DISTRIBUTED SYSTEMS – TRAN HAI ANH

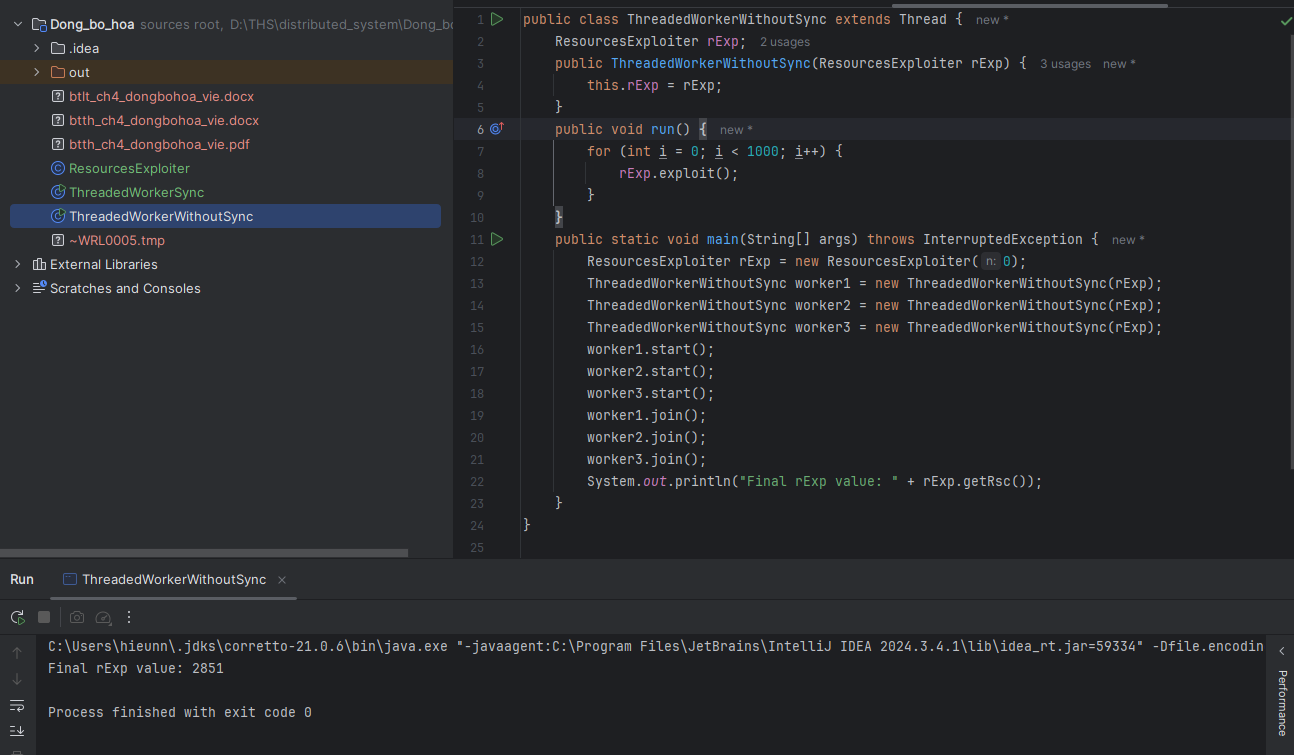
**BÀI THỰC HÀNH**

**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.**

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



Chạy vài lần kết quả trả ra đều < 3000

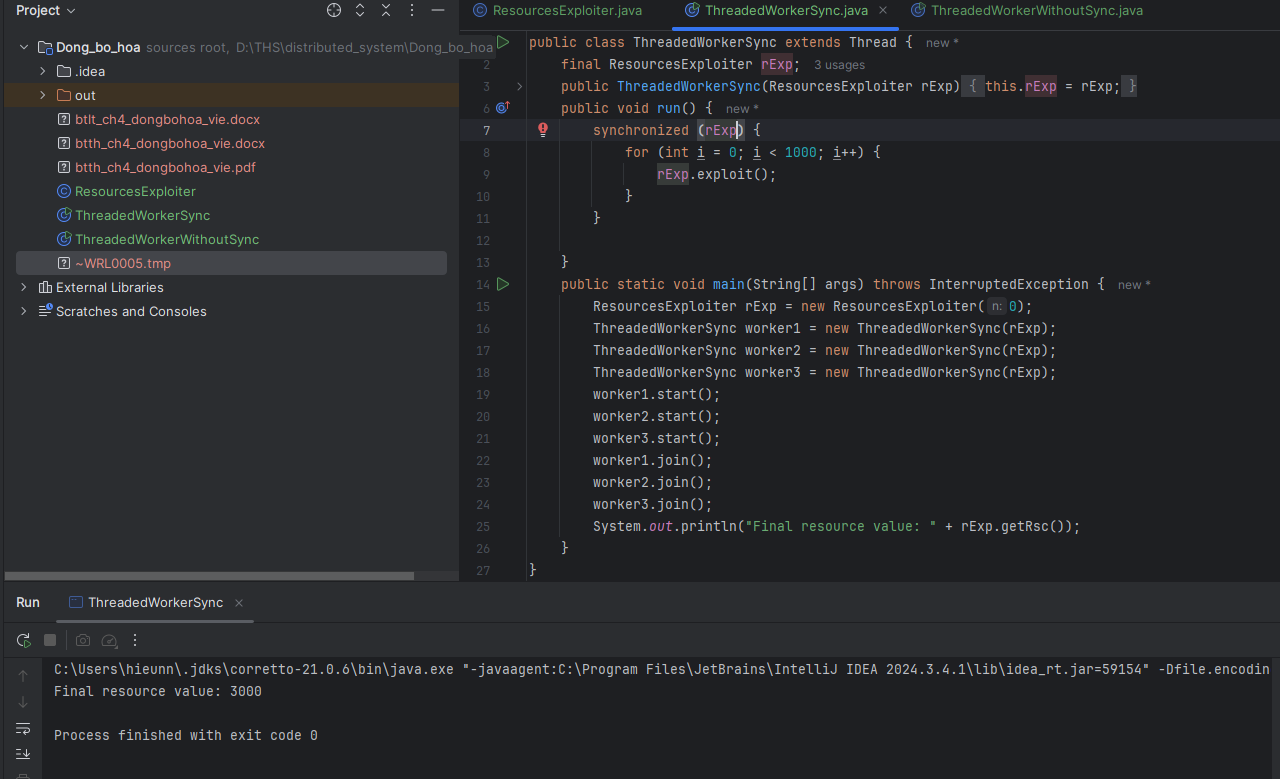
Logic xử lý của các hàm exploit là:

* Bước 1: Lấy giá trị biến dùng chung **rsc** trong cùng một Object ResourceExploiter.
* Bước 2: Cộng giá trị biến **rsc** thêm 1
* Bước 3: Trả về kết quả **rsc**

Các worker1, worker2, worker3 xử lý song song.

Trong quá trình lấy biến toàn cục, sẽ có nhiều thời điểm các worker thực thi Bước 1 cùng một lúc, nên kết quả sẽ bị ghi đè lên nhau.

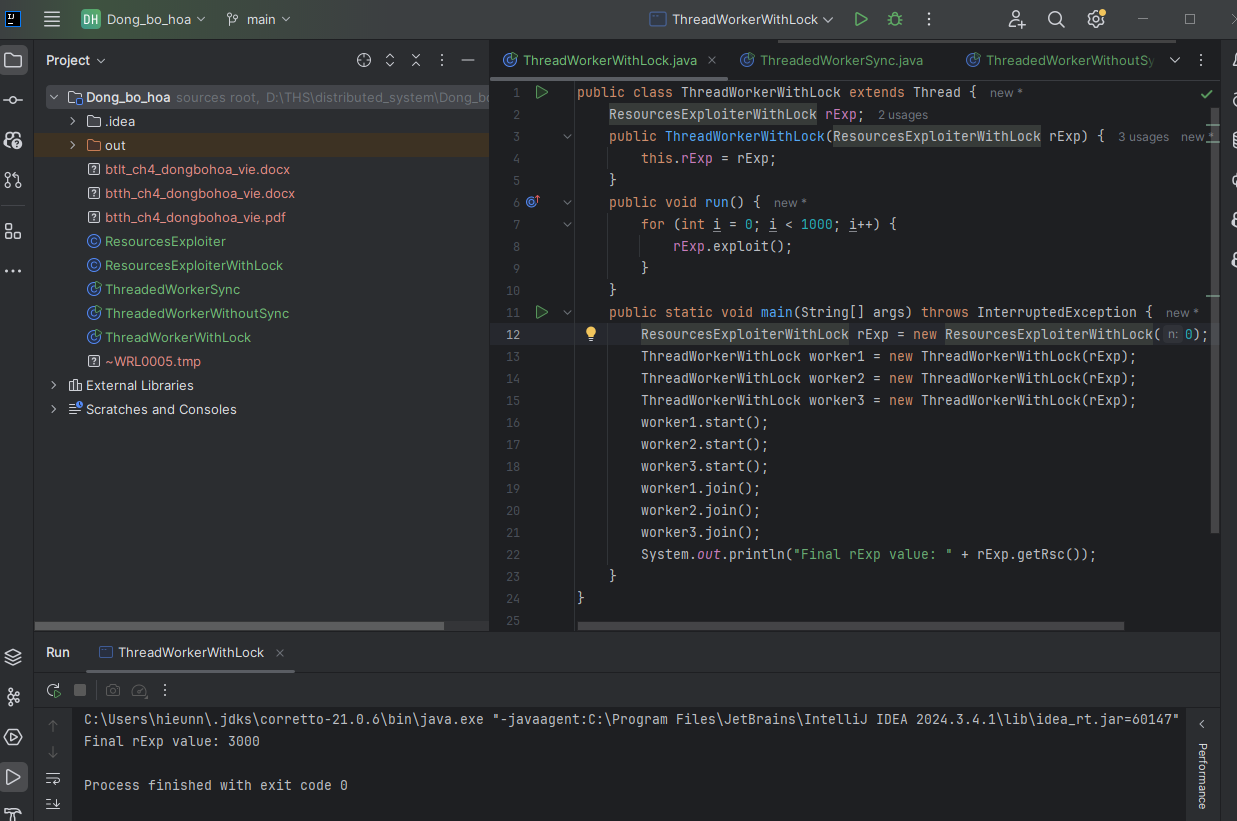
|  |
| --- |
| Câu hỏi 2: