BÀI THỰC HÀNH
MÔN HỌC: HỆ PHÂN TÁN
CHƯƠNG 4: ĐỒNG BỘ HÓA

1. Triển khai đồng bộ các luồng trong một chương trình đa luồng sử dụng ngôn ngữ Java.

1.1.Nội dung

Ở buổi học lý thuyết chúng ta đã xem xét vấn đề đồng bộ hóa trong Hệ Phân Tán. Chúng ta đã xem xét vấn đề khi có nhiều tiến trình cùng muốn sử dụng một tài nguyên chia sẻ dùng chung, nhưng tài nguyên này tại một thời điểm chỉ có thể đáp ứng cho một tiến trình (hoặc một vài). Hoặc tình huống khác khi nhiều tiến trình cần thống nhất thứ tự của các sự kiến. Từ đó, chúng ta thấy cần thiết phải có cơ chế/giải thuật thực hiện đồng bộ hóa cho Hệ Phân Tán.
Ở bài thực hành này chúng ta sẽ học cách triển khai các kỹ thuật đồng bộ hóa trong chương trình viết bằng Java. Và để đơn giản hóa vấn đề, chúng ta sẽ xem xét cơ chế đồng bộ hóa đối với các luồng của cùng một tiến trình.

1.2.Yêu cầu

1.2.1. Lý thuyết

- Đồng bộ hóa trong HPT

1.2.2. Phần cứng

- Laptop/PC dùng bất cứ OS nào.

1.2.3. Phần mềm

- bất cứ Java IDE nào

1.3.Các bước thực hành

Chúng ta sẽ mô phỏng môi trường với việc nhiều luồng cùng muốn truy cập vào sử dụng một tài nguyên dùng chung.

Tạo một lớp, đặt tên là *ResourcesExploiter,* với một biến dạng private đặt tên là *rsc* có kiểu *int*. Biến này sẽ được coi như tài nguyên chia sẻ dùng chung. Vì đó là một biến dạng private nên bạn sẽ cần các phương thức để lấy hoặc cập nhật giá trị (set & get):

```
public void setRsc(int n){
        rsc = n;
}

public int getRsc(){
        return rsc;
```

```
}
```

Tiếp tới là phương thức khởi tạo cho lớp này:

```
public ResourcesExploiter(int n){
        rsc = n;
}
```

Thêm vào một phương thức exploit() có nhiệm vụ tăng biến rsc lên 1 đơn vị:

```
public void exploit(){
        setRsc(getRsc()+1);
}
```

Tạo một lớp tên là *ThreadedWorkerWithoutSync* kế thừa từ lớp *Thread* (lớp Thread là lớp có sẵn trong bộ thư viện Java). Trong lớp này, khai báo một biến private lấy tên *rExp* có kiểu *ResourcesExploiter.*
Trong phương thức *run()* mà bạn phải override (khai báo đè), hãy đưa một vòng lặp *for* với 1000 lần gọi phương thức *exploit()* của biến *rExp*.

Tạo một lớp cho chương trình chạy (lớp có phương thức *main*).

Trong phương thức *main*, làm các bước sau:
- Tạo một thực thể tên là *resource* có kiểu *ResourcesExploiter* với giá trị khởi tạo là 0:

```
ResourcesExploiter resource = new ResourcesExploiter(0);
```

- Tạo 3 thực thể có tên là *worker1*, *worker2*, và *worker3* có kiểu *ThreadedWorkerWithoutSync* (đừng quên đưa thực thể resource vừa tạo ở trên vào 3 phương thức khởi tạo:

```
ThreadedWorkerWithoutSync worker1 = new ThreadedWorkerWithoutSync(resource);
ThreadedWorkerWithoutSync worker2 = new ThreadedWorkerWithoutSync(resource);
ThreadedWorkerWithoutSync worker3 = new ThreadedWorkerWithoutSync(resource);
```

- Khởi động 3 luồng trên (bằng cách gọi phương thức *start()*)
- Sau khi khởi động các luồng trên thì đừng quên gọi phương thức join() để chờ tiến trình đó kết thúc công việc.
- Sau đó, in giá trị của biến *rsc* của thực thể *resource*.

Câu hỏi 1: Chạy chương trình trên vài lần. Bạn nhận thấy điều gì? Giải thích!

Bây giờ là lúc chúng ta sẽ đưa các cơ chế đồng bộ vào chương trình. Tạo thêm một lớp tên là *ThreadedWorkerWithSync* tương tự như lớp *ThreadedWorkerWithoutSync* trừ việc nó có áp dụng **synchronized** ở biến *rExp*,

và áp dụng nó cho toàn bộ vòng lặp *for*:


```
synchronized(rExp){
for(int i=0;i<1000;i++){
rExp.exploit();
}
```

Câu hỏi 2: Thay đổi đoạn mã trong chương trình chạy (phương thức *main*), thay đổi kiểu của 3 thực thể worker1-3 thành *ThreadedWorkerWithSync.* Bạn nhận thấy sự thay đổi gì ở đầu ra khi chạy chương trình đó khi so sánh với câu hỏi 1? Giải thích!

