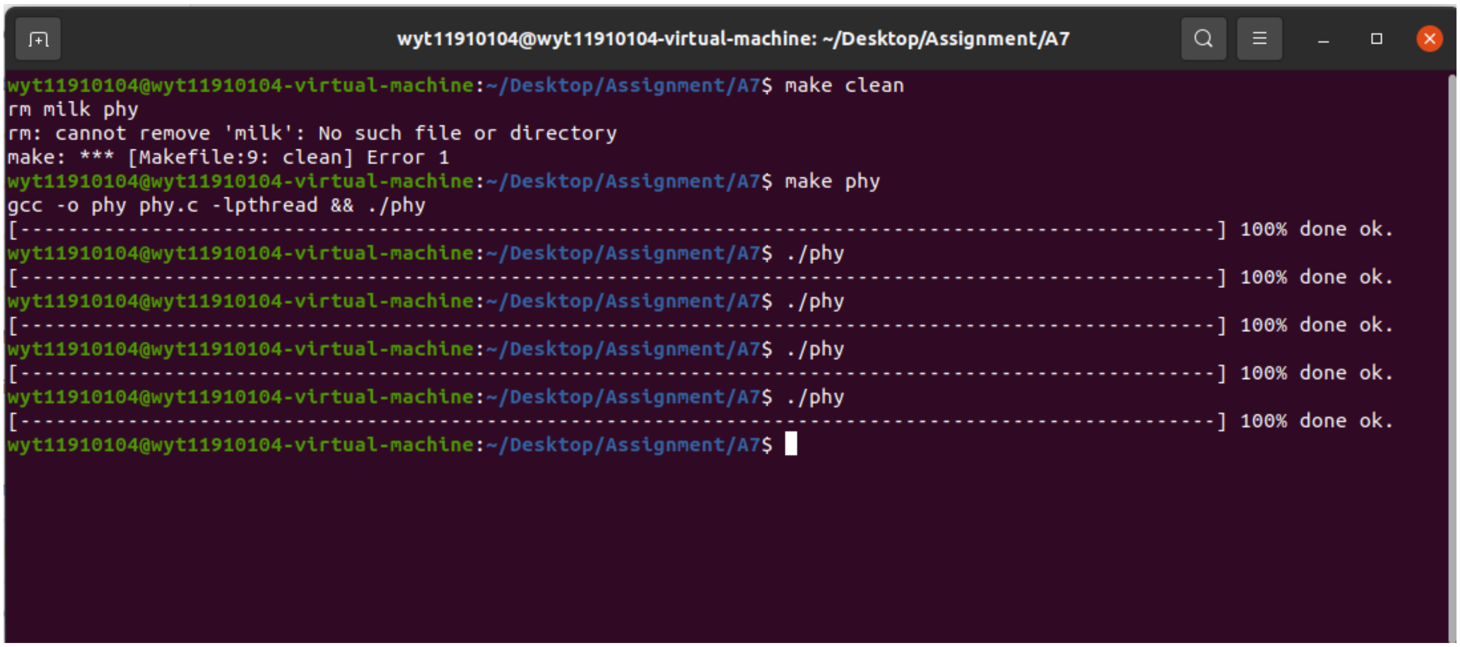
Use 2 locks: 1 mutex lock and 1 conditional lock. For each eating operation, we need to check the forks on the left and right, and waiting until the forks are released. And we use 1 integer array to represent the forks using status.

The modified code screenshots:

```c
//-----------------start---------------------
// you can only modify this part

pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int eating[5];

void init() {
    // write code if you desire.
    for (int i = 0; i < 5; i++)
    {
        eating[i] = 0;
        /* code */
    }

}

```

```c
void wants_to_eat(int p_no) {
    // fixme
    pthread_mutex_lock(&mtx);

    int leftFork = p_no, rightFork = (p_no + 1) % 5;

    while(eating[leftFork] != 0 || eating[rightFork] != 0){
        pthread_cond_wait(&cond, &mtx);
    }

    eating[leftFork] = 1;
    eating[rightFork] = 1;

    pick_left_fork(p_no);
    pick_right_fork(p_no);

    eat(p_no);

    eating[leftFork] = 0;
    put_left_fork(p_no);

    eating[rightFork] = 0;
    put_right_fork(p_no);

    pthread_mutex_unlock(&mtx);
}
```

Running screenshot:



> Reference: https://leetcode.cn/problems/the-dining-philosophers/solution/zhe-xue-jia-jin-can-by-skyshine94-7blq/

Q3. The too much milk problem

Explain your design idea:

Use 2 semaphores to solve this problem. One to handle the milk buying threads(i.e. dad, mom and grandfather) and another to handle the milk drinking thread(i.e. son).

