MP4_report

109070025 林泓錩

1 Implement

1.1 Fork

fork.hpp:

Below is the screenshot of my fork.hpp. I didn't make ant change of it. It is same as the original .hpp copy from the server.

```
#ifndef FORK_HPP
#define FORK_HPP

#include <pthread.h>

class Fork {
public:
    Fork();
    void wait();
    void signal();
    ~Fork();
private:
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int value;
};



#endif // FORK_HPP
```

fork.cpp:

In fork.cpp, the first function I implement is the Fork::Fork(), and the TODO give the hint that this function is used to implement fork constructor (value, mutex, cond). Therefore, I used the function in pthread.h, pthread_mutex_init() and pthread_cond_init(). pthread_mutex_init() is to initialize a mutex, and pthread_cond_init() is to initialize a condition variable. Furthermore, I set the value=1, which is a private variable in fork class. I set it equals to one represents that the fork is able to be used now.

```
Fork::Fork() {
    // TODO: implement fork constructor (value, mutex, cond)
    value = 1;                                   // means that this fork is able to be used
    pthread_mutex_init(&mutex, NULL);            // initialize a mutex
    pthread_cond_init(&cond, NULL);              // initialize a condition variable
}
```

Secondly, I implement the Fork::wait() function, and the TODO give the hint that this function is used to implement semaphore wait. Hence, in the first step, I use the pthread_mutex_lock() function to represent that the process enters the critical section. Then, I set a while() loop, and if the value is not equal to 1, it will keep running the loop. In the loop, I use the pthread_cond_wait() function, which will Wait for condition variable to be signaled or broadcast and MUTEX is assumed to be locked before. After the while() loop, set value equals to 0. It means that the fork is being used now. Finally, use the pthread_mutex_unlock() function to release the lock and represent that the process leaves the critical section. The variable "value" will change in Fork::wait(), however, we use mutex lock and unlock to ensure there won't occur the critical section problem.

```
void Fork::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);                 // enter critical section, lock mutex
    while(value != 1){
        pthread_cond_wait(&cond, &mutex);       // Wait for condition variable COND to be signaled or broadc
                                                // MUTEX is assumed to be locked before.
    }
    value = 0;                                  // means that this fork is not able to be used
    pthread_mutex_unlock(&mutex);               // leave critical section, release the lock
}
```

Thirdly, I implement the Fork::signal() function, and the TODO give the hint that this function is used to implement semaphore signal. Hence, in the first step, I use the pthread_mutex_lock() function to represent that the process enters the critical section. Then, set value equals to 1. It means that this fork is able to be used, someone put down it. After that, I used pthread_cond_signal() to wake up one thread waiting for condition variable, which is the pthread_cond_wait() part I mentioned in the before Fork::wait(). IN the end, use the pthread_mutex_unlock() function to release the lock and represent that the process leaves the critical section. The variable "value" will also change in Fork::signal(), however, we use mutex lock and unlock to ensure there won't occur the critical section problem.

```
void Fork::signal() {
    // TODO: implement semaphore signal
    pthread_mutex_lock(&mutex);                 // enter critical section, lock mutex
    value = 1;                                  // means that this fork is able to be used
    pthread_cond_signal(&cond);                 // Wake up one thread waiting for condition variable
    pthread_mutex_unlock(&mutex);               // leave critical section, release the lock
}
```

Last, I implement the Fork:: Fork() function, and the TODO give the hint that this function is used to implement fork destructor (mutex, cond). Thus, I used the function in pthread.h, pthread_mutex_destroy() and pthread_cond_destroy(). pthread_mutex_destroy() is to destroy a mutex, and pthread_cond_destroy() is to destroy a condition variable.

```
Fork::~Fork() {
    // TODO: implement fork destructor (mutex, cond)
    pthread_mutex_destroy(&mutex);          // destroy a mutex
    pthread_cond_destroy(&cond);            // destroy a condition variable
}
```

These are the part of my Fork implement.

