

BÀI THỰC HÀNH SỐ 4

CÁC HỆ THỐNG PHÂN TÁN VÀ ỨNG DỤNG

CHƯƠNG 4: ĐỒNG BỘ HÓA TRONG HỆ PHÂN TÁN

Họ và tên: Nguyễn Duy Khánh

Mã lớp: 157542

MSSV: 20225019

Mã học phần: IT4611

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

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);
```

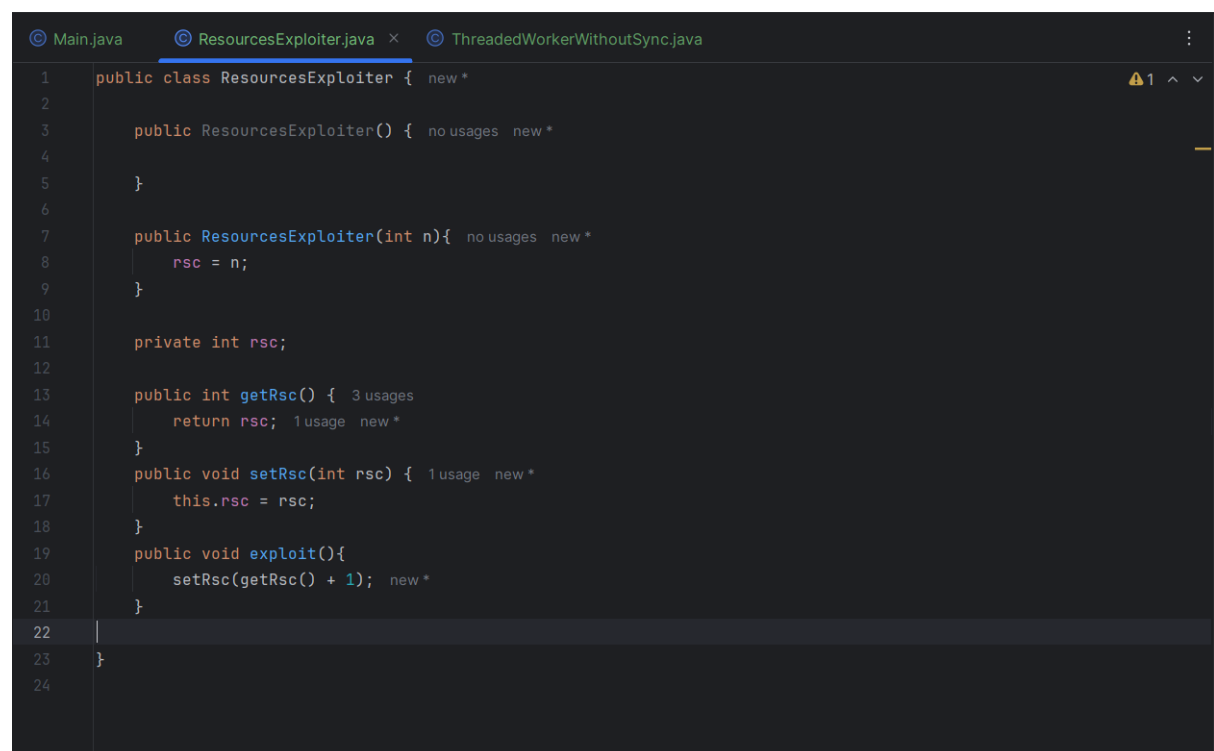
```
ThreadedWorkerWithoutSync worker3 = new  
ThreadedWorkerWithoutSync(resource);
```

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*.

Lớp *ResourcesExploiter.java*:



```
1 public class ResourcesExploiter { new *
2
3     public ResourcesExploiter() { no usages new *
4
5     }
6
7     public ResourcesExploiter(int n){ no usages new *
8         rsc = n;
9     }
10
11     private int rsc;
12
13     public int getRsc() { 3 usages
14         return rsc; 1 usage new *
15     }
16     public void setRsc(int rsc) { 1 usage new *
17         this.rsc = rsc;
18     }
19     public void exploit(){
20         setRsc(getRsc() + 1); new *
21     }
22
23 }
24
```

Lớp *ThreadedWorkerWithoutSync.java*:

```
© Main.java © ResourcesExploiter.java © ThreadedWorkerWithoutSync.java ×
1 public class ThreadedWorkerWithoutSync extends Thread{ 6 usages new *
2     private ResourcesExploiter rExp; 2 usages
3     public ThreadedWorkerWithoutSync(ResourcesExploiter rExp) { 3 usages new *
4         this.rExp = rExp;
5     }
6     @Override new *
7     public void run() {
8         for (int i = 0; i < 1000; i++) {
9             rExp.exploit();
10        }
11    }
12
13 }
14
```

Lớp Main.java:

```
© Main.java × © ResourcesExploiter.java © ThreadedWorkerWithoutSync.java
To Run code, press Shift F10 or click the ▶ icon in the gutter.
3 ▶ public class Main { new *
4 ▶     public static void main(String[] args) { new *
5         ResourcesExploiter resource = new ResourcesExploiter(0);
6
7         ThreadedWorkerWithoutSync worker1 = new ThreadedWorkerWithoutSync(resource);
8         ThreadedWorkerWithoutSync worker2 = new ThreadedWorkerWithoutSync(resource);
9         ThreadedWorkerWithoutSync worker3 = new ThreadedWorkerWithoutSync(resource);
10
11         worker1.start();
12         worker2.start();
13         worker3.start();
14
15         try {
16             worker1.join();
17             worker2.join();
18             worker3.join();
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22
23         System.out.println("Last value of rsc: " + resource.getRsc());
24     }
25 }
```

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

```
C:\Users\Lenovo\jdk\openjdk-22.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=53464:C:\Program Files\JetBrains\IntelliJ IDEA
Last value of rsc: 1960

Process finished with exit code 0
```

```
C:\Users\Lenovo\.jdk\openjdk-22.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=53472:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\bin" -Dfile.encoding=UTF-8
Last value of rsc: 2416

Process finished with exit code 0
|

C:\Users\Lenovo\.jdk\openjdk-22.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=53573:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\bin" -Dfile.encoding=UTF-8
Last value of rsc: 1638

Process finished with exit code 0
```

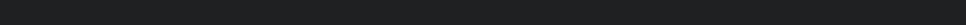
Khi chạy chương trình, mỗi lần sẽ in ra một kết quả khác nhau.

Hiện tượng này do các luồng cùng truy cập và thay đổi tài nguyên dùng chung (biến rsc) từ nhiều luồng mà không có cơ chế đồng bộ. Khi các luồng được thi hành, chúng sẽ truy cập vào biến rsc và tăng giá trị của nó lên một đơn vị. Tuy nhiên, do không có sự đồng bộ hóa trong quá trình này, các luồng có thể cùng ghi đè lên giá trị hiện tại của biến rsc, dẫn đến kết quả không đạt như mong đợi là 3000.

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!



Run Main x

C:\Users\Lenovo\jdk\openjdk-22.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=54363:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\bin" -Dfile.encoding=UTF-8

Last value of rsc: 3000

Process finished with exit code 0

Các luồng thực hiện đúng theo thứ tự và luồng sau chỉ thực hiện khi luồng trước đó đã kết thúc. Các kết quả được in ra tăng liên tục theo thời gian. Kết quả cuối cùng là 3000 và đúng với kết quả của thread chạy sau cùng. Khi thay đổi kiểu của các thực thể `worker1-3` thành **ThreadedWorkerWithSync**, lớp này sử dụng `synchronized` để đồng bộ hóa việc truy cập vào biến chung `rExp`. Khi một luồng sở hữu khóa `lock` cho biến `rExp`, các luồng khác phải chờ đợi trước khi được phép tiếp tục truy cập vào biến này. Do đó, các luồng không ghi đè lên giá trị của nhau và trả ra kết quả chính xác là 3000. Các giá trị của biến `rExp` được cập nhật đúng số lần bởi ba luồng mà không có sự xung đột giữa các luồng. Việc sử dụng `synchronized` trong quá trình truy cập và sửa đổi giá trị của các biến chung giữa các luồng sẽ đảm bảo tính đúng đắn của kết quả và tránh xảy ra lỗi khi thực hiện các tác vụ đồng thời.

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!

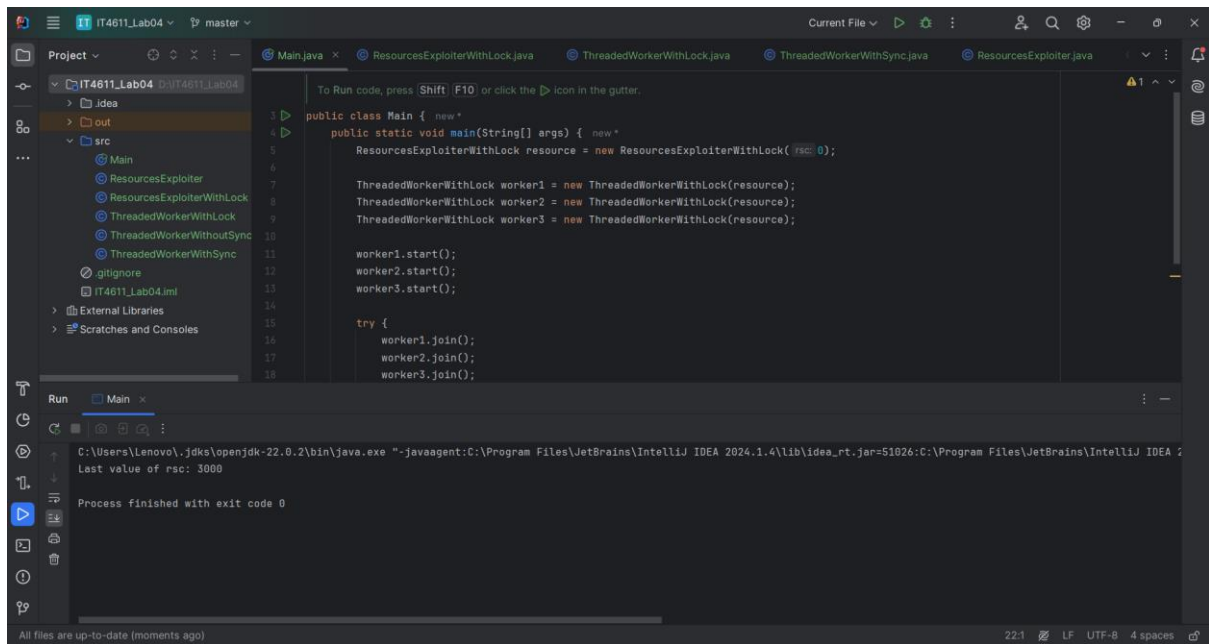
Lớp ResourcesExploiterWithLock.java:

```
© Main.java © ResourcesExploiterWithLock.java × © ThreadedWorkerWithLock.java © ThreadedWorkerWithSync.java © ResourcesExploiter.java
1 import java.util.concurrent.TimeUnit;
2 import java.util.concurrent.locks.ReentrantLock;
3
4 public class ResourcesExploiterWithLock extends ResourcesExploiter { no usages new *
5     private ReentrantLock lock; 3 usages
6     public ResourcesExploiterWithLock(int rsc){ no usages new *
7         super(rsc);
8         lock = new ReentrantLock();
9     }
10
11     @Override 2 usages new *
12     public void exploit(){
13         try {
14             if(lock.tryLock(10, TimeUnit.SECONDS)){
15                 setRsc(getRsc()+1);
16             }
17         } catch (InterruptedException ex){
18             ex.printStackTrace();
19         } finally {
20             lock.unlock();
21         }
22     }
23
24 }
```

Lớp ThreadedWorkerWithLock.java:

```
© Main.java © ResourcesExploiterWithLock.java × © ThreadedWorkerWithLock.java © ThreadedWorkerWithSync.java © ResourcesExploiter.java
1 public class ThreadedWorkerWithLock extends Thread { no usages new *
2
3     private ResourcesExploiterWithLock rExp; 2 usages
4     public ThreadedWorkerWithLock(ResourcesExploiterWithLock rExp) { no usages new *
5         this.rExp = rExp;
6     }
7
8     @Override new *
9     public void run() {
10         for (int i = 0; i < 1000; ++i) {
11             rExp.exploit();
12         }
13     }
14 }
```

Kết quả đầu ra của chương trình trên là:



Kết quả luôn luôn ra 3000 trong mọi tình huống. Điều này là vì khi thay đổi kiểu của các thực thể worker1-3 thành **ThreadedWorkerWithLock**, ta sử dụng lớp **ReentrantLock** để đồng bộ hóa việc truy cập vào biến chung rsc. Mỗi luồng muốn truy cập và thay đổi giá trị của biến rsc phải trước tiên sở hữu khóa lock tương tự như việc sử dụng synchronized. Sau khi sở hữu khóa lock, luồng được phép thực hiện truy cập và sửa đổi giá trị của biến rsc. Khi hoàn tất, luồng giải phóng khóa lock để cho phép các luồng khác tiếp tục truy cập vào biến chung này. Sử dụng ReentrantLock đã giúp đảm bảo tính nhất quán của biến rsc trong môi trường đa luồng, tránh được các vấn đề về điều kiện tranh chấp (race conditions) và đảm bảo kết quả chính xác.

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

Hãy bắt đầu bằng một chương trình đa luồng đơn giản.

Tạo một file simple.c với nội dung như sau

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).

A screenshot of a VS Code editor running in an Oracle VM VirtualBox window titled 'Ubuntu 24.04.2 LTS [Running] - Oracle VirtualBox'. The editor is open to a file named 'simple.c'. The code defines a function 'fun' that takes no arguments and returns NULL. Inside 'fun', it sets 'start' to 'time(NULL)' and 'end' to 'start + 5' with a comment '// run for 5 seconds'. A while loop 'while (time(NULL) < end) {' contains a 'continue;' statement. After the loop, 'shared' is incremented and NULL is returned. The 'main' function declares a 'pthread_t' variable, creates a thread with 'pthread_create', joins it with 'pthread_join', and prints 'shared: %d\n' before returning 0. The terminal at the bottom shows the compilation command 'gcc -pthread simple.c -o simple' and the execution output 'shared: 11'.

// YOUR CODE FILE:

```
while (time(NULL) < end) {  
    continue;  
}  
shared++;
```

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.

Kết quả thu được khi build và chạy thử chương trình với 5 luồng phụ:

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

```
Ubuntu 24.04.2 LTS [Running] - Oracle VirtualBox
File Machine View Input Devices Help
May 13 02:20
EXPLORER
UNTITLED (WORKSPACE)
IT4611_Lab04
  .vscode
  simple
  without-lock
  without-lock.c
C without-lock.c
C simple.c
C without-lock.c
Welcome
C simple.c
C without-lock.c
IT4611_Lab04 > C without-lock.c > transactions(void *)
37 int main(int argc, char *argv[]){
63     printf("\t Debits:\t%d\n", debits);
64     printf("%d+%d-%d= \t%d\n", INIT_BALANCE,credits,debits,
65     INIT_BALANCE+credits-debits);
66     printf("\t Balance:\t%d\n", balance);
67     //free array
68     free(threads);
69     return 0;
70 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
See 'snap info <snapname>' for additional versions.
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ^C
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ gcc -pthread simple.c -o simpl
e
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./simple
shared: 11
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ gcc -pthread without-lock.c -o
without-lock
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./without-lock 5
Credits: 0
Debits: 21000

50+0-21000= -20950
Balance: -20950
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$
```

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.

Khi tăng lên 10 luồng, chưa có hiện tượng gì xảy ra, chương trình vẫn chạy đúng.


```
IT4611_Lab04 > C without-lock.c > transactions(void *)
37 int main(int argc, char *argv[]){
63     printf("\t Debits:\t%d\n\n", debits);
64     printf("%d+%d-%d= \t%d\n", INIT_BALANCE,credits,debits,
65     INIT_BALANCE+credits-debits);
66     printf("\t Balance:\t%d\n", balance);
67     //free array
68     free(threads);
69     return 0;
70 }
```

```
Credits:      0
Debits:     21000

50+0-21000=   -20950
Balance:     -20950
• kxanhskii04@kxanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ gcc -pthread without-lock.c -o
without-lock
• kxanhskii04@kxanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./without-lock 10
Credits:     74000
Debits:      0

50+74000-0=   74050
Balance:     74050
• kxanhskii04@kxanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$
```

Khi tăng lên 100000 luồng, có sự khác biệt giữa giá trị Balance (giá trị còn lại trong tài khoản) và $INIT_BALANCE + credits - debits$

```
• kxanhskii04@kxanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./without-lock 100000
Credits:     273953554
Debits:     209509648

50+273953554-209509648=   64443956
Balance:     64398056
• kxanhskii04@kxanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$
```

Khi chạy chương trình without-lock.c với số lượng luồng và số lượng giao dịch tăng lên, sẽ xảy ra sự khác biệt giữa giá trị balance và $INIT_BALANCE + credits - debits$, nguyên nhân là do hiện tượng race condition trong môi trường đa luồng khi không có cơ chế đồng bộ hóa. Nhiều luồng có thể cùng truy cập và sửa đổi các biến balance, credits, debits cùng một lúc, gây ra xung đột và dẫn đến kết quả sai.

Ví dụ, khi hai luồng cùng muốn cộng vào balance, chúng có thể đọc giá trị hiện tại của balance, thực hiện phép cộng và ghi lại giá trị mới. Tuy nhiên, do không có bất kỳ cơ chế đồng bộ hóa nào, có thể xảy ra tình huống cạnh tranh giữa hai luồng, dẫn đến việc một luồng ghi đè lên giá trị được cập nhật bởi luồng khác, làm mất mát dữ liệu.

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

thể 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:

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