The code screenshots:

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

sem_t sem_buying;
sem_t sem_eating;
pthread_mutex_t fri_lock;

```

```c
void *mom(int *num){
    for(int i=0;i<10;i++){

        sem_wait(&sem_buying);
        pthread_mutex_lock(&fri_lock);
        printf("Mom comes home.\n");
        sleep(rand()%2+1);
        printf("Mom goes to buy milk.\n");
        *num += 1;
        sem_post(&sem_eating);
        if (*num > 2){
            printf("What a waste of food! The fridge can not hold so much milk!\n"
            while(1)printf("TAT~");
        }
        printf("Mom puts milk in fridge and leaves.\n");
        pthread_mutex_unlock(&fri_lock);
    }
}
```

```c
void *dad(int *num){
    for(int i=0;i<10;i++){
        sem_wait(&sem_buying);
        pthread_mutex_lock(&fri_lock);
        printf("Dad comes home.\n");
        sleep(rand()%2+1);
        printf("Dad goes to buy milk.\n");
        *num += 1;
        sem_post(&sem_eating);
        if (*num > 2){
            printf("What a waste of food! The fridge can not hold so much milk!\n"
            while(1)printf("TAT~");
        }
        printf("Dad puts milk in fridge and leaves.\n");
        pthread_mutex_unlock(&fri_lock);
    }
}
```

```c
void *grandfather(int *num){
    for(int i=0;i<10;i++){
        sem_wait(&sem_buying);
        pthread_mutex_lock(&fri_lock);
        printf("Grandfather comes home.\n");
        sleep(rand()%2+1);
        printf("Grandfather goes to buy milk.\n");
        *num += 1;
        sem_post(&sem_eating);
        if (*num > 2){
            printf("What a waste of food! The fridge can not hold so much milk!\n"
            while(1){
                printf("TAT~");
            }
        }
        printf("Grandfather puts milk in fridge and leaves.\n");
        pthread_mutex_unlock(&fri_lock);
    }
```

```c
void *son(int *num){
    for(int i = 0; i < 30 ; i++){
        sem_wait(&sem_eating);
        pthread_mutex_lock(&fri_lock);
        printf("Son comes home.\n");
        if(*num == 0){
            printf("The fridge is empty!\n");
            while(1){
                printf("TAT~");
            }
        }
        printf("Son fetches a milk\n");
        *num -= 1;
        sem_post(&sem_buying);
        printf("Son leaves\n");
        pthread_mutex_unlock(&fri_lock);
    }
}
```

```c
int main(int argc, char * argv[]) {
    srand(time(0));

    int num_milk = 0;
    pthread_t p1, p2, p3, p4;
    sem_init(&sem_buying, 0, 2);
    sem_init(&sem_eating, 0, 0);

    pthread_mutex_init(&fri_lock,NULL);

    // Create two threads (both run func)
    pthread_create(&p1, NULL, mom, &num_milk);
    pthread_create(&p2, NULL, dad, &num_milk);
    pthread_create(&p3, NULL, grandfather, &num_milk);
    pthread_create(&p4, NULL, son, &num_milk);

    // Wait for the threads to end.
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);
    pthread_join(p4, NULL);

    printf("success!\n");
}
```

Running Result:

```
wyt11910104@wyt11910104-virtual-machine: ~/Desktop/Assignment/A7
Son comes home.
Son fetches a milk
Son leaves
Grandfather comes home.
Grandfather goes to buy milk.
Grandfather puts milk in fridge and leaves.
Son comes home.
Son fetches a milk
Son leaves
Mom comes home.
Mom goes to buy milk.
Mom puts milk in fridge and leaves.
Son comes home.
Son fetches a milk
Son leaves
Dad comes home.
Dad goes to buy milk.
Dad puts milk in fridge and leaves.
Dad comes home.
Dad goes to buy milk.
Dad puts milk in fridge and leaves.
Son comes home.
Son fetches a milk
Son leaves
Son comes home.
Son fetches a milk
Son leaves
success!
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Assignment/A7$
```