2.
Fork

```
Fork::Fork() {
    // TODO: implement fork constructor (value, mutex, cond)

    pthread_mutex_init(&mutex, NULL);
    value = 1;
    //avaliable = true;
    if (pthread_cond_init(&cond, NULL) != 0) {
       fprintf(stderr, "Error : pthread_cond_init \n");
    }

}

void Fork::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);
    while(value!=1){  //if not avaliable
        pthread_cond_wait(&cond,&mutex);
    }
    value = 0;
    pthread_mutex_unlock(&mutex);

}

void Fork::signal() {
    // TODO: implement semaphore signal
    pthread_mutex_lock(&mutex);
    value = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

Fork::~Fork() {
    // TODO: implement fork destructor (mutex, cond)

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

}
```

首先宣告 mutex 和 condition variable,value 設成 1

Value 只會在 1 和 0 徘徊,如果有人使用了這個 fork 會把他設成 0.

Wait():如果要使用這個 fork,會 call 這個函式,把 value 設成 0,並且使用 pthread_cond_wait 將這個 fork put to sleep.

由於會使用 value 這個 share data,頭尾必須使用 lock 以及 unlock

Signal():當哲學家放下這個 fork 到餐桌上的時候會 call 這個函式,把 value 設成 1,並且使用 pthread_cond_signal wake up

這個 fork.由於會使用 value 這個 share data,頭尾必須使用 lock 以及 unlock

最後在解構子把 mutex 以及 cond release 掉.

Table

```
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    pthread_mutex_init(&mutex, NULL);
    value = n;
    if (pthread_cond_init(&cond, NULL) != 0) {
        fprintf(stderr, "Error : pthread_cond_init \n");
    }
}

void Table::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);
    while(value<=1){
        //printf("Now full of philosopher\n");
        pthread_cond_wait(&cond,&mutex);
    }
    value--;
    //printf("%d\n",value);
    pthread_mutex_unlock(&mutex);
}

void Table::signal() {
    // TODO: implement semaphore signal
    pthread_mutex_lock(&mutex);
    value++;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

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

首先宣告 mutex 和 condition variable,value 設成 n

Value 會在 2~5 徘徊(5 是哲學家的數量)

Wait():如果要進入這個 table,會 call 這個函式,把 value--,如果 value<=1(代表進去 4 位哲學家了)就使用 pthread_cond_wait 把 table put to sleep,確保只有 4 個哲學家在用餐.

由於會使用 value 這個 share data,頭尾必須使用 lock 以及 unlock

Signal():當哲學家離開餐桌的時候會 call 這個函式,把 value++,並且使用 pthread_cond_signal wake up 這個 table,讓其他哲學家可以進來 table 用餐(餐桌上仍然最多 4 個人).由於會使用 value 這個 share data,頭尾必須使用 lock 以及 unlock

最後在解構子把 mutex 以及 cond release 掉.

Philosopher:

```cpp
Philosopher::Philosopher(int id, Fork *leftFork, Fork *rightFork, Table *table) :id(id), cancelled(false), leftFork(leftFork), rightFork(rightFork), table(table), priority(0
    srand((unsigned) time(&t1));



}

void Philosopher::start() {
    // TODO: start a philosopher thread

    if(pthread_create(&t,NULL,run,this) != 0){
        fprintf(stderr, "Error: pthread_create\n");
    }

}

int Philosopher::join() {
    // TODO: join a philosopher thread
    pthread_join(t,NULL);
    //printf("Now using join()\n");
}

int Philosopher::cancel() {
    // TODO: cancel a philosopher thread
    pthread_cancel(t);
    //printf("Now using cancel()\n");
}

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

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

```cpp
void Philosopher::pickup(int id) {
    // TODO: implement the pickup interface, the philosopher needs to pick up the left fork first, then the right fork
    leftFork->wait();
    printf("Now %d philosopher pick up %d fork \n",id,id);
    rightFork->wait();
    printf("Now %d philosopher pick up %d fork \n",id,(id+1)%PHILOSOPHERS);


}

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("Now %d philosopher put down %d fork \n",id,id);
    rightFork->signal();
    printf("Now %d philosopher put down %d fork \n",id,(id+1)%PHILOSOPHERS);

}

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

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

```
void* Philosopher::run(void* arg) {
    // TODO: complete the philosopher thread routine.
    // Run pick up put down

    Philosopher *p = (Philosopher*) arg;


    p->enter();
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    //enter放外面:philosopher沒進table就用餐了, enter放裡面:可能有5個philosopher在餐桌
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while (!p->cancelled) {
        //pickup


        p->think();
        p->pickup(p->id);
        p->eat();
        p->putdown(p->id);
        //putdown
        //test();
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

        p->leave();
        //加入aging機制
        p->enter();
    }



    //p->leave();


