作業系統 - MP4 – Dining Philosophers

tags NachOS

studentID: 1102003S

workshop account: os23s77

teamID: 陳觀宇 (kcem104@gmail.com)

name: 陳觀宇

HackMD link: LINK

(建議用HackMD看,格式比PDF美觀)

Part 1. Implementation

Class Fork

Fork::Fork()

```
Fork::Fork() {
    // TODO: implement fork constructor (value, mutex, cond)
    value = 1;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
}
```

- 第3行 : 設定value初始值為 1，每個 Fork 物件的 value 表示該 Fork 目前的狀態，1 代表 Fork 是可用的，0 代表 Fork 被佔用。
- 第4行 : 初始化了一個互斥鎖（mutex），並將其儲存在 Fork 物件的成員變數 mutex 中。互斥鎖用於實現臨界區（critical section），確保在同一時間只有一個執行緒能夠訪問共享資源。
- 第5行 : 初始化了一個條件變量（condition variable），並將其儲存在 Fork 物件的成員變數 cond 中。條件變量用於在執行緒之間進行通信和同步，它可以等待特定的條件滿足或發出信號來通知其他執行緒。

void Fork::wait()

```
void Fork::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);
    while (value <= 0) {
        pthread_cond_wait(&cond, &mutex);
    }
    value--;
    pthread_mutex_unlock(&mutex);
}
```

- 第3&8行 : 對互斥鎖（mutex）進行加鎖與解鎖操作。為了確保在進行條件變量檢查之前，不會有其他執行緒進入叉子的臨界區域。
- 第4行 : 使用一個 while 迴圈來檢查叉子的 value 狀態。如果 value 小於等於 0，表示該叉子當前不可用，有其他哲學家正在使用它。
- 第5行 : 符合while條件情況下，該執行緒會呼叫 pthread_cond_wait(&cond, &mutex)，這會將該執行緒放入等待狀態，同時釋放叉子的鎖，並等待其他執行緒發送信號。當另一個執行緒釋放叉子並發送信號後，當前執行緒會被喚醒，並再次檢查 value 的狀態。如果 value 仍然小於等於 0，則再次進入等待狀態，否則，該執行緒可以繼續執行下去。
- 第7行 : 如果當 value 大於 0 時，表示叉子可用。將value減少 1，表示叉子被占用而被哲學家拿走。

void Fork::signal()

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

- 第3&6行 : 對互斥鎖（mutex）進行加鎖與解鎖操作。為了讓叉子的狀態value不會受到Race Condition影響，因此需要使用mutex讓value在critical section內，達到互斥存取效果。以確保在進行後續操作之前，只有當前的執行緒能夠訪問 Fork 物件。
- 第4行 : value++ 將 Fork 的狀態 value 增加 1，表示 Fork 已經被釋放，變成可用狀態。
- 第5行 : 發送一個信號給條件變量（condition variable）cond。這個信號通知等待該條件變量的其他哲學家，表示 Fork 已經被釋放，其他等待這個 Fork 的哲學家可以使用這個 Fork 。

Fork::~Fork()

```
Fork::~Fork() {
    // TODO: implement fork destructor (mutex, cond)
    pthread_mutex_destroy(&mutex);
```

```
    pthread_cond_destroy(&cond);
}
```

- 第3&4行 : 呼叫pthread的destructor刪除Fork的mutex&cond

Class Table

Table::Table(int n)

```
Table::Table(int n) {
    // TODO: implement table constructor (value, mutex, cond)
    value = n - 1;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
}
```

