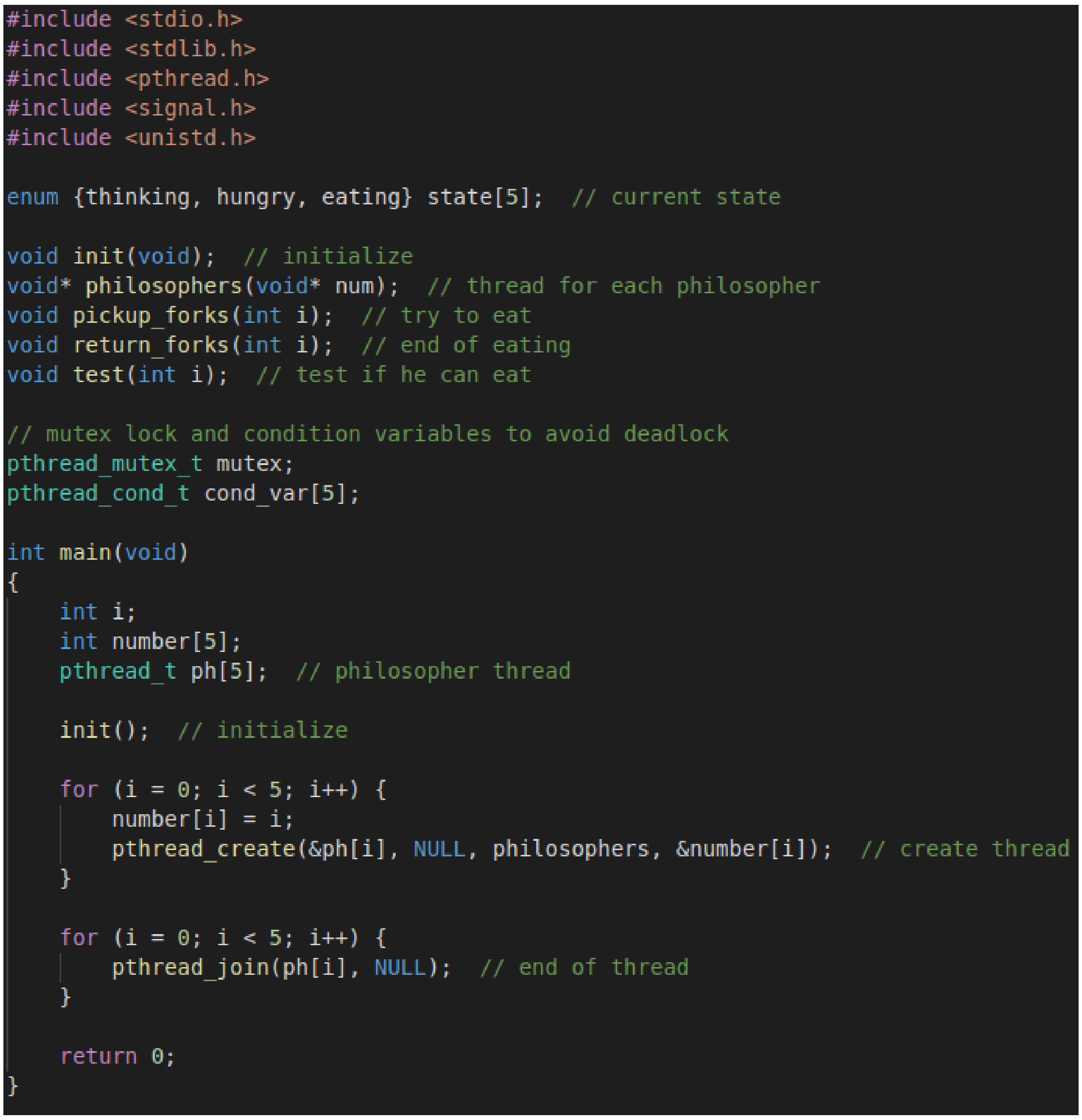
Operating Systems hw4 105061212 王家駿

Code and description:

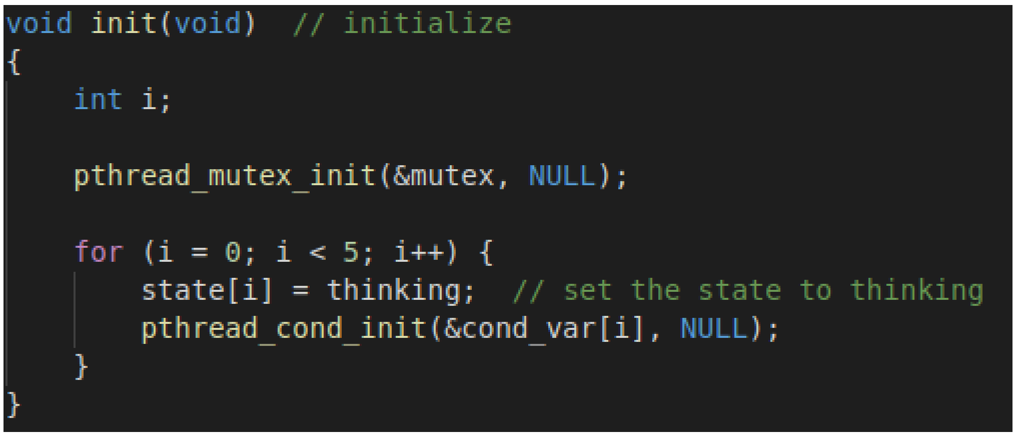
1. Main function and global variables:



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <unistd.h>

enum {thinking, hungry, eating} state[5];  // current state

void init(void);  // initialize
void* philosophers(void* num);  // thread for each philosopher
void pickup_forks(int i);  // try to eat
void return_forks(int i);  // end of eating
void test(int i);  // test if he can eat

// mutex lock and condition variables to avoid deadlock
pthread_mutex_t mutex;
pthread_cond_t cond_var[5];

int main(void)
{
    int i;
    int number[5];
    pthread_t ph[5];  // philosopher thread

    init();  // initialize

    for (i = 0; i < 5; i++) {
        number[i] = i;
        pthread_create(&ph[i], NULL, philosophers, &number[i]);  // create thread
    }

    for (i = 0; i < 5; i++) {
        pthread_join(ph[i], NULL);  // end of thread
    }

    return 0;
}
```

In the main function, we create five threads for each philosopher by pthread_create() function, and terminates with the pthread_join() function. In addition, we use a mutex lock and five condition variables to prevent the threads from deadlock.

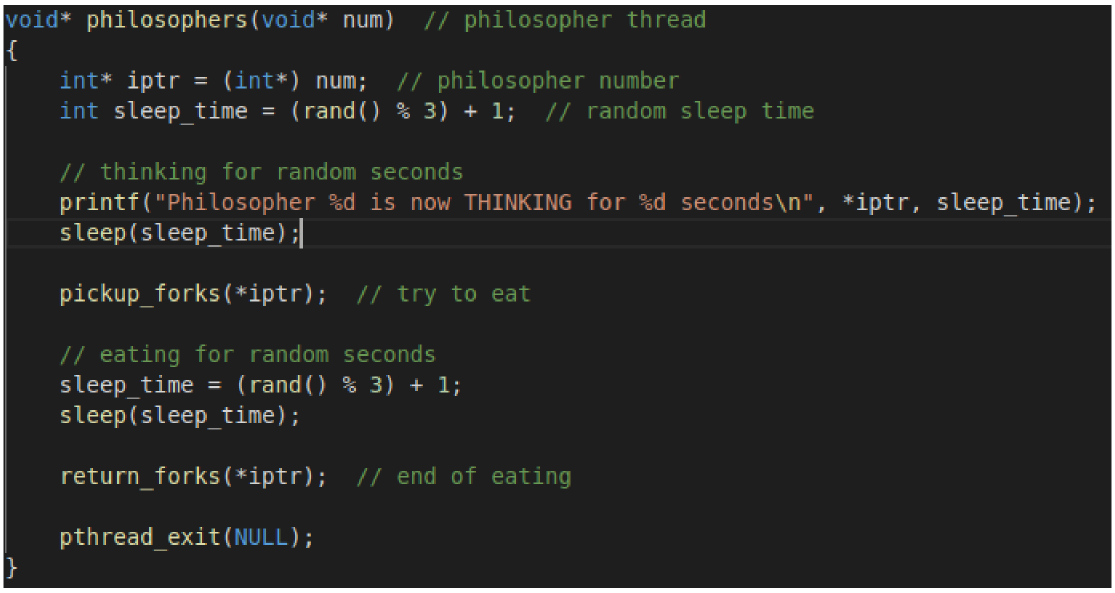
2. Init function:



```c
void init(void)  // initialize
{
    int i;

    pthread_mutex_init(&mutex, NULL);

    for (i = 0; i < 5; i++) {
        state[i] = thinking;  // set the state to thinking
        pthread_cond_init(&cond_var[i], NULL);
    }
}
```

In the init function, we initialize the mutex lock and condition variables, and set all the state of philosophers to thinking.

3. Philosopher thread

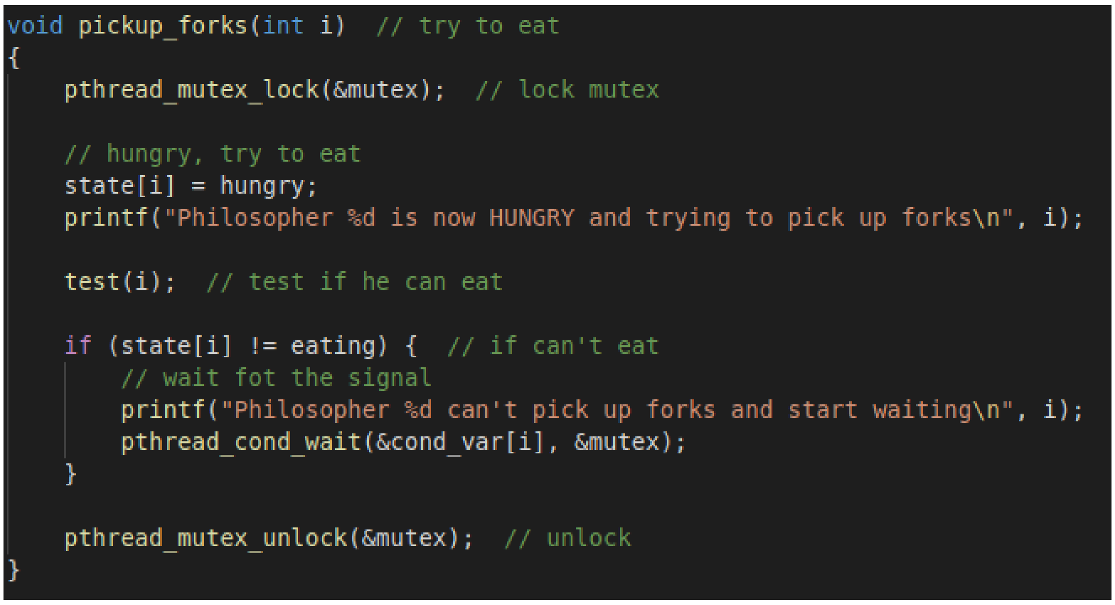


```c
void* philosophers(void* num)  // philosopher thread
{
    int* iptr = (int*) num;  // philosopher number
    int sleep_time = (rand() % 3) + 1;  // random sleep time

    // thinking for random seconds
    printf("Philosopher %d is now THINKING for %d seconds\n", *iptr, sleep_time);
    sleep(sleep_time);

    pickup_forks(*iptr);  // try to eat

    // eating for random seconds
    sleep_time = (rand() % 3) + 1;
    sleep(sleep_time);

    return_forks(*iptr);  // end of eating

    pthread_exit(NULL);
}
```

The thread contains the behavior of a philosopher, waiting for a few seconds to eat, picking up the forks and starting to eat, eating for a few seconds, and return the forks when end of eating.

At first, the philosopher is at thinking state. After a few seconds, he tries to eat by the pickup_forks() function, and keep eating for a few seconds. The time is generated by rand() function. And at the end of eating, he resumes to the thinking state by the return_forks() function.

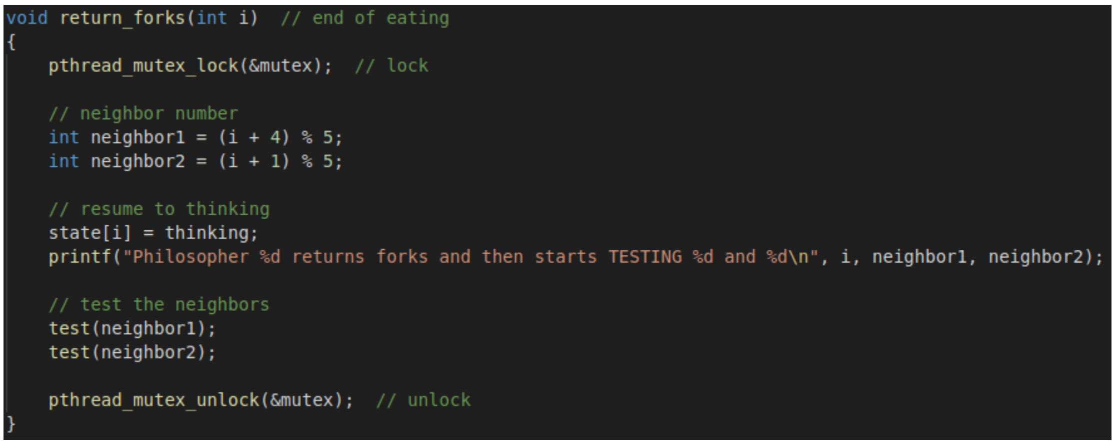
4. Pickup_fork function



```c
void pickup_forks(int i)  // try to eat
{
    pthread_mutex_lock(&mutex);  // lock mutex

    // hungry, try to eat
    state[i] = hungry;
    printf("Philosopher %d is now HUNGRY and trying to pick up forks\n", i);

    test(i);  // test if he can eat

    if (state[i] != eating) {  // if can't eat
        // wait fot the signal
        printf("Philosopher %d can't pick up forks and start waiting\n", i);
        pthread_cond_wait(&cond_var[i], &mutex);
    }

    pthread_mutex_unlock(&mutex);  // unlock
}
```

When a philosopher is going to eat, the state of him is change to hungry, and we use the test() function to check if he could eat successfully. If he couldn't eat due to the constraints, we make the thread wait for the signal to continue.

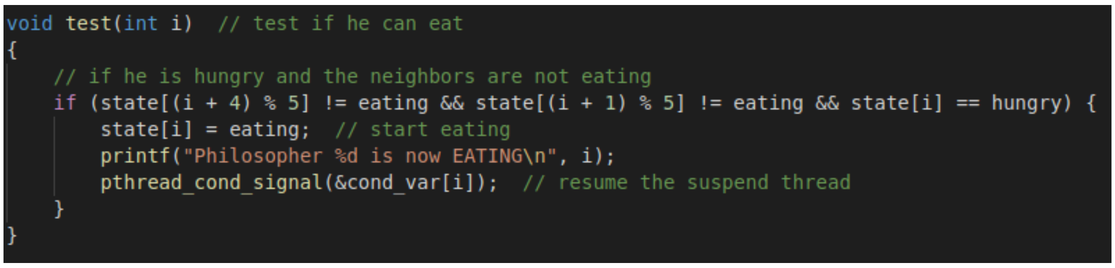
5. Return_fork function



```c
void return_forks(int i)  // end of eating
{
    pthread_mutex_lock(&mutex);  // lock

    // neighbor number
    int neighbor1 = (i + 4) % 5;
    int neighbor2 = (i + 1) % 5;

    // resume to thinking
    state[i] = thinking;
    printf("Philosopher %d returns forks and then starts TESTING %d and %d\n", i, neighbor1, neighbor2);

    // test the neighbors
    test(neighbor1);
    test(neighbor2);