    return NULL;
}
```

Start():create 這個 thread

Join():main thread 之外的 thread 要執行完畢

Cancel():把 thread 取消掉,不使用的話程式會無限執行下去

Think():哲學家在思考一個 random 時間

Eat():哲學家在 eating

Pickup():將左右叉子拿起來,call wait(),將 value 設成 0,並把叉子 put to sleep

Putdown():將左右叉子放下來 call signal,將 value 設成 1,並

wake up 左右叉子

Enter():call wait(),哲學家進入這個 table,最多四個哲學家進入

Leave():call signal(),哲學家離開這個 table

Run():一開始哲學家入座,並且在 loop 不斷思考 拿筷子 吃飯 放筷子,如果哲學家要離開可以再由其他哲學家進入 table

3.

1.由於上下會有 mutex_lock 綁住如果 call pthread_cond_wait()時,該 thread 會卡在 critical section 出不去,而這時候由於該 thread 已經把 mutex 搶走了,導致其他 thread 進不去 critical section,導致 deadlock,所以要在 pthread_cond_wait()的參數傳入 mutex parameter,當 thread 被 put to sleep 的時候把 mutex 釋放出來,讓其他 thread 可以進入他們的 critical section.

而,pthread_cond_signal()僅僅是發送一個訊號通知等待在 condition variable 的 thread,不涉及 mutex 的操作,因此不需要 mutex 當作參數.

2.

```
void Fork::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);
    while(value!=1){  //if not avaliable
        pthread_cond_wait(&cond,&mutex);
    }
    value = 0;
    pthread_mutex_unlock(&mutex);

}
```

這一個部分.創建 fork 的時候會把 value initialize 成 1,當 call wait()時候會將 value 設成 0,代表這個 fork 正在被使用,且會被 put to sleep,等待其他人 call signal(),叫醒這個 fork, value 只有 0 跟 1,確保一次只有一個哲學家使用.

3.

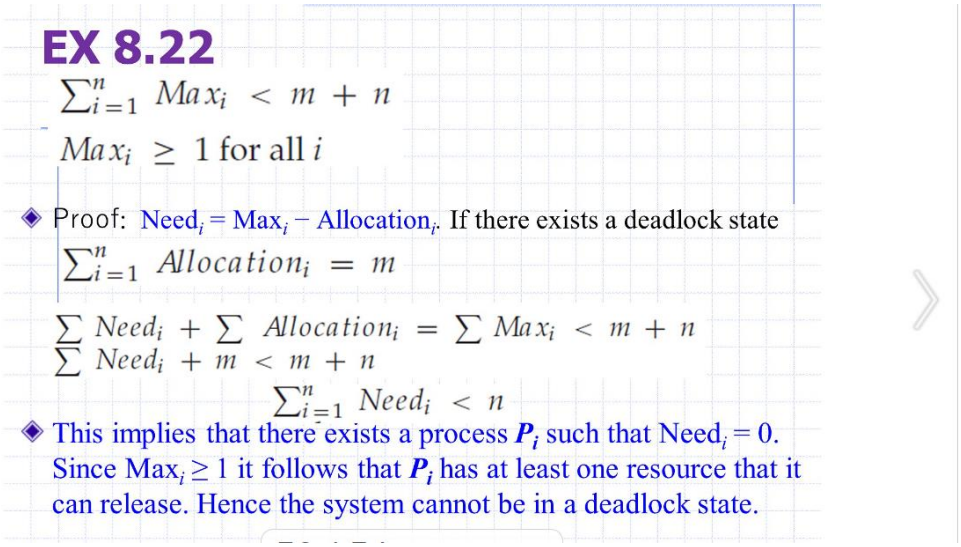
```
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    pthread_mutex_init(&mutex, NULL);
    value = n;
    if (pthread_cond_init(&cond, NULL) != 0) {
       fprintf(stderr, "Error : pthread_cond_init \n");
    }
}

void Table::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);
    while(value<=1){
        //printf("Now full of philosopher\n");
        pthread_cond_wait(&cond,&mutex);
    }
    value--;
    //printf("%d\n",value);
    pthread_mutex_unlock(&mutex);
}
```

這部分,由於五個哲學家只取四個哲學家,至少會多出來一個

resource 可以釋放,一個人一定可以完工並且放掉資源

概念類似這題:



4.並沒有 starvation-free,如果 A 哲學家離開後,馬上回到餐桌,就有可能有一個 philosopher 完全沒機會拿到 fork 用餐

5.

在迴圈以前就把 state 設成 disable,這樣做的目的是確保在執行 critical section 時不被取消,確保在做一系列動作後才可以取消,維持程式的完整性和一致性.

這樣的設計可以控制 thread 在特定時刻是否可以被取消,並確保系統的正確運行和資源的正確釋放.

4.

Monitor 優點:

提供一個簡單且直觀的機制解決同步問題,容易理解和實作,

且提供一個 high-level 的抽象層將 share data 和相關的同步機制封裝再一個 module 裡,並且 monitor 確保同一時間只有一個 thread 能夠使用 share data,避免 race condition

Monitor 缺點:

1.通常需要使用 lock,很多 thread 在等待 lock 的釋放,影響整體效能

2.只有單一點入口,當一個 thread 進入 monitor 時候,thread 必須等待該 thread 完成操作才能進入,尤其是程式有大量同步需求時候.

3.另外,所有 thread 都必須進入 monitor 才能共享 data,可擴展性有問題.

不用 monitor 優點:

1. 可以根據問題需求使用合適的同步機制,如 semaphore 和 mutex
2. 不需要進入和離開 monitor 的同步成本,提高效率

不用 monitor 缺點:

需要 programmer 去做同步的問題,可能會出錯,增加程式的

複雜性和錯誤

可讀性降低,因為同步需要寫在程式裡面,會讓程式碼變的冗長難以理解.

5.

如果可以在作業截止前,上一個作業的成績可以出來是最好的~