- 第1行 : Table建構子函式接收一個整數 n 作為參數，n 表示哲學家的數量。
- 第3行 : 將n - 1 賦值給 value 變數。value 變數表示有幾把可用的叉子，等同於有幾個可以拿叉子吃飯的哲學家人數，等同於目前有多少人能夠進入Table用餐。
- 初始化為 n - 1 的原因是在桌子上有 n 把叉子，但每個哲學家需要兩把叉子來吃飯，所以只有 n - 1 把叉子是初始可用的，也就是允許最多只有 n - 1 個哲學家能拿叉子，等同於允許最多只有 n - 1 個人能進入Table用餐。設定為 n - 1 才能夠避免circular waiting導致Deadlock產生。
- 第4行 : 初始化互斥鎖 mutex。互斥鎖用於確保對Table物件的操作是互斥存取。
- 第5行 : 初始化條件變量cond。此條件變量用於控制哲學家進入和離開桌子的操作，以確保只有當桌子上有可用的叉子時，哲學家才能進入吃飯。

void Table::wait()

```
void Table::wait() {
    // TODO: implement semaphore wait
    pthread_mutex_lock(&mutex);
    while (value <= 0) {
        pthread_cond_wait(&cond, &mutex);
    }
    value--;
    pthread_mutex_unlock(&mutex);
}
```

- 第3&8行 : 對互斥鎖（mutex）進行加鎖與解鎖操作。以確保在進行等待操作時不會有其他執行緒同時存取Table物件。
- 第4行 : 使用 while 迴圈檢查 value 變數的值是否小於等於 0。value 變數表示可用的叉子數量，如果值小於等於 0，表示沒有可用的叉子，此時執行緒需要進入等待狀態。

- 第5行 : 在進入等待狀態之前，使用 pthread_cond_wait(&cond, &mutex) 函式讓執行緒等待條件變量 cond。這個函式會將執行緒置於等待狀態，同時釋放之前鎖住的互斥鎖 mutex，讓其他執行緒能夠存取Table物件並改變其狀態。當有其他執行緒調用 pthread_cond_signal(&cond) 函式來通知等待中的執行緒時，該執行緒會被喚醒並重新鎖住互斥鎖 mutex，繼續執行。
- 第7行 : 如果當 value 大於 0 時，表示目前有可以使用的叉子，將 value 減 1，表示取走一把叉子，也代表Table的座位減少一個(目前能進入table用餐的人數減一)。

void Table::signal()

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

- 第3&6行 : 對互斥鎖（mutex）進行加鎖與解鎖操作。為了讓可用的叉子數量value不會受到Race Condition影響，因此需要使用mutex讓value在critical section內，達到互斥存取效果，以確保在進行通知操作時不會有其他執行緒同時存取Table物件。
- 第4行 : 將 value 變數加 1，表示增加可用的叉子數量，也就是目前可進入Table用餐的人數增加 1。
- 第5行 : 發送一個信號給條件變量（condition variable）cond。這個信號通知等待該條件變量的其他哲學家，表示 Table 已經被釋放，其他等待這個 Table 的哲學家可以進入Table用餐。

Table::~Table()

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

- 第3&4行 : 呼叫pthread的destructor刪除Table的mutex&cond

Class Philosopher

void Philosopher::start()

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

- 第3行 : 呼叫 pthread_create() 函式來建立一個新的執行緒。這個函式需要四個參數：
    - &t：用於存放新建立的執行緒的識別碼。
    - NULL：用於設置執行緒的屬性，這裡使用預設屬性。
    - run：指向靜態成員函式 run() 的指針，它是執行緒的入口點。
    - this：指向目前物件的指標，即呼叫 start() 函式的哲學家物件。
- 執行緒建立後，它會開始執行 run() 函式中的程式碼，並且可以同時與其他執行緒並行運行。每個哲學家物件都有自己的執行緒，以便它們可以並行地進行思考、取叉子、進餐等動作。

int Philosopher::join()

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

- 第3行 : 呼叫 pthread_join() 函式，等待指定的執行緒結束並回收其資源。函式需要兩個參數：
    - t：要等待的執行緒的識別碼，這裡是哲學家的執行緒識別碼。
    - NULL：用於接收執行緒的返回值，這裡不需要返回值，因此設置為 NULL。
- pthread_join() 函式會阻塞呼叫它的執行緒，直到指定的執行緒結束為止。當目標執行緒結束後，它的資源會被回收，並且可以通過 pthread_join() 函式的返回值來獲取目標執行緒的退出狀態。
- Philosopher::join() 函式將等待哲學家的執行緒結束並回收資源，並且將 pthread_join() 函式的返回值傳遞給呼叫者。

int Philosopher::cancel()

