

School of Computer and Information Science

《Operating-System-Concepts(10th)》

Project Report

Student information:

Name: 张乐之 ID: 222022321102120
Major/Grade: 2022 计科中外 04 Time: 2024-11-08

Experiment information:

Topic:

Programming Projects in Chapter 7

Requirements:

1. Master the basic concepts, design ideas and operation status of the process. Understand the basic principles and ideas of process synchronization and mutual exclusion, and identify the key links, steps and constraints of engineering problems in this field. Use the basic means of realizing process synchronization and mutual exclusion to analyze and solve the dining problem of philosophers in the field of process synchronization and mutual exclusion, and learn the basic means to solve the deadlock problem.

Procedure:

1. Define the algorithm and figure out how to avoid deadlocks.
2. Define the behavior of each thread.
3. Code with the header pthread.h and unistd.h to create thread under one process.
4. Compile and link the codes so that it can run on Linux system.
5. Run the program and verify the result.

Results:

(Code and Figures)

To solve the dining-philosophers problem using Pthreads, we can use mutex locks and condition variables to manage fork resources and control the state of each philosopher (thinking, hungry, or eating). Here's a structured breakdown for the solution:

1. Define the States

Each philosopher can be in one of three states:

- **Thinking:** Not trying to eat.
- **Hungry:** Waiting to acquire forks.
- **Eating:** Currently eating and holding both forks.

2. Initialize Mutexes and Condition Variables

We need a mutex to ensure only one philosopher is attempting to change state at any given time and condition variables for each philosopher to signal when they can start eating.

3. Philosopher Functions

Each philosopher tries to pick up forks, eat, and then put down forks.

Pickup Forks

In the `pickup_forks()` function, a philosopher changes to HUNGRY, checks if they can eat, and waits if not.

Return Forks

In `return_forks()`, a philosopher returns the forks and checks if their neighbors can eat.

Test Function

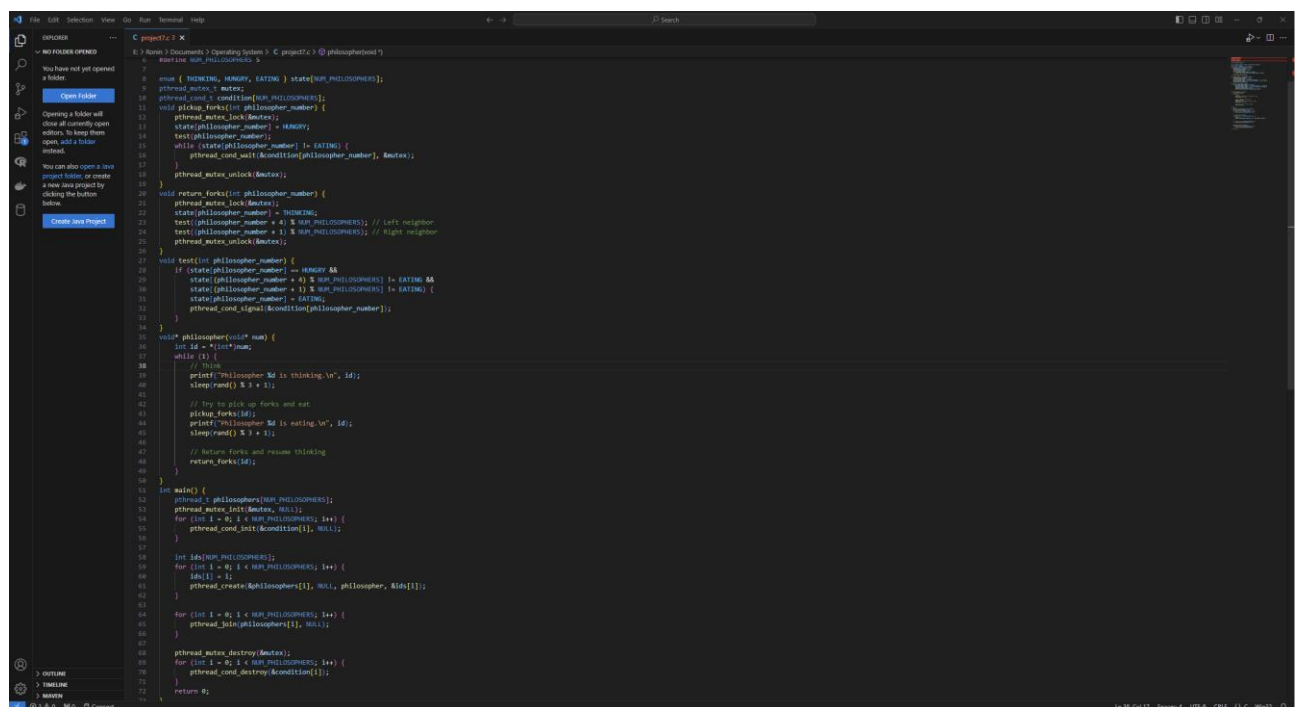
The `test()` function allows a philosopher to eat if they're hungry and both neighbors aren't eating.

4. Philosopher Thread Function

Each philosopher thread alternates between thinking and eating.

5. Main Function

Initialize the mutex, condition variables, and create philosopher threads.



```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #define NUM_PHILOSOPHERS 5
7
8 enum { THINKING, HUNGRY, EATING } state[NUM_PHILOSOPHERS];
9 pthread_mutex_t mutex;
10 pthread_cond_t condition[NUM_PHILOSOPHERS];
11
12 void pickup_forks(int philosopher_number) {
13     pthread_mutex_lock(&mutex);
14     state[philosopher_number] = HUNGRY;
15     test(philosopher_number);
16     while (state[philosopher_number] != EATING) {
17         pthread_cond_wait(&condition[philosopher_number], &mutex);
18     }
19     pthread_mutex_unlock(&mutex);
20 }
21
22 void return_forks(int philosopher_number) {
23     pthread_mutex_lock(&mutex);
24     state[philosopher_number] = THINKING;
25     test(philosopher_number + 4 % NUM_PHILOSOPHERS); // left neighbor
26     test(philosopher_number + 1 % NUM_PHILOSOPHERS); // right neighbor
27     pthread_mutex_unlock(&mutex);
28 }
29
30 void test(int philosopher_number) {
31     if (state[philosopher_number] == HUNGRY &&
32         state[philosopher_number + 4 % NUM_PHILOSOPHERS] != EATING &&
33         state[philosopher_number + 1 % NUM_PHILOSOPHERS] != EATING) {
34         state[philosopher_number] = EATING;
35         pthread_cond_signal(&condition[philosopher_number]);
36     }
37 }
38
39 void* philosopher(void* num) {
40     int id = *(int*)num;
41     while (1) {
42         // Think
43         printf("Philosopher %d is thinking.\n", id);
44         sleep(rand() % 5 + 1);
45
46         // Try to pick up forks and eat
47         pickup_forks(id);
48         printf("Philosopher %d is eating.\n", id);
49         sleep(rand() % 5 + 1);
50
51         // Return forks and resume thinking
52         return_forks(id);
53     }
54 }
55
56 int main() {
57     pthread_t philosophers[NUM_PHILOSOPHERS];
58     pthread_mutex_init(&mutex, NULL);
59     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
60         pthread_cond_init(&condition[i], NULL);
61     }
62
63     int id[NUM_PHILOSOPHERS];
64     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
65         id[i] = i;
66         pthread_create(&philosophers[i], NULL, philosopher, &id[i]);
67     }
68
69     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
70         pthread_join(philosophers[i], NULL);
71     }
72
73     pthread_mutex_destroy(&mutex);
74     for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
75         pthread_cond_destroy(&condition[i]);
76     }
77
78     return 0;
79 }
```

```
#include <pthread.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#define NUM_PHILOSOPHERS 5
```

```
enum { THINKING, HUNGRY, EATING } state[NUM_PHILOSOPHERS];
```

```
pthread_mutex_t mutex;
```

```
pthread_cond_t condition[NUM_PHILOSOPHERS];
```

```
void pickup_forks(int philosopher_number) {
```

```
    pthread_mutex_lock(&mutex);
```

```
    state[philosopher_number] = HUNGRY;
```

```
    test(philosopher_number);
```

```
    while (state[philosopher_number] != EATING) {
```

```

        pthread_cond_wait(&condition[philosopher_number], &mutex);
    }
    pthread_mutex_unlock(&mutex);
}

void return_forks(int philosopher_number) {
    pthread_mutex_lock(&mutex);
    state[philosopher_number] = THINKING;
    test((philosopher_number + 4) % NUM_PHILOSOPHERS); // Left neighbor
    test((philosopher_number + 1) % NUM_PHILOSOPHERS); // Right neighbor
    pthread_mutex_unlock(&mutex);
}

void test(int philosopher_number) {
    if (state[philosopher_number] == HUNGRY &&
        state[(philosopher_number + 4) % NUM_PHILOSOPHERS] != EATING &&
        state[(philosopher_number + 1) % NUM_PHILOSOPHERS] != EATING) {
        state[philosopher_number] = EATING;
        pthread_cond_signal(&condition[philosopher_number]);
    }
}

void* philosopher(void* num) {
    int id = *(int*)num;
    while (1) {
        // Think
        printf("Philosopher %d is thinking.\n", id);
        sleep(rand() % 3 + 1);

        // Try to pick up forks and eat
        pickup_forks(id);
        printf("Philosopher %d is eating.\n", id);
        sleep(rand() % 3 + 1);

        // Return forks and resume thinking
        return_forks(id);
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    pthread_mutex_init(&mutex, NULL);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_init(&condition[i], NULL);
    }

    int ids[NUM_PHILOSOPHERS];
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }
}

```

```

    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_cond_destroy(&condition[i]);
    }

    return 0;
}

```

As is seen in the graph, we can find that the program runs smoothly without triggering a deadlock.