1.2 Table

table.hpp:

Below is the screenshot of my table.hpp. Excpet the original .hpp content copy from the server, I also added a private variable "currentTurn" in the table class and two public functions "getCurrentTurn()" 、updateCurrentTurn() in the table class. I created these variable and functions to implement solving possibility of the starvation problem. I will explain it in the later part. In this part, I will explain how I implement table to solve the dining philosopher problem.

```
#ifndef TABLE_HPP
#define TABLE_HPP

#include <pthread.h>

class Table {
public:
    Table(int n);
    void wait();
    void signal();
    int getCurrentTurn();      // Starvation
    void updateCurrentTurn(); // Starvation
    ~Table();

private:
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int value;
    int currentTurn;           // Starvation
};



#endif // TABLE_HPP
```

table.cpp:

In table.cpp, the first function I implement is the Table::Table(), and the TODO give the hint that this function is used to implement table constructor (value, mutex, cond). Therefore, same as the previos Table::Table() part, I used the function in pthread.h, pthread_mutex_init() and pthread_cond_init(). pthread_mutex_init() is to initialize a mutex, and pthread_cond_init() is to initialize a condition variable. Furthermore, I set the value=4, which is a private variable in Table class. I set it equals to four represents that the table at most allow four philosophers to enter the table.

```
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    currentTurn = 0;                        // Starvation
    value = 4;                              // means that the table can contain n people
    pthread_mutex_init(&mutex, NULL);       // initialize a mutex
    pthread_cond_init(&cond, NULL);         // initialize a condition variable
}
```

Secondly, I implement the Table::wait() function, and the TODO give the hint that this function is used to implement semaphore wait. Hence, in the first step, I use the pthread_mutex_lock() function to represent that the process enters the critical section. Then, I set a while() loop, and if the value is bigger than or equal to 0, it will keep running the loop. In the loop, I use the pthread_cond_wait() function, which will Wait for condition variable to be signaled or broadcast and MUTEX is assumed to be locked before. After the while() loop, set value decreases 1. It means that there is someone enter the table, so the rest amount of people can enter the table decreases 1. Finally, use the pthread_mutex_unlock() function to release the lock and represent that the process leaves the critical section. The variable "value" will change in Table::wait(), however, we use mutex lock and unlock to ensure there won't occur the critical section problem.

```
void Table::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);                 // enter critical section, lock mutex
    while(value <= 0){
        pthread_cond_wait(&cond, &mutex);       // Wait for condition variable COND to be signaled or bro
                                                // MUTEX is assumed to be locked before.
    }
    value--;                                    // means that the amount of people the table can contain
    pthread_mutex_unlock(&mutex);               // leave critical section, release the lock
}
```

Thirdly, I implement the Table::signal() function, and the TODO give the hint that this function is used to implement semaphore signal. Hence, in the first step, I use the pthread_mutex_lock() function to represent that the process enters the critical section. Then, set value increases 1. It means that the amount of people can enter the table increases, that is, someone leaves it. After that, I used pthread_cond_signal() to wake up one thread waiting for condition variable, which is the pthread_cond_wait() part I mentioned in the before Table::wait(). In the end, use the pthread_mutex_unlock() function to release the lock and represent that the process leaves the critical section. The variable ”value” will also change in Table::signal(), however, we use mutex lock and unlock to ensure there won’t occur the critical section problem.

```
void Table::signal() {
    // TODO: implement semaphore signal
    pthread_mutex_lock(&mutex);                 // enter critical section, lock mutex
    value++;                                    // means that the amount of people the table can contain
    pthread_cond_signal(&cond);                 // Wake up one thread waiting for condition variable
    pthread_mutex_unlock(&mutex);               // leave critical section, release the lock
}
```

Last, I implement the Table:: Table() function, and the TODO give the hint that this function is used to implement Table destructor (mutex, cond). Thus, I used the function in pthread.h, pthread_mutex_destroy() and pthread_cond_destroy(). pthread_mutex_destroy() is to destroy a mutex, and pthread_cond_destroy() is to destroy a condition variable.

```
Table::~Table() {
    // TODO: implement table destructor (mutex, cond)
    pthread_mutex_destroy(&mutex);          // destroy a mutex
    pthread_cond_destroy(&cond);            // destroy a condition variable
}
```

These are the part of my Table implement.