Bây giờ chúng ta sẽ áp dụng cơ chế khác có tên là **lock**.
Tạo một lớp có tên là *ResourcesExploiterWithLock* kế thừa từ lớp *ResourcesExploiter.* Đầu tiên, cần phải import các lớp sau vào:

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
```

Khai báo biến lock như sau:

```
private ReentrantLock lock;
```

Viết phương thức khởi tạo, đừng quên đưa phương thức super vào để gọi phương thức khởi tạo của lớp cha (*ResourcesExploiter*).

```
public ResourcesExploiterWithLock(int n){
        super(n);
        lock = new ReentrantLock();
}
```

Ở phương thức *exploit()*, sử dụng phương thức *lock.tryLock* để block công việc tăng của biến rsc:

```
try{
                if(lock.tryLock(10, TimeUnit.SECONDS)){
                 setRsc(getRsc()+1);
        }
} catch (InterruptedException e) {
        e.printStackTrace();
}finally{
        //release lock
```

```
                lock.unlock();
        }
```

Tạo một lớp có tên *ThreadedWorkerWithLock* kế thừa từ lớp Thread.

Lớp này giống lớp *ThreadedWorkerWithoutSync* trừ việc kiểu của biến rExp là *ResourcesExploiterWithLock.*

Câu hỏi 3: Thay đổi đoạn code của chương trình chạy chính bằng cách thay thế kiểu của 3 thực thể worker1-3 thành *ThreadedWorkerWithLock.* Có khác nhau gì so với đầu ra của câu hỏi 1. Giải thích!

2. Lập trình song song với đoạn găng (ngôn ngữ

C). 2.1.Nội dung

Nội dung tương tự phần 1, chỉ có điều chúng ta sẽ tìm hiểu các kỹ thuật lập trình song song của ngôn ngữ C để giải quyết tương tranh trong việc vào sử dụng đoạn găng của các luồng.

2.2.Yêu cầu

2.2.1. Lý thuyết

- Lập trình song song với đoạn găng

2.2.2. Phần cứng

- Laptop/PC dùng Linux

2.2.3. Phần mềm

- gcc

2.3.Các bước thực hành

Hãy bắt đầu bằng một chương trình đa luồng đơn giản.
Tạo một file simple.c với nội dung như sau:

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

#include <unistd.h>
#include <sys/types.h>

#include <pthread.h>

int shared = 10;

void * fun(void * args){
```

```
 time_t start = time(NULL);
 time_t end = start+5; //run for 5 seconds

 YOUR-CODE-HERE

 return NULL;
}
```



```
int main(){
 pthread_t thread_id;

 pthread_create(&thread_id, NULL, fun, NULL);

 pthread_join(thread_id, NULL);

 printf("shared: %d\n", shared);

 return 0;
}
```

Câu hỏi 4: Hoàn thiện file trên (điền vào phần **YOUR-CODE-HERE**) với một vòng lặp để tăng biến *shared* lên một đơn vị trong vòng 5 giây.
(gợi ý: hàm time(NULL) sẽ trả về giá trị thời gian của hệ thống với đơn vị là giây).

Dịch và chạy chương trình đó bằng câu lệnh sau:

```
$gcc -pthread simple.c -o simple
$./simple
```

Đoạn chương trình trên khá đơn giản và chưa có sự tương tranh do chỉ có 1 luồng phụ tác động vào biến *shared*.

Bây giờ chúng ta sẽ phát triển chương trình với nhiều hơn 2 luồng phụ. Chương trình sẽ phải sử dụng phương thức Locking để quản lý tài nguyên chia sẻ. Khái niệm khóa tài nguyên là một lĩnh vực nghiên cứu rộng. Ý tưởng chính ở đây là khi một chương trình (hoặc luồng) vào sử dụng đoạn găng (critical section), chương trình (luồng) đó sẽ giữ *lock* trong khi sử dụng đoạn găng. Như vậy, tại 1 thời điểm chỉ có một tiến trình (luồng) được vào đoạn găng.

Bây giờ hãy thử viết 1 chương trình đa luồng không sử dụng phương thức *locking*.

Tạo 1 file without-lock.c để mô phỏng một dịch vụ đơn giản của ngân hàng.

