Operating Systems Hw4

學號 1073007S

姓名 葉致廷

- Part 1.　Explanation

The implementation is based on the pseudo code in lecture slide.

```
typedef struct {
    int index;
} parameters;

enum { thinking, hungry, eating } state[5]; // philospher's states
pthread_mutex_t lock;           // mutex to protect condition variables
pthread_cond_t self[5];                     // 5 condition variables
```

First, we declare a struct to be able to pass in the number of each philosopher in the pthread_create(), and also, declare states and condition variable for each philosopher. Meanwhile, a mutex lock is also declared to protect each condition variable. The details will be addressed later.

```
void init() {
    // Init seed
    srand((unsigned)time(0));

    // Init mutex lock
    pthread_mutex_init(&lock, NULL);

    // Init states to thinking and each condition variable
  for (int i = 0; i < THREAD_COUNT; i++ )
  {
    state[i] = thinking;

    pthread_cond_init (&self[i], NULL);
    }
}
```

Second, initialize everything we need, which includes the time seed, the mutex with the help of pthread_mutex_init(), the state of each philosopher, and the condition variable for each philosopher with the help of pthread_cond_init().

```
void pickup_forks(int i) {

    state[i] = hungry;

    test(i);

    /* Fail to pick up forks */
    pthread_mutex_lock(&lock);

    while(state[i] != eating) {

        printf("Philosopher %d can’t pick up forks and start waiting.\n", i);

        pthread_cond_wait(&self[i], &lock);

    }

    pthread_mutex_unlock(&lock);

}
```

Then, we start implementing functions that are going to be used by each philosopher.

Let's first begin with pickup_forks(int i).

Whenever a philosopher picks up a fork, his/her state will be altered into "hungry", and soon after, he or she will start testing whether he or she could eat or not by calling function test(). Next, he or she will enter the mutex to see whether she or he has to wait.

The simplest way to understand why condition variables need a mutex is that it prevents an early wakeup message (signal or broadcast functions) from being 'lost.' Imagine the following sequence of events (time runs down the page) where the condition is satisfied just before pthread_cond_wait is called. In this example, the wake-up signal is lost!

Thread 1	Thread 2
`while (answer < 42) {`	
	`answer++`
	`p_cond_signal(cv)`
`p_cond_wait(cv, m)`	

If both threads had locked a mutex, the signal can not be sent until after pthread_cond_wait(cv, m) is called (which then internally unlocks the mutex).

Note that the call `pthread_cond_wait` performs three actions:

1. unlock the mutex
2. waits (sleeps until `pthread_cond_signal` is called on the same condition variable). It does 1 and 2 atomically.
3. Before returning, locks the mutex

Another thing to notice is why are we using a loop instead of an if-statement here.

The problem here is that the thread must release the mutex before waiting(`pthread_cond_wait`), potentially allowing another thread to 'steal' whatever that thread was waiting for. Unless it is guaranteed that only one thread can wait on that condition, it is incorrect to assume that the condition is valid when a thread wakes up. On the other hand, the condition might be fluctuating, as a result, it might better to use a loop to keep on checking the condition before we actually lock the mutex. However, in our case, it doesn't matter which one we're using since each thread maintains its own condition variable. Still, we'll use a loop here to make sure the logic is correct.

```
void test(int i) {

    pthread_mutex_lock(&lock);

    if( (state[(i+4)%5] != eating) &&
        (state[i] == hungry) &&
        (state[(i+1)%5] != eating) ) {

            state[i] = eating;

            pthread_cond_signal(&self[i]);

    }

    pthread_mutex_unlock(&lock);
}
```

This function is used to test whether one is legal to eat. Much like the reason carried out above, we need the mutex to make sure the modification is safe. Yet, we don't need a loop here cause pthread_cond_signal will not unlock the mutex.

A thread that modifies the shared data(Changing state to "eating") can invoke the pthread_cond_signal() function, thereby signaling one thread waiting on the condition variable.

It is important to note that the call to pthread_cond_signal() does not release the mutex lock. It is the subsequent call to pthread_mutex_unlock() that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to pthread_cond_signal().

```
void return_forks(int i) {

    state[i] = thinking;

    test((i+4)%5);

    test((i+1)%5);
}
```

This one is rather straight forward as the philosopher returns his/her forks by turning oneself back into thinking and start testing two of his/her neighbors to proceed the progress.

```
void *philosopher(void *param) {

    parameters *p = (parameters *)param;

    int index = p->index;

    int random = (int)(rand()%3 + 1);

    printf("Philosopher %d is now THINKING for %d seconds.\n", index, random);

    sleep(random);

    printf("Philosopher %d is now HUNGRY and trying to pick up forks.\n", index);

    pickup_forks(index);

    random = (int)(rand()%3 + 1);

    printf("Philosopher %d is now EATING.\n", index);

    sleep(random);

    printf("Philosopher %d returns forks and then starts TESTING %d and %d.\n", index, (index+4)%5, (index+1)%5);

    return_forks(index);

    //return NULL;

    pthread_exit(0);
}
```

Up until now, we're talking about the actions which the philosopher takes, and now it's time to focus on how the philosophers perform these actions.

As soon as a philosopher thread been created, he/she will sleep for a random period between one and three seconds and then pick up the forks, if he/she is able to eat, he/she will eat for another random period between one and three seconds. Otherwise, he/she will have to wait and states "Philosopher %d can’t pick up forks and start waiting." until the thread is signaled. After a philosopher

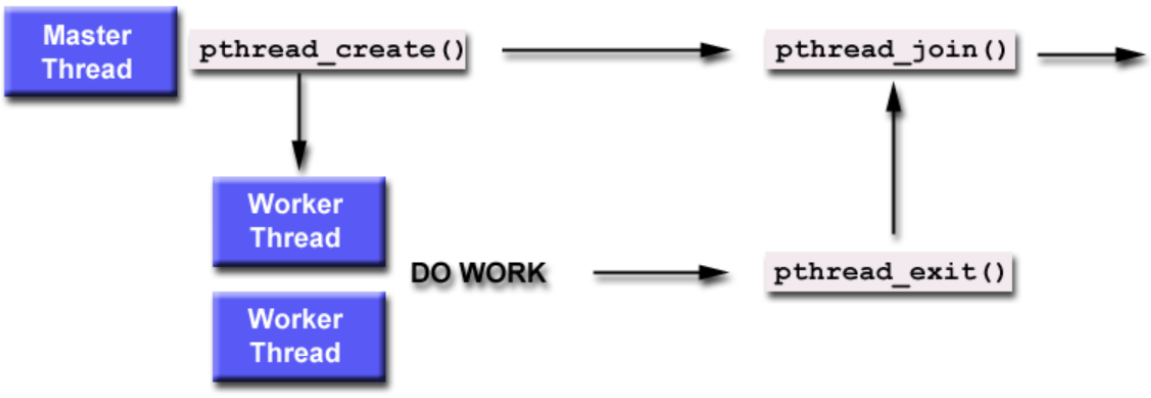
finishes eating, he or she will return the forks and exits the thread with either return NULL or pthread_exit(0).

```
int main() {

    // Initialize philosophers' state & condition variable & mutex lock
    init();

    // Declare threads
    pthread_t threads[THREAD_COUNT];


    // Create threads in order
    for(int i = 0; i < THREAD_COUNT; i++) {

        parameters *param = malloc (sizeof(parameters*));

        param->index = i;

        pthread_create(&threads[i], 0, philosopher, param);
    }

    // Join threads in order
    for (int i = 0; i < THREAD_COUNT; i++) {

        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Finally, we will walk through the whole process of the Dining Philosopher here.

We'll first call init() to initialize all the elements we need, and then create threads in order (0, 1, 2, 3, 4) by call pthread_create(&threads[i], 0, philosopher, param), where the philosopher will be activated and the param will pass in the corresponding number the philosopher need. However, in modern systems, these 5 threads run in parallel so we can't guarantee that threads will run in order 0, 1, 2, 3, 4. It is up to the host OS to decide a certain order.

Lastly, we'll join the threads in order by calling pthread_join(threads[i], NULL) as pthread_join() will wait till the threads to join and then proceed.



- Part 2. Code

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define THREAD_COUNT 5
```

```
void pickup_forks(int i);
void return_forks(int i);
void test(int i);
void init();
void *philosopher(void *param);

typedef struct {
    int index;
} parameters;

enum { thinking, hungry, eating } state[5]; // philospher's states
pthread_mutex_t lock;           // mutex to protect condition variables
pthread_cond_t self[5];                     // 5 condition variables

int main() {

    // Initialize philosophers' state & condition variable & mutex lock
    init();

    // Declare threads
    pthread_t threads[THREAD_COUNT];


    // Create threads in order
    for(int i = 0; i < THREAD_COUNT; i++) {

        parameters *param = malloc (sizeof(parameters*));

        param->index = i;

        pthread_create(&threads[i], 0, philosopher, param);
    }

    // Join threads in order
    for (int i = 0; i < THREAD_COUNT; i++) {

        pthread_join(threads[i], NULL);
    }

    return 0;
}

void init() {
    // Init seed
    srand((unsigned)time(0));

    // Init mutex lock
    pthread_mutex_init(&lock, NULL);

    // Init states to thinking and each condition variable
  for (int i = 0; i < THREAD_COUNT; i++ )
  {
    state[i] = thinking;

    pthread_cond_init (&self[i], NULL);
    }
}

void pickup_forks(int i) {

    state[i] = hungry;

    test(i);

    /* Fail to pick up forks */
    pthread_mutex_lock(&lock);

    while(state[i] != eating) {

        printf("Philosopher %d can’t pick up forks and start waiting.\n", i);

        pthread_cond_wait(&self[i], &lock);

    }

    pthread_mutex_unlock(&lock);
```

```
}

void return_forks(int i) {

    state[i] = thinking;

    test((i+4)%5);

    test((i+1)%5);
}

void test(int i) {

    pthread_mutex_lock(&lock);

    if( (state[(i+4)%5] != eating) &&
        (state[i] == hungry) &&
        (state[(i+1)%5] != eating) ) {

            state[i] = eating;

            pthread_cond_signal(&self[i]);

    }

    pthread_mutex_unlock(&lock);
}

void *philosopher(void *param) {

    parameters *p = (parameters *)param;

    int index = p->index;

    int random = (int)(rand()%3 + 1);

    printf("Philosopher %d is now THINKING for %d seconds.\n", index, random);

    sleep(random);

    printf("Philosopher %d is now HUNGRY and trying to pick up forks.\n", index);

    pickup_forks(index);

    random = (int)(rand()%3 + 1);

    printf("Philosopher %d is now EATING.\n", index);

    sleep(random);

    printf("Philosopher %d returns forks and then starts TESTING %d and %d.\n", index, (index+4)%5, (index+1)%5);

    return_forks(index);

    //return NULL;

    pthread_exit(0);
}
```

- Part 3. Results

```
2/Hw4_1073007S$ ./hw4.o
Philosopher 4 is now THINKING for 1 seconds.
Philosopher 3 is now THINKING for 1 seconds.
Philosopher 2 is now THINKING for 3 seconds.
Philosopher 1 is now THINKING for 3 seconds.
Philosopher 0 is now THINKING for 1 seconds.
Philosopher 4 is now HUNGRY and trying to pick up forks.
Philosopher 4 is now EATING.
Philosopher 3 is now HUNGRY and trying to pick up forks.
Philosopher 3 can't pick up forks and start waiting.
Philosopher 0 is now HUNGRY and trying to pick up forks.
Philosopher 0 can't pick up forks and start waiting.
Philosopher 4 returns forks and then starts TESTING 3 and 0.
Philosopher 3 is now EATING.
Philosopher 0 is now EATING.
Philosopher 2 is now HUNGRY and trying to pick up forks.
Philosopher 2 can't pick up forks and start waiting.
Philosopher 1 is now HUNGRY and trying to pick up forks.
Philosopher 1 can't pick up forks and start waiting.
Philosopher 0 returns forks and then starts TESTING 4 and 1.
Philosopher 1 is now EATING.
Philosopher 3 returns forks and then starts TESTING 2 and 4.
Philosopher 1 returns forks and then starts TESTING 0 and 2.
Philosopher 2 is now EATING.
Philosopher 2 returns forks and then starts TESTING 1 and 3.
justinyeh1995@justinyeh1995-VirtualBox:~/Desktop/Operatin_System-NTHU_108
2/Hw4_1073007S$
```

To make the result more readable, we added the duration time of eating, and here's the result.

```
justinyeh1995@justinyeh1995-VirtualBox:~/Desktop/Operatin_System-NTHU_108
2/Hw4_1073007S$ ./hw4.o
Philosopher 4 is now THINKING for 3 seconds.
Philosopher 3 is now THINKING for 3 seconds.
Philosopher 2 is now THINKING for 2 seconds.
Philosopher 1 is now THINKING for 3 seconds.
Philosopher 0 is now THINKING for 1 seconds.
Philosopher 0 is now HUNGRY and trying to pick up forks.
Philosopher 0 is now EATING for 3 seconds.
Philosopher 2 is now HUNGRY and trying to pick up forks.
Philosopher 2 is now EATING for 3 seconds.
Philosopher 4 is now HUNGRY and trying to pick up forks.
Philosopher 4 can't pick up forks and start waiting.
Philosopher 3 is now HUNGRY and trying to pick up forks.
Philosopher 3 can't pick up forks and start waiting.
Philosopher 1 is now HUNGRY and trying to pick up forks.
Philosopher 1 can't pick up forks and start waiting.
Philosopher 0 returns forks and then starts TESTING 4 and 1.
Philosopher 4 is now EATING for 3 seconds.
Philosopher 2 returns forks and then starts TESTING 1 and 3.
Philosopher 1 is now EATING for 2 seconds.
Philosopher 1 returns forks and then starts TESTING 0 and 2.
Philosopher 4 returns forks and then starts TESTING 3 and 0.
Philosopher 3 is now EATING for 2 seconds.
Philosopher 3 returns forks and then starts TESTING 2 and 4.
justinyeh1995@justinyeh1995-VirtualBox:~/Desktop/Operatin_System-NTHU_108
2/Hw4_1073007S$
```