1.3 Philosopher

philosopher.hpp:
Below is the screenshot of my philosopher.hpp. I didn’t make ant change of it. It is same as the original .hpp copy from the server.

```
#ifndef PHILOSOPHER_HPP
#define PHILOSOPHER_HPP

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "fork.hpp"
#include "table.hpp"
#include "config.hpp"

class Philosopher {
public:
    Philosopher(int id, Fork *leftFork, Fork *rightFork, Table *table);
    void start();
    int join();
    int cancel();
    void think();
    void enter();
    void pickup(int id);
    void eat();
    void putdown(int id);
    void leave();

private:
    static void* run(void* arg);
    int id;
    bool cancelled;
    Fork *leftFork, *rightFork;
    Table *table;
    pthread_t t;
    time_t t1;
};


#endif // Philosopher_Hpp
```

philosopher.cpp:

The first function is Philosopher::Philosopher(). I didn't make ant change of it. It is same as the original .hpp copy from the server. The purpose is to declare the variable it need to use.

```
Philosopher::Philosopher(int id, Fork *leftFork, Fork *rightFork, Table *table) :id(id), cancelled(false), leftFork(leftFork), rightFork(rightFork), table(table) {
    srand((unsigned) time(&t1));
}
```

Secondly, I implement the philosopher::start() function, and the TODO give the hint that this function is used to start a philosopher thread. Therefore, I used the pthread_create(&t, NULL, &Philosopher::run, this) to create a new thread(&t), starting with execution of START-ROUTINE(&Philosopher::run) getting passed ARG(this). Creation attributed come from ATTR(NULL). The new handle is stored in *NEWTHREAD.

```
void Philosopher::start() {
    // TODO: start a philosopher thread
    pthread_create(&t, NULL, &Philosopher::run, this);


}
```

Thirdly, I implement the philosopher::join() function, and the TODO give the hint that this function is used to join a philosopher thread. Hence, I used the pthread_join(t, NULL) to make calling thread wait for termination of the thread TH(t). The exit status of the thread is stored in *THREAD_RETURN(NULL), if THREAD_RETURN is not NULL, however, it is NULL here.

```
int Philosopher::join() {
    // TODO: join a philosopher thread
    return pthread_join(t, NULL);


}
```

Fourthly, I implement the philosopher::cancel() function, and the TODO give the hint that this function is used to cancel a philosopher thread. Thus, in the function I will set the bool variable "cancelled" to be true, which means that this philosopher is been cancelled. Then, I used the pthread_cancel(t) function to cancel THREAD(t) immediately or at the next possibility.

```
int Philosopher::cancel() {
    // TODO: cancel a philosopher thread
    cancelled = true;
    return pthread_cancel(t);
}
```

The fifth function is Philosopher::think(). I didn't make ant change of it. It is same as the original .hpp copy from the server. The purpose is to calculate the think time of the philosopher, and use the sleep() fuunciton to let the philosopher sleep the think time that calculate at the last step.

```
void Philosopher::think() {
    int thinkTime = rand() % MAXTHINKTIME + MINTHINKTIME;
    sleep(thinkTime);
    printf("Philosopher %d is thinking for %d seconds.\n", id, thinkTime);
}
```

The sixth function is Philosopher::eat(). I didn't make ant change of it. It is same as the original .hpp copy from the server. The purpose is to declare that this philosopher is eating now, and use the sleep function to that the philosopher sleeps 2 second, which is the definition of eattime defined in the config.hpp.

```
void Philosopher::eat() {
    printf("Philosopher %d is eating.\n", id);
    sleep(EATTIME);
}
```

Seventhly, I implement the philosopher::pickup() function, and the TODO give the hint that this function is used to implement the pickup interface, and the philosopher needs to pick up the left fork first, then the right fork. Thus, in the function I call the leftFork->wait() to check out whether this philosopher can pick up the left fork, if not, he will wait ; also, call the rightfork->wait() to check out whether this philosopher can pick up the right fork, if not, he will wait. If he picked up both the left and right fork, I set a printf() to show and check out.

```
void Philosopher::pickup(int id) {
    // TODO: implement the pickup interface, the philosopher needs to pick up the left fork first, then the right fork
    //printf("Philosopher %d is trying to pick up forks.\n", id);

    leftFork->wait();
    //printf("Philosopher %d picked up left fork.\n", id);
    rightFork->wait();
    //printf("Philosopher %d picked up right fork.\n", id);
    printf("Philosopher %d picked up both left and right fork.\n", id );
    //table->updateCurrentTurn();                        // Starvation
}
```

Eighthly, I implement the philosopher::putdownp() function, and the TODO give the hint that this function is used to implement the putdown interface, the philosopher needs to put down the left fork first, then the right fork. Thus, in the function I call the leftFork->signal() to let the philosopher put down the left fork ; also, call the rightfork->signal() to let the philosopher put down the right fork. If he put down both the left and right fork, I set a printf() to show and check out.

```
void Philosopher::putdown(int id) {
    // TODO: implement the putdown interface, the philosopher needs to put down the left fork first, then the right fork
    leftFork->signal();
    //printf("Philosopher %d put down left fork.\n", id );
    rightFork->signal();
    //printf("Philosopher %d put down right fork.\n", id );
    printf("Philosopher %d put down both left and right fork.\n", id );
}
```

Ninthly, I implement the philosopher::enter() function, and the TODO give the hint that this function is used to implement the enter interface, the philosopher needs to join the table first. Accordingly, I call the table->wait() to to check out whether this philosopher can enter the table, if not, he will wait. If he enter the table, I set a printf() to show and check out.

```
void Philosopher::enter() {
    // TODO: implement the enter interface, the philosopher needs to join the table first
    table->wait();
    printf("Philosopher %d enter the table.\n", id);
}
```

Tenthly, I implement the philosopher::leave() function, and the TODO give the hint that this function is used to implement the leave interface, the philosopher needs to let the table know that he has left. Accordingly, I call the table->signal() to to check out whether this philosopher left the table. If he left the table, I set a printf() to show and check out.

```
void Philosopher::leave() {
    // TODO: implement the leave interface, the philosopher needs to let the table know that he has left
    table->signal();
    printf("Philosopher %d leave the table.\n", id);
}
```