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

#include <unistd.h>
#include <sys/types.h>
```

```
#include <pthread.h>

#define INIT_BALANCE 50
#define NUM_TRANS 100

int balance = INIT_BALANCE;


int credits = 0;
int debits = 0;
```



```
void * transactions(void * args){
 int i,v;

 for(i=0;i<NUM_TRANS;i++){

 //choose a random value
 srand(time(NULL));
 v = rand() % NUM_TRANS;

 //randomnly choose to credit or debit
 if( rand()% 2){
 //credit

 balance = balance + v;
 credits = credits + v;

 }else{
 //debit

 balance = balance - v;
 debits = debits + v;

 }

 }

 return 0;
}

int main(int argc, char * argv[]){

 int n_threads,i;
 pthread_t * threads;

 //error check
 if(argc < 2){
 fprintf(stderr, "ERROR: Require number of threads\n");
exit(1);
 }

 //convert string to int
 n_threads = atol(argv[1]);

 //error check
 if(n_threads <= 0){
 fprintf(stderr, "ERROR: Invalivd value for number of threads\n");
```

```
exit(1);
 }

 //allocate array of thread identifiers
 threads = calloc(n_threads, sizeof(pthread_t));

 //start all threads
 for(i=0;i<n_threads;i++){
 pthread_create(&threads[i], NULL, transactions, NULL);  }

 //wait for all threads finish its jobs
 for(i=0;i<n_threads;i++){
 pthread_join(threads[i], NULL);
 }
```



```
 printf("\tCredits:\t%d\n", credits);
 printf("\t Debits:\t%d\n\n", debits);
 printf("%d+%d-%d= \t%d\n", INIT_BALANCE,credits,debits,
INIT_BALANCE+credits-debits);
 printf("\t Balance:\t%d\n", balance);

 //free array
 free(threads);

 return 0;
}
```

Bây giờ hãy build và chạy chương trình. Thử với 5 luồng phụ.

```
$gcc –pthread without-lock.c –o without-lock
$./without-lock 5
```

Câu hỏi 5: Bây giờ hãy tăng giá trị số luồng và giá trị của số lần giao dịch NUM_TRANS sau mỗi lần chạy chương trình cho đến khi nào bạn thấy sự khác nhau giữa giá trị Balance (giá trị còn lại trong tài khoản) và *INIT_BALANCE+credits-debits.* Giải thích tại sao lại có sự khác nhau đó.

Để giải quyết vấn đề ở câu hỏi 5 ở trên, chúng ta phải sử dụng kỹ thuật đoạn găng để cho phép tại 1 thời điểm chỉ một luồng vào đoạn găng. Khi một đoạn găng được định danh, chúng ta sử dụng các biến chia sẻ để lock đoạn găng đó lại. Chỉ một luồng được giữ *lock*, vì thế chỉ có một luồng được vào đoạn găng tại 1 thời điểm

Đầu tiên chúng ta hãy áp dụng kỹ thuật Naive-Lock (lock thô sơ) bằng cách sử dụng biến lock như chương trình sau. Hãy tạo 1 file *naive-lock.c* với nội dung như sau:

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

#include <unistd.h>
#include <sys/types.h>
```

```
#include <pthread.h>

int lock = 0; //0 for unlocked, 1 for locked

int shared = 0; //shared variable

void * incrementer(void * args){
 int i;

 for(i=0;i<100;i++){

 //check lock
 while(lock > 0); //spin until unlocked

 lock = 1; //set lock

 shared++; //increment
```



```
 lock = 0; //unlock
 }

 return NULL;
}

int main(int argc, char * argv[]){
 pthread_t * threads;
 int n,i;

 if(argc < 2){
 fprintf(stderr, "ERROR: Invalid number of threads\n");
exit(1);
 }

 //convert argv[1] to a long
 if((n = atol(argv[1])) == 0){
 fprintf(stderr, "ERROR: Invalid number of threads\n");
exit(1);
 }

 //allocate array of pthread_t identifiers
 threads = calloc(n,sizeof(pthread_t));

 //create n threads
 for(i=0;i<n;i++){
 pthread_create(&threads[i], NULL, incrementer, NULL);  }

 //join all threads
 for(i=0;i<n;i++){
 pthread_join(threads[i], NULL);
 }

 //print shared value and result
 printf("Shared: %d\n",shared);
 printf("Expect: %d\n",n*100);

 return 0;
}
```

Câu hỏi 6: Hãy build và chạy chương trình này. Chạy lặp đi lặp lại đến bao giờ bạn thấy sự khác nhau giữa 2 giá trị *Shared* và *Expect*. Phân tích mã nguồn để hiểu vấn đề.

Bây giờ để giải quyết vấn đề trên của naive-lock, chúng ta sẽ sử dụng kỹ thuật có tên *mutex lock*. Một biến mutex không phải là biến chuẩn, thay vì thế nó đảm bảo các tác vụ sẽ được *atomic* (có nghĩa là việc nắm giữ lock sẽ không bị ngắt như naive lock).