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

- 第3行 : 將哲學家的 cancelled 標誌設置為 true。這個標誌通常用於表示執行緒是否被取消的狀態。
- 第4行 : 調用 pthread_cancel() 函式來取消指定的執行緒。這將發送一個取消請求給目標執行緒，導致它被中斷並終止。
- pthread_cancel() 函式用於向指定的執行緒發送取消請求。該請求將使目標執行緒接收到一個特殊的取消信號，從而導致它終止執行。

void Philosopher::pickup(int id)

```
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();
    rightFork->wait();
    if(id!=(PHILOSOPHERS-1)){
        printf("Philosopher %d picked up left fork %d & right fork %d.\n", id, id,
id+1);
    }else if(id==PHILOSOPHERS-1){
        printf("Philosopher %d picked up left fork %d & right fork %d.\n", id, id,
0);
    }
}
```

- 第3行 : 哲學家嘗試取起左邊的叉子。這裡使用了 leftFork 的 wait() 函式，該函式會在叉子不可用時阻塞執行緒，直到叉子可用為止。取得左邊叉子後會將左邊叉子的狀態設定為占用。
- 第4行 : 哲學家嘗試取起右邊的叉子。同樣地，這裡使用了 rightFork 的 wait() 函式來等待右邊的叉子變得可用。取得右邊叉子後會將右邊叉子的狀態設定為占用。
- 第5~9行 : 根據哲學家的編號 id 印出目前第幾號哲學家拿起對應編號的左右叉子。根據spec要求，左邊叉子編號與哲學家id相同，右邊叉子編號則是id+1。除了最後一個編號的哲學家(PHILOSOPHERS-1)，其右邊叉子編號為0(編號循環回0)。

void Philosopher::putdown(int id)

```
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();
    rightFork->signal();
    if(id!=(PHILOSOPHERS-1)){
        printf("Philosopher %d put down left fork %d & right fork %d.\n", id, id,
id+1);
    }else if(id==PHILOSOPHERS-1){
        printf("Philosopher %d put down left fork %d & right fork %d.\n", id, id,
0);
    }
}
```

- 第3行 : 哲學家放下左邊的叉子。這裡使用了 leftFork 的 signal() 函式，該函式會將對應哲學家的左邊叉子狀態恢復為可用。
- 第4行 : rightFork->signal();：哲學家放下右邊的叉子。同樣地，這裡使用了 rightFork 的 signal()，該函式會將對應哲學家的右邊叉子狀態恢復為可用。
- 第5~9行 : 根據哲學家的編號 id 印出目前第幾號哲學家放下對應編號的左右叉子。根據spec要求，左邊叉子編號與哲學家id相同，右邊叉子編號則是id+1。除了最後一個編號的哲學家(PHILOSOPHERS-1)，其

右邊叉子編號為0(編號循環回0)。

void Philosopher::enter()

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

- 第3行 : 哲學家呼叫 table 的 wait() 函式，等待進入餐桌。如果餐桌上沒有足夠的可用位置，則此函式會進入等待狀態，直到有可用位置為止。進入餐桌後，目前可以進入Table用餐的人數-1。
- 第4行 : 印出哲學家進入餐桌的訊息，其中 id 是哲學家的編號。

void Philosopher::leave()

```
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 left the table.\n", id);
}
```

- 第3行 : 哲學家呼叫 table 的 signal() 函式，通知餐桌他已經離開。離開餐桌後，目前可以進入Table用餐的人數+1。
- 第4行 : 印出哲學家離開餐桌的訊息，其中 id 是哲學家的編號。

void* Philosopher::run(void* arg)

```
void* Philosopher::run(void* arg) {
    // TODO: complete the philosopher thread routine.
    Philosopher *p = (Philosopher*) arg;
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    while (!p->cancelled) {
        p->think();
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        p->enter();
        p->pickup(p->id);
        p->eat();
        p->putdown(p->id);
        p->leave();
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    }
```

```
    return NULL;
}
```

