MP4 Dining-philosopher

111062502

涂博允

Fork.cpp

```cpp
#include "fork.hpp"
#include <pthread.h>
bool isUsed;
Fork::Fork() {
    // TODO: implement fork constructor (value, mutex, cond)
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    isUsed = false;
}

void Fork::wait() {
    // TODO: implement semaphore wait
    // 等待互斥鎖
    pthread_mutex_lock(&mutex);
    while (isUsed) {
        // 如果叉子已被使用，等待條件變數
        pthread_cond_wait(&cond, &mutex);

    }
    // 將叉子標記為使用中
    isUsed = true;
    // 釋放互斥鎖
    pthread_mutex_unlock(&mutex);

}

void Fork::signal() {

    // TODO: implement semaphore signal
    // 等待互斥鎖
    pthread_mutex_lock(&mutex);
    // 將叉子標記為未使用
    isUsed = false;
    // 發送條件變數信號，通知等待的哲學家
    pthread_cond_signal(&cond);
    // 釋放互斥鎖
    pthread_mutex_unlock(&mutex);
}

Fork::~Fork() {
    // TODO: implement fork destructor (mutex, cond)
     // 釋放互斥鎖和條件變數
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
}
```

為了避免一把叉子同時被兩個哲學家使用，因此我額外使用的參數isUsed來控制。
如果叉子還尚未被使用時，則為false，之後第一位來使用叉子的人，若取得叉子後則將isUsed改為true，因此後面也想使用這把叉子的哲學家就會因為isUsed為true而被卡在while loop內等待。

當持有該把叉子的人使用結束後，會先把isUsed改成false並signal通知還在wait的哲學家

Table.cpp

```cpp
#include "table.hpp"
#include "stdio.h"
#include <pthread.h>
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    // 初始化信號量的值為 n
    value = n;

    // 初始化互斥鎖和條件變數
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
}

void Table::wait() {
    // TODO: implement semaphore wait
    // 鎖住互斥鎖，確保同一時間只有一個執行緒能夠進入這段程式碼
    pthread_mutex_lock(&mutex);

    // 如果信號量的值 <= 0，代表已經有 4 個哲學家正在用餐，需要等待
    while (value <= 0) {
        // 等待條件變數的激發，並且釋放互斥鎖
        pthread_cond_wait(&cond, &mutex);
    }

    // 減少信號量的值，代表有一個哲學家正在用餐
    value--;

    // 釋放互斥鎖
    pthread_mutex_unlock(&mutex);
}

void Table::signal() {
    // TODO: implement semaphore signal
    // 鎖住互斥鎖，確保同一時間只有一個執行緒能夠進入這段程式碼
    pthread_mutex_lock(&mutex);

    // 增加信號量的值，代表有一個哲學家用餐完畢
    value++;

    // 激發條件變數，通知其他等待中的執行緒有空位可以用餐
    pthread_cond_signal(&cond);

    // 釋放互斥鎖
    pthread_mutex_unlock(&mutex);
}

Table::~Table() {
    // TODO: implement table destructor (mutex, cond)
    // 刪除互斥鎖和條件變數
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
}
```

題目要求table最多只能容納4人，因此假設n=4，table每進一個人(wait())，就會判斷value是否>=1，若滿足則可以成功入桌並把value-1。

若有人要離桌，則把value+1即可離桌，代表多一個位置。

Philosopher.cpp

```
2
3  void Philosopher::start() {
4      // TODO: start a philosopher thread
5
6      pthread_create(&tid[i], NULL,run, this);
7  }
8
9  int Philosopher::join() {
0      // TODO: join a philosopher thread
1
2
3      pthread_join(tid[i],NULL);
4
5      return 0;
6  }
7
8  int Philosopher::cancel() {
9      // TODO: cancel a philosopher thread
0      int i ;
1      cancelled = true;
2
3      pthread_cancel(tid[i]);
4
5      return 0;
6  }
```

pthread_create 可以用來建立新的執行緒，並以函數指標指定子執行緒所要執行的函數，子執行緒在建立之後，就會以平行的方式執行，在子執行緒的執行期間，主執行緒還是可以正常執行自己的工作，最後主執行緒再以 pthread_join 等待子執行緒執行結束，處理後續收尾的動作。

```
void Philosopher::pickup(int id) {
    // TODO: implement the pickup interface, the philosopher needs to pick up the left fork first, then the right fork

    int left,right;
    if(id == 4){
      left = id;
      right = 0;
    }
    else{
      left = id;
      right = id+1;
    }
    fk[left].wait(); //拿到左手的叉子。
    beUsed[left]=1;
    printf("philosopher %d fetches fork %d\n",id,left);

    while(beUsed[right==1]){//右邊已被拿走 拿右手的叉子失敗
      fk[left].signal();//右手叉子被拿走，放下左手的叉子。
      beUsed[left]=0;
      continue;
    }
    printf("philosopher %d fetches fork %d\n",id,right);

}

void Philosopher::putdown(int id) {
    // TODO: implement the putdown interface, the philosopher needs to put down the left fork first, then the right fork
    int left,right;
    if(id == 4){
      left = 0;
      right = id;
    }
    else{
      left = id;
      right = id+1;
    }
    fk[right].signal(); //放下右手的叉子。
    beUsed[right]=0;
    printf("philosopher %d release fork %d\n",id,right);

    fk[left].signal(); //放下左手的叉子。
    beUsed[left]=0;
    printf("philosopher %d release fork %d\n",id,left);

}


void Philosopher::enter() {
    // TODO: implement the enter interface, the philosopher needs to join the table first
    table->wait();
}

void Philosopher::leave() {
    // TODO: implement the leave interface, the philosopher needs to let the table know that he has left
    table->signal();
}

void* Philosopher::run(void* arg) {
    // TODO: complete the philosopher thread routine.
    Philosopher* p = static_cast<Philosopher*>(arg);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    //給好的
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    while (!p->cancelled) {
        p->enter();
        p->think();

        //pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        p->pickup(p->id);
        p->eat();
        p->putdown(p->id);

        p->leave();
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    }


    return NULL;
}
```

Pickup的部分由於id=0的哲學家會拿到id=0,1的叉子，以此類推，而id=4的哲學家會拿到4,0叉子。
為了防止deadlock，我額外使用了beUsed來判斷，如果已經取得左叉子後，發現右叉子已經被拿走，則此時該哲學家會放下左叉子，避免hold and wait。
那putdown的部分就單純的放下右叉子以及左叉子，並且把beUsed設為0，代表沒有被使用了。

底下則是table跟philosopher的interface以及philosophe的thread routine

1. Why does the function pthread_cond_wait() need a mutex variable as second parameter, while function pthread_cond_signal() does not?

pthread_cond_wait() :
用於等待條件變數被其他線程訊號通知。它需要一個互斥變數作為第二個參數，是因為在等待條件變數之前，呼叫的線程必須先獲取互斥鎖。這是為了確保對條件變數所代表的共享資源的獨佔訪問。通過在等待之前獲取互斥鎖，線程避免了多個線程在沒有同步的情況下同時等待條件變數可能產生的競爭條件或衝突。

pthread_cond_signal() :
用於通知一個等待的線程條件發生。它不需要互斥鎖，因為它假設在調用此函式時，呼叫的線程已經持有互斥鎖。通常在修改由互斥鎖保護的共享資源之後，才會發出條件信號。因此，由於互斥鎖已經持有，不需要將其作為單獨的參數傳遞給
pthread_cond_signal()。

2. Which part of the implementation ensures the fork is only used by one philosopher at a time ? How ?

```cpp
#include "fork.hpp"
#include <pthread.h>
bool isUsed;
Fork::Fork() {
    // TODO: implement fork constructor (value, mutex, cond)
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    isUsed = false;
}

void Fork::wait() {
    // TODO: implement semaphore wait
    // 等待互斥鎖
    pthread_mutex_lock(&mutex);
    while (isUsed) {
        // 如果叉子已被使用，等待條件變數
        pthread_cond_wait(&cond, &mutex);

    }
    // 將叉子標記為使用中
    isUsed = true;
    // 釋放互斥鎖
    pthread_mutex_unlock(&mutex);

}

void Fork::signal() {

    // TODO: implement semaphore signal
    // 等待互斥鎖
    pthread_mutex_lock(&mutex);
    // 將叉子標記為未使用
    isUsed = false;
    // 發送條件變數信號，通知等待的哲學家
    pthread_cond_signal(&cond);
    // 釋放互斥鎖
    pthread_mutex_unlock(&mutex);
}

Fork::~Fork() {
    // TODO: implement fork destructor (mutex, cond)
     // 釋放互斥鎖和條件變數
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
}
```

為了避免一把叉子同時被兩個哲學家使用，因此我額外使用的參數isUsed來控制。
如果叉子還尚未被使用時，則為false，之後第一位來使用叉子的人，若取得叉子後則將isUsed改為true，因此後面也想使用這把叉子的哲學家就會因為isUsed為true而被卡在while loop內等待。

當持有該把叉子的人使用結束後，會先把isUsed改成false並signal通知還在wait的哲學家

3. Which part of the implementation avoids the deadlock (i.e. philosophers are all waiting for the forks to eat) happen? How does it avoid this?

```
void Philosopher::pickup(int id) {
    // TODO: implement the pickup interface, the philosopher needs to pick up the left fork first, then the right fork

    int left,right;
    if(id == 4){
      left = id;
      right = 0;
    }
    else{
      left = id;
      right = id+1;
    }
    fk[left].wait(); //拿到左手的筷子。
    beUsed[left]=1;
    printf("philosopher %d fetches fork %d\n",id,left);

    while(beUsed[right==1]){//右邊已被拿走 拿右手的筷子失敗
      fk[left].signal();//右手筷子被拿走，放下左手的筷子。
      beUsed[left]=0;
      continue;
    }
    printf("philosopher %d fetches fork %d\n",id,right);

}

void Philosopher::putdown(int id) {
    // TODO: implement the putdown interface, the philosopher needs to put down the left fork first, then the right fork
    int left,right;
    if(id == 4){
      left = 0;
      right = id;
    }
    else{
      left = id;
      right = id+1;
    }
    fk[right].signal(); //放下右手的筷子。
    beUsed[right]=0;
    printf("philosopher %d release chopstick %d\n",id,right);

    fk[left].signal(); //放下左手的筷子。
    beUsed[left]=0;
    printf("philosopher %d release chopstick %d\n",id,left);

}
```

Pickup的部分由於id=0的哲學家會拿到id=0,1的叉子，以此類推，而id=4的哲學家會拿到4,0叉子。
為了防止deadlock，我額外使用了beUsed來判斷，如果已經取得左叉子後，發現右叉子已經被拿走，則此時該哲學家會放下左叉子，避免hold and wait。
那putdown的部分就單純的放下右叉子以及左叉子，並且把beUsed設為0，代表沒有被使用了。

4. After finishing the implementation, does the program is starvation-free? Why or why not?

會有starvation，舉個例子，因為當1號哲學家要用餐時，必須等旁邊的哲學家(0號or2號)用完餐放下叉子，因此如果旁邊的哲學家(0號or2號)用餐時間非常大時，那就會導致該1號哲學家starvation。

5. What is the purpose of using pthread_setcancelstate() ?

pthread_setcancelstate() 函式用於在thread運行時動態地更改取消狀態。通常情況下，可以在thread的起始點或適當的程式碼位置調用這個函式，以根據需要設置thread的取消狀態。

使用 pthread_setcancelstate() 的目的是提供對thread取消功能的控制。在某些情況下，可能需要在thread run的期間禁用取消，以確保特定的操作或臨界區域不會被中斷。一旦臨界區域完成，可以恢復取消功能，使thread可以被取消。

Explain the pros and cons of using monitor to solve a dining-philosopher problem compared to this homework ? (You should at least analyze the performance and scalability.)

使用monitor是另一種常見的解決Dining-philosopher問題的方法，它提供了Structured and abstract的方式來管理共享資源和同步線程。

優點：

Structured and abstract ： monitor提供了一種結構化的方式來組織代碼，使得共享資源的訪問和同步操作更加清晰和易於理解。這有助於提高代碼的可讀性和可維護性。

Simplified synchronization ： monitor將資源的同步細節封裝在內部，使得thread之間的同步操作更加簡單。開發人員只需聚焦於定義monitor的方法和條件，而不必擔心互斥鎖和條件變數的操作。

Safety： monitor提供了一種安全的共享資源訪問機制，避免了race conditio和deadloc等multithread problem的發生。它確保了在任何時候只有一個thread可以訪問monitor中的方法，從而維護了資源的一致性和完整性。

缺點：

Performance：使用monitor可能引入額外的性能開銷。每個thread進入monitor時，需要獲取monitor的互斥鎖，這可能導致其他thead的等待和上下文切換。在高並發情況下，這可能影響系統的效能。

Scalability：monitor的可擴展性可能受到限制。當多個thread需要同時訪問多個不同的monitor時，可能會出現競爭和效能瓶頸的問題。

5. Any feedback you would like to let us know

由於以前很少接觸到，因此花了些許時間去瞭解pthread的功能以及用法，
以及瞭解哲學家問題的解決方法，還有比較Monitor跟使用phtread的差異性。