Các bước để triển khai *mutex lock* sẽ được mô tả như sau:
- Đầu tiên khai báo biến mutex:

```
pthread_mutext_t mutex;
```

- Sau đó, khởi tạo biến mutex trước khi sử dụng:


```
pthread_mutex_init(&mutex, NULL);
```

- Bây giờ bạn có thể lock và unlock như sau

```
pthread_mutex_lock(&mutex);

/* code đoạn găng ở đây */

pthread_mutex_unlock(&mutex);
```

- Cuối cùng đừng quên hủy và giải phóng biến *mutex*:

```
pthread_mutex_destroy(&mutex);
```

Câu hỏi 7: Bây giờ hãy thay đổi đoạn code của file without-lock.c bằng cách triển khai cơ chế mutex lock như trên (bạn có thể tạo file mới và đặt tên khác đi như *mutex-lock-banking.c*). Chạy chương trình nhiều lần và đánh giá đầu ra. Nó có cải thiện gì hơn so với naive-lock?

Có 2 kỹ thuật để thực hiện khóa đoạn găng: **Coarse Locking** và **Fine Locking**. Coarse Locking sẽ khóa chương trình bằng cách sử dụng duy nhất 1 lock cho toàn bộ đoạn găng. Đó là điều mà bạn đã làm ở câu 7. Có thể thấy cách thức vận hành của Coarse Locking không hiệu quả. Chúng ta cần một cơ chế song song khác vì không phải tất cả các phần của đoạn găng cũng cần phải được đảm bảo cấm truy cập với nhau. Ví dụ, ở chương trình về dịch vụ ngân hàng ở trên, chúng ta làm việc với 2 biến credits và debits, nhưng mỗi luồng chỉ thực thi trên một biến, chứ không phải cả 2. Vì vậy sẽ hiệu quả hơn nếu chúng ta sử dụng một cơ chế lock khác được gọi là Fine Locking.

Bây giờ chúng ta sẽ cải thiện lại chương trình cho dịch vụ ngân hàng ở trên bằng cách sao chép và tạo file mới và đặt tên là *fine-locking-bank.c*
Thay vì dùng duy nhất 1 biến mutex, chúng ta sẽ tạo ra 3 biến:

```
pthread_mutex_t b_lock,c_lock,d_lock;
```

ở đó `b_lock` là để cho biến `balance`
`c_lock` là cho biến `credits`
`d_lock` cho biến `debits`

Ở trong vòng lặp `for(i=0;i<NUM_TRANS;i++),` bạn thêm vào câu lệnh sau cho mỗi lock:

```
pthread_mutex_lock(&b_lock);
 balance = balance + v;
 pthread_mutex_unlock(&b_lock);
```

Làm tương tự với *credits* và *debits*.

Câu hỏi 8: so sánh và đo đạt thời gian để chứng minh là Fine Locking sẽ nhanh hơn Coarse Locking.

Trong lúc sử dụng kỹ thuật Fine Locking, chú ý là rất dễ xảy ra deadlocks, có nghĩa là không luồng nào vào dùng đoạn găng được (chờ đợi lẫn nhau). Thử chạy chương trình sau để thấy rõ điều đó: (đặt tên file là *deadlocks-test.c*):

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

int a=0,b=0;
pthread_mutex_t lock_a, lock_b;

void * fun_1(void * arg){
 int i;
 for (i = 0 ; i< 10000 ; i++){

 pthread_mutex_lock(&lock_a); //lock a then b
 pthread_mutex_lock(&lock_b);

 //CRITICAL SECTION
 a++;
 b++;


 pthread_mutex_unlock(&lock_a);
 pthread_mutex_unlock(&lock_b);

 }

 return NULL;
}

void * fun_2(void * arg){
```

```
 int i;
 for (i = 0 ; i< 10000 ; i++){

 pthread_mutex_lock(&lock_b); //lock b then a
 pthread_mutex_lock(&lock_a);

 //CRITICAL SECTION
 a++;
 b++;


 pthread_mutex_unlock(&lock_b);
 pthread_mutex_unlock(&lock_a);

 }

 return NULL;
}

int main(){

 pthread_t thread_1,thread_2;
```



```
pthread_mutex_init(&lock_a, NULL);
 pthread_mutex_init(&lock_b, NULL);

 pthread_create(&thread_1, NULL, fun_1, NULL);
 pthread_create(&thread_2, NULL, fun_2, NULL);

 pthread_join(thread_1, NULL);
 pthread_join(thread_2, NULL);

 printf("\t a=%d b=%d \n", a,b);

 return 0;
}
```

Câu hỏi 9: chạy chương trình trên và bạn nhận thấy điều gì? Giải thích thông qua việc phân tích mã nguồn.