Last, I implement the philosopher::Run() function, and the TODO give the hint that this function is used to complete the philosopher thread routine. Therefore, I set a p of philosopher class to record the philosopher now is executed. And, used pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL) to set cancellation state of current thread to TYPE(PTHREAD_CANCEL_DEFERRED), which means if there is a cancel at this stage it will be deferred. After that, enter the while loop(), in the while() loop, first call the think() function to let the philosopher think. Then, used pthread_setcanceltype(PTHREAD_CANCEL_DISABLE, NULL) to set cancellation state of current thread to TYPE(PTHREAD_CANCEL_DISABLE), which means if there is a cancel at this stage it will be disable. Next, call the enter() to let the philosopher enter the table. After entering the table, call the pickup function to let the philosopher pick up the forks. After picking up, call eat() to let the philosopher eat. After eating, call putdown() to make philosopher put down the forks. After putting down, call leave() function to make the philosopher leave the table. In the end, used pthread_setcanceltype(PTHREAD_CANCEL_ENABLE, NULL) to set cancellation state of current thread to TYPE(PTHREAD_CANCEL_ENABLE), which means the cancel is enable now.

```cpp
void* Philosopher::run(void* arg) {
    // TODO: complete the philosopher thread routine.
    Philosopher* p = static_cast<Philosopher*>(arg);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);   // Set cancellation state of current thread to TYPE, returning the old
                                                            // type in *OLDTYPE if OLDTYPE is not NULL.

    while (!p->cancelled) {
        p->think();
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

        p->enter();
        printf("Now is %d's turn.\n", p->table->getCurrentTurn());

        if (p->table->getCurrentTurn() == p->id || ((p->table->getCurrentTurn()+1)%5 != p->id  && (p->table->getCurrentTurn()-1)%5 != p->id)) {
            //printf("Now is %d's turn.\n", p->table->getCurrentTurn());
            p->pickup(p->id); // Attempt to pick up forks if it's the philosopher's turn

            p->eat();

            p->putdown(p->id); // Put down forks

        }
        //p->table->wait();          // Starvation
        /*p->pickup(p->id);
        p->eat();
        p->putdown(p->id);*/
        //p->table->signal();         // Starvation
        //p->table->updateCurrentTurn();
        p->leave();

        p->table->updateCurrentTurn();

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    }
```

These are the part of my Philosopher implement.

1.4 Starvation problem(Bonus)

To prevent the possibility of occurring starvation problem, I decide to set a fair-access mechanism to let every philosopher has its turn to enter the table. Below is how I implement:

table.hpp:
In the table.hpp, I add a private variable ”currentTurn” to record what the currentTurn is now. Furthermore, I also add two public function ”getCurrentTurn()” to get the what the currentTurn is and ”updateCurrentTurn()” to update the currentTurn.

```cpp
#ifndef TABLE_HPP
#define TABLE_HPP

#include <pthread.h>

class Table {
public:
    Table(int n);
    void wait();
    void signal();
    int getCurrentTurn();     // Starvation
    void updateCurrentTurn(); // Starvation
    ~Table();

private:
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int value;
    int currentTurn;          // Starvation
};



#endif // TABLE_HPP
```

table.cpp:

In the table.cpp I did the below implementation:

First is in the Table::Table(), I set the currentTurn be zero, it is the initial turn.

```
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    currentTurn = 0;                        // Starvation
    value = 4;                              // means that the table can contain n people
    pthread_mutex_init(&mutex, NULL);       // initialize a mutex
    pthread_cond_init(&cond, NULL);         // initialize a condition variable
}
```

The other thing I implemented is I wrote the getCurrentTurn() function and updateCurrentTurn() in the table.cpp. getCurrentTurn() is simple, it just return the currentTurn. updateCurrentTurn() will add one to the currentTurn and check whether it is bigger than the amount of total philosopher, if yes, diveide the amount of total philosopher and set the remainder to be the currentTurn.

```
int Table::getCurrentTurn(){                        // Starvation
    return currentTurn;
}

void Table::updateCurrentTurn(){                    // Starvation
    currentTurn = (currentTurn + 1) % PHILOSOPHERS;
    printf("update to %d's turn.\n", currentTurn);
}
```

philosopher.cpp:

The thing I implement in philosopher.cpp are all in the Philosopher::Run() function.

```
void* Philosopher::run(void* arg) {
    // TODO: complete the philosopher thread routine.
    Philosopher* p = static_cast<Philosopher*>(arg);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);   // Set cancellation state of current thread to TYPE, returning the old
                                                            // type in *OLDTYPE if OLDTYPE is not NULL.

    while (!p->cancelled) {
        p->think();
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

        p->enter();
        printf("Now is %d's turn.\n", p->table->getCurrentTurn());

        if (p->table->getCurrentTurn() == p->id || ((p->table->getCurrentTurn()+1)%5 != p->id  && (p->table->getCurrentTurn()-1)%5 != p->id)) {
            //printf("Now is %d's turn.\n", p->table->getCurrentTurn());
            p->pickup(p->id); // Attempt to pick up forks if it's the philosopher's turn

            p->eat();

            p->putdown(p->id); // Put down forks

        }
        //p->table->wait();          // Starvation
        /*p->pickup(p->id);
        p->eat();
        p->putdown(p->id);*/
        //p->table->signal();         // Starvation
        //p->table->updateCurrentTurn();
        p->leave();

        p->table->updateCurrentTurn();

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    }
```