- 第3行 : 將傳遞給執行緒的引數轉換為 Philosopher 對象的指針，以便後續使用。
- 第4行 : 設置執行緒的取消類型為延遲取消。代表在收到取消請求時，執行緒會繼續執行到下一個取消點才被取消。取消點包含一些可能會導致線程阻塞的函數調用，如 pthread_cond_wait() 和 pthread_join() 等。
- 第5行 : cancelled 標誌為 false，則進入while loop無限循環執行，直到 cancelled 標誌被設置為 true，則跳出while loop。
- 第6行 : 模擬哲學家在思考。
- 第7行 : 在 enter() ~ p->leave() 執行期間，禁用取消請求(PTHREAD_CANCEL_ENABLE)。這是為了確保在哲學家一旦進入餐桌後不會被立即取消動作，而是能繼續執行到哲學家離開餐桌為止。
- 第8行 : 哲學家進入餐桌。
- 第9行 : 哲學家拿起叉子。
- 第10行 : 模擬哲學家在進餐。
- 第11行 : 哲學家放下叉子。
- 第12行 : 哲學家離開餐桌。
- 第13行 : 在 enter() ~ p->leave() 階段之後，重新啟用取消請求(PTHREAD_CANCEL_ENABLE)。此時哲學家已完成一輪操作，現在可以進行取消檢查，如果符合條件可立即中止該執行緒(哲學家)。

Part 2. Question Answering

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(): 如果當thread進入critical section變成waiting狀態，此時該thread需要release thread所持有的 mutex lock，否則其他thread需要讀取或是寫入critical section內的共有變數時，會被卡在critical section外面而無法進入，導致其他thread產生堵塞。直到另一個thread對應的條件變量發出信號。當thread被信號喚醒後，在從pthread_cond_wait()返回之前，它會重新獲取互斥鎖。因此為了要知道thread釋放與獲取的對應持有mutex lock是哪一個，因此需要傳入mutex參數。
- pthread_cond_signal(): 只是發送一個信號來喚醒一個正在等待該條件變量的線程，因此不會直接訪問共有變數，故不需要傳入mutex參數。

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

- 此實作方式確保叉子同一時間只會被一位哲學家使用的部分是使用了互斥鎖(mutex)和條件變量(condition variable)的結合。

- 在Class Fork中，互斥鎖和條件變量用於實現信號量(semaphore)的概念，即表示叉子的可用性。每個叉子都有一個對應的互斥鎖和條件變量。當一位哲學家希望使用一支叉子時，它首先需要調用wait()方法等待該叉子可用。當叉子可用時，它才能被該哲學家使用。

- 當一位哲學家調用wait()方法時，它會先鎖定對應叉子的互斥鎖，然後檢查叉子的可用性。如果叉子不可用(value=0)，則哲學家會進入等待狀態，同時釋放互斥鎖，等待其他哲學家釋放叉子並發出信號調用signal()方法。一旦叉子變為可用(value=1)，該哲學家將獲取叉子，同時減少value的值，表示這支叉子現在已經被使用。

- 當一位哲學家獲取一支叉子時，其他哲學家對同一支叉子的請求將被阻塞，直到該叉子被釋放並且條件變量發出信號。這保證了每支叉子同一時間只會被一位哲學家使用，從而避免了競爭條件和資源衝突。

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?