    pthread_mutex_unlock(&mutex);  // unlock
}
```

When a philosopher ends his eating, we resume his state to thinking. Since we also have to return the forks he got, that is, the neighbors of him could have the chance to eat, we test the two neighbors of this philosopher.

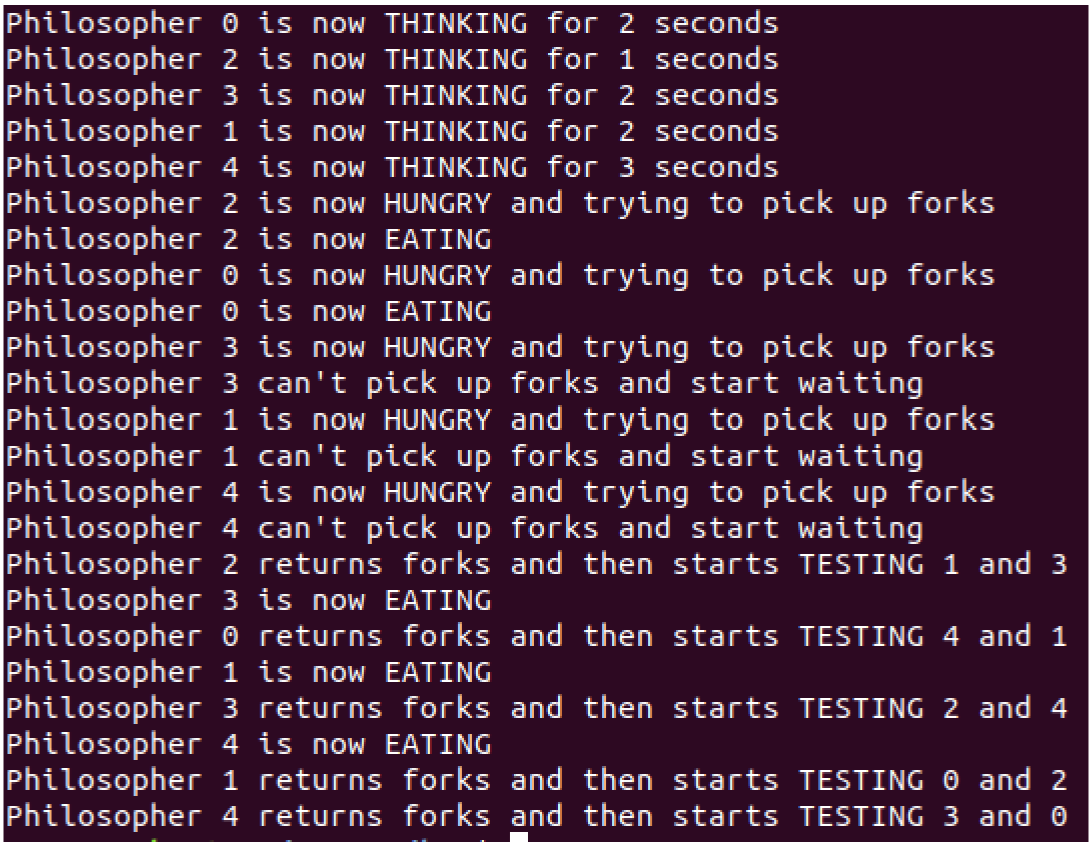
6. Test function



```
void test(int i)  // test if he can eat
{
    // if he is hungry and the neighbors are not eating
    if (state[(i + 4) % 5] != eating && state[(i + 1) % 5] != eating && state[i] == hungry) {
        state[i] = eating;  // start eating
        printf("Philosopher %d is now EATING\n", i);
        pthread_cond_signal(&cond_var[i]);  // resume the suspend thread
    }
}
```

In the test function, we are requested to check if the philosopher could eat successfully. So, we check the two neighbors if they're eating and if this philosopher is hungry (ready to eat). If the check success, we change the state to eating, and signal the condition variable to resume the process which might be suspended.

Execution results:



```
Philosopher 0 is now THINKING for 2 seconds
Philosopher 2 is now THINKING for 1 seconds
Philosopher 3 is now THINKING for 2 seconds
Philosopher 1 is now THINKING for 2 seconds
Philosopher 4 is now THINKING for 3 seconds
Philosopher 2 is now HUNGRY and trying to pick up forks
Philosopher 2 is now EATING
Philosopher 0 is now HUNGRY and trying to pick up forks
Philosopher 0 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 1 is now HUNGRY and trying to pick up forks
Philosopher 1 can't pick up forks and start waiting
Philosopher 4 is now HUNGRY and trying to pick up forks
Philosopher 4 can't pick up forks and start waiting
Philosopher 2 returns forks and then starts TESTING 1 and 3
Philosopher 3 is now EATING
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 4 is now EATING
Philosopher 1 returns forks and then starts TESTING 0 and 2
Philosopher 4 returns forks and then starts TESTING 3 and 0
```