In Philosopher::Run() in the while loop I use a if () statement to restrict the philosopher can pick up the forks. Only the philosopher is its turn or it is not the current turn's left or right neighbor cam pick up the forks, else they can't pick up the fork until it meets the condition. And Once there is a philosopher finished eating and leave the table it will call the updateCurrentTurn() function to update the currentTurn.

These are how I implement a fair-access mechanism to prevent starvation problem.

2 Questions

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

The reason that pthread_cond_wait() requires a mutex variable as the second parameter, while pthread_cond_signal()' does not, is related to the purpose and usage of condition variables in concurrent programming.

In concurrent programming, condition variables are typically used to synchronize the execution of threads based on certain conditions. The pthread_cond_wait() function is used by a thread to wait for a condition to become true before proceeding further. When a thread calls pthread_cond_wait(), it atomically releases the associated mutex and waits for a signal on the condition variable.

The mutex is required as a parameter to 'pthread_cond_wait()' for two reasons:

First is Mutual Exclusion => The mutex ensures mutual exclusion and protects the shared data associated with the condition. When a thread is waiting on a condition variable, it releases the mutex to allow other threads to modify the shared data safely.

Second is Atomicity: The mutex provides atomicity to the entire wait operation. It prevents critical section problem between checking the condition and entering the wait state. Without the mutex, another thread might modify the condition just after the first thread checks it, leading to incorrect behavior.

On the other hand, pthread_cond_signal() is used to signal a waiting thread that a condition has changed and it should evaluate the condition again. It is called when a thread has finished modifying the shared data and wants to wake up a waiting thread.

The mutex is not required for pthread_cond_signal() because it does

not modify shared data directly. It is simply a signal to wake up a waiting thread.

In summary, the mutex parameter in 'pthread_cond_wait()' is necessary for mutual exclusion and atomicity, while 'pthread_cond_signal()' does not require a mutex for its specific signaling purpose.

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

The part of the implementation ensures the fork is only used by one philosopher is the Fork::wait() and the Fork::signal() in the fork.cpp. The wait() will check whether this fork is able to be used now, if yes, its value will be one, otherwise, its value will be zero if it is being used now. And the signal() will be called if there is any philosopher put down the fork, so the value will return to one to represent no one use it now. The thing need to notice is that the value will be used synchronized so we need to use the mutex function to enter and leave the critical section to avoid the critical section problem.

```
void Fork::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);                    // enter critical section, lock mutex
    while(value != 1){
        pthread_cond_wait(&cond, &mutex);          // Wait for condition variable COND to be signaled or broadc
                                                   // MUTEX is assumed to be locked before.
    }
    value = 0;                                     // means that this fork is not able to be used
    pthread_mutex_unlock(&mutex);                  // leave critical section, release the lock
}
```

```
void Fork::signal() {
    // TODO: implement semaphore signal
    pthread_mutex_lock(&mutex);                    // enter critical section, lock mutex
    value = 1;                                     // means that this fork is able to be used
    pthread_cond_signal(&cond);                    // Wake up one thread waiting for condition variable
    pthread_mutex_unlock(&mutex);                  // leave critical section, release the lock
}
```

2.3 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?