- Table 建構子函初始化時接收參數n(哲學家的數量)，將value設定為n-1，因為value 變數表示有幾把可用的叉子，等同於有幾個可以拿叉子吃飯的哲學家人數，等同於目前有多少人能夠進入Table用餐。

- value初始化為 n - 1 的原因是在桌子上有 n 把叉子，但每個哲學家需要兩把叉子來吃飯，所以只有 n - 1 把叉子是初始可用的，也就是允許最多只有 n - 1 個哲學家能拿叉子，等同於允許最多只有 n - 1 個人能進入Table用餐。因此設定為 n - 1 才能夠避免circular waiting導致Deadlock產生。

- 以spec為例，假如有5個哲學家(n=5)，則最多同時只能有4個哲學家(n-1=4)進入Table用餐，最多同時只能有4把叉子被同時使用。如果允許5個人進入Table，且每個人都已經持有左邊的叉子，則此時每個人都在等待其他人把右邊的叉子釋放，產生circular waiting的條件，此時deadlock就會產生。

- 不會產生circular waiting(Deadlock)證明如下:

    - 初始參數:
        - n (同時可拿起叉子用餐的哲學家人數)
        - m = 5 (餐桌上有5雙叉子)
        - Max = 2 (每個哲學家可持有資源上限,最多持有2雙叉子)
    - 條件:
        - 1 <= Maxi <= m -> 條件符合
        - Sum(Maxi) < n+m -> 2*n < n+5 -> n < 5
        - n <=4 -> 代表最多同時可拿起叉子用餐的哲學家人數為4人

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

- 此實作方式偏向是FCFS(First Come First Serve)，也就是先呼叫wait的人能夠優先獲取mutex lock，而非Round Robin或是排隊的方式輪流獲取mutex lock，因此不符合bounded-waiting的條件，有機會產生Starvation的情況，讓某一個哲學家總是在等待其他哲學家釋放叉子，而不停地在等待。

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

- pthread_setcancelstate() 函數用於設置執行緒的取消狀態。執行緒的取消狀態可以是取消啟用（PTHREAD_CANCEL_ENABLE）或取消禁用（PTHREAD_CANCEL_DISABLE）兩種狀態。

- 設置取消狀態的目的在於控制執行緒是否可以被取消。當取消狀態為取消禁用時，執行緒不會被取消，即使收到取消請求也會被忽略。而當取消狀態為取消啟用時，執行緒可以被取消。

- 在多執行緒環境中，設置取消狀態是一種管理執行緒取消的方式。在某些重要的程式段或操作期間，我們希望取消請求被暫時禁用，以確保這段程式碼可以完整執行。因此，我們可以在這些區域之前調用 pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL) 來禁用取消請求，並在區域結束之前恢復取消狀態，以使取消請求再次生效。

- pthread_setcancelstate() 的目的是設置執行緒的取消狀態，以控制執行緒是否可以被取消，並提供一種管理和調節取消請求的機制。

Part 3. Comparision with Monitor

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.)

- monitor vs. semaphore & mutex
    - pros:
        - 1.簡單易用：監視器是一種高階抽象，提供了一個統一的界面來管理共享資源的訪問。相較於作業中使用的信號量（semaphore）和互斥鎖（mutex）等原始同步機制，監視器更容易理解和使用。
        - 2.安全性：監視器提供了內部的互斥鎖，可以確保同一時間只有一個執行緒可以進入監視器的保護範圍，從而防止競態條件和資源衝突的發生。這有助於確保程式的正確性和安全性。
        - 3.簡化同步邏輯：監視器提供了內建的等待（wait）和通知（notify）機制，使得執行緒可以更簡單地進行等待和通信，而不需要額外管理信號量或條件變數。這簡化了同步邏輯，降低了程式的複雜度。
    - cons:
        - 1.效能：監視器的內部同步機制可能引入額外的開銷，例如鎖的競爭和等待/通知的處理。在高度競爭的情況下，這可能導致效能下降。相比之下，作業中使用的原始同步機制（信號量和互斥鎖）在性能上可能更為高效。
        - 2.可延展性：監視器的設計通常是基於中央控制的，所有執行緒必須通過監視器進行同步，這可能限制了程式的可延展性。當執行緒數量增加時，監視器可能成為瓶頸，影響整體效能。