```
Shared: 974
Expect: 1000
```

Khi chạy chương trình *naive-lock.c* với nhiều luồng, đôi khi kết quả *Shared* khác *Expect* do biến lock không phải là một cơ chế đồng bộ an toàn. Có một số thời điểm nào đó mà khi *lock = 0*, tức là đang unlock, lúc đó có một số thread đồng thời kiểm tra điều kiện của vòng lặp while và đều có thể truy cập được vào resource.

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_mutex_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*?

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash + v [ ] [ ] ... ^ x

Balance: 54284
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./mutex-lock-banking 14214
Credits: 35062244
Debits: 35093156

50 + 35062244 - 35093156 = -30862
Balance: -30862
khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./mutex-lock-banking 44252
Credits: 111005463
Debits: 108468266

50 + 111005463 - 108468266 = 2537247
Balance: 2537247
o khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$
```

Một biến mutex đảm bảo rằng việc nắm giữ lock sẽ không bị ngắt như naive lock. Khi sử dụng mutex, mỗi luồng sẽ phải thực hiện hai bước quan trọng: khóa mutex khi thực hiện việc tăng giá trị shared và mở khóa mutex sau khi hoàn thành. Bằng cách này, chỉ có một luồng được phép khóa mutex và thực hiện phép tăng giá trị shared tại một thời điểm. Các luồng khác sẽ chờ đến khi mutex được mở khóa trước khi tiếp tục thực hiện việc tăng giá trị. Khi áp dụng mutex lock vào chương trình without-lock.c, kết quả luôn luôn đúng và không bị sai lệch như khi sử dụng naive lock. Vì vậy, mutex lock là một cơ chế đồng bộ an toàn và hiệu quả 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*.

```
• khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./mutex-lock-banking 300
Time consuming: 0.151784      Credits:      1320000
Debits:      0

50+1320000-0=      1320050
Balance:      1320050

• khanhskii04@khanhskii04-VirtualBox:~/Desktop/IT4611_Lab04$ ./fine-lock-banking 300
Time consuming: 0.027484      Credits:      1230000
Debits:      0

50+1230000-0=      1230050
Balance:      1230050
```

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

Fine locking tạo ra đoạn găng riêng cho mỗi tài nguyên. Coarse locking tạo ra một đoạn găng duy nhất cho tất cả tài nguyên.

Với Coarse Locking, dù đảm bảo an toàn dữ liệu nhưng gây tắc nghẽn luồng vì tất cả luồng đều phải chờ cùng một lock để truy cập bất kỳ phần nào của đoạn gang trong khi đó Fine Locking chỉ khóa đúng phần cần thiết, nếu một luồng đang cập nhật credits, luồng khác vẫn có thể cập nhật debits.

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).

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

```
[Running] cd "/home/khanhskii04/Desktop/IT4611_Lab04/" && gcc deadlocks-test.c -o deadlocks-test && "/home/khanhskii04/Desktop/IT4611_Lab04/"deadlocks-test
a=20000 b=20000

[Running] cd "/home/khanhskii04/Desktop/IT4611_Lab04/" && gcc deadlocks-test.c -o deadlocks-test && "/home/khanhskii04/Desktop/IT4611_Lab04/"deadlocks-test

[Done] exited with code=null in 93.679 seconds
```

Sau hơn 90s, chương trình bị treo, không in ra màn hình bất cứ gì (phải dùng Ctrl-C để thoát).

Chương trình trên gặp vấn đề Deadlock vì có một xung đột khóa trong các luồng. Trong fun_1, trước khi tăng giá trị a và b, luồng này khóa lock_a rồi khóa lock_b. Trong fun_2, luồng này khóa lock_b rồi khóa lock_a. Khi cả hai luồng đạt đến điểm này, mỗi luồng đang giữ một khóa mutex và đang chờ lấy khóa mutex khác dẫn đến tình trạng kẹt khi không có luồng nào có thể tiếp tục. Do đó, chương trình sẽ không bao giờ hoàn thành và bị treo (stuck) vì deadlock.