The part of the implementation avoids the deadlock is the limitation that only allow at most 4 philosopher to allow the able, and there are 5 seats, so there won't be a deadlock occurred. It avoid the deadlock by limit resource allocation strategy. And I implement it in Table::Table() function in the table.cpp => set the value n(the ampunt of philosopher can enter the table) equals to 4.

```
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    currentTurn = 0;                          // Starvation
    value = 4;                                // means that the table can contain n people
    pthread_mutex_init(&mutex, NULL);         // initialize a mutex
    pthread_cond_init(&cond, NULL);           // initialize a condition variable
}
```

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

After doing the basic implementation, there is still the possibility of occurring starvation problem, however, doing the bonus part I mentioned above. The program will become starvation-free. I create a fair-access mechanism in the bonus part, thus, every philosopher will have his turn to enter the table. Therefore, the program will be starvation-free.

2.5 What is the purpose of using pthread_setcancelstate()?

The purpose of using pthread_setcancelstate() is to control the thread's response to cancellation requests. It allows you to specify whether a thread should be cancellable or not, based on the requirements of the program.

In this program, there are three type of pthread_setcancelstate been used.

1. **PTHREAD_CANCEL_DEFERRED**: This type defers the cancellation request until the thread reaches a cancellation point. The cancellation request is held pending until the thread voluntarily enters a cancellation point by making a cancellation-enabled function call.

2. **PTHREAD_CANCEL_ENABLE**: This state enables cancellation for the thread. If cancellation is enabled and a cancellation request is received, the thread will be canceled.

3. **PTHREAD_CANCEL_DISABLE**: This state disables cancellation for the thread. If cancellation is disabled, cancellation requests will be deferred until cancellation is enabled again.

In summary, pthread_setcancelstate() provides control over thread cancellation, allowing you to manage cancellation points and ensure thread safety during critical sections of code.

3 Explain the pros and cons of using monitor to solve a dining-philosopher problem compared to this homework ?

Pros of using a monitor:

Simplified synchronization: Monitors provide a higher-level abstraction for synchronization compared to semaphores and mutexes. They encapsulate the shared resources and the synchronization mechanisms into a single construct, making it easier to reason about the correctness of the solution.

Encapsulation of synchronization logic: With a monitor, the synchronization logic is encapsulated within the monitor itself. This reduces the chances of errors or oversights in managing locks, condition variables, and state transitions manually.

Easier to understand and maintain: Monitors promote a more structured and intuitive approach to synchronization. The code is organized around the shared resource and its associated operations, making it easier to understand and maintain the solution.

Automatic resource management: Monitors often provide automatic resource management, ensuring that locks are acquired and released correctly. This reduces the risk of deadlocks, race conditions, and other synchronization-related issues.

Cons of using a monitor:

Language and library support: Monitors are typically supported by specific programming languages or libraries. If the programming language or library being used does not provide support for monitors, implementing a monitor-based solution can be more challenging.

Limited to a specific programming paradigm: Monitors are closely tied to the concept of object-oriented programming (OOP) and may not fit well with other programming paradigms. If working in a non-OOP context, using monitors might not be the most natural or efficient approach.

Less flexibility and customization: Monitors provide a predefined set of synchronization mechanisms, and their usage might not be flexible enough to handle complex synchronization scenarios. If needing fine-grained control over synchronization or if having specific requirements that don't fit the monitor abstraction, using monitors may limit options.

In summary, if using a monitor to solve the dining philosophers problem offers simplified synchronization, encapsulation of synchronization logic, and improved code readability. However, it relies on language and library support, may be tied to a specific programming paradigm, and can be less flexible in certain scenarios. The approach of semaphores and mutexes are more control and flexibility, but it requires manual management of synchronization primitives. The choice between the two approaches depends on the specific requirements, constraints, and preferences of the project or context in which the problem is being solved.

4 Feedback

1. I have a question about this homework. I am not very sure the role think() play in the eating process. No matter set the think() after entering the table or before entering the table, it will make sense, and won't cause any wrong. Or there is something I neglect. Maybe TA can notice me.

2. Thank for the instruction from professor and TA in this semester. I think OS is such a challenging courses, and its homework is difficult, too. However, through this semester, I also learn a lot of things about OS. Thank for professor and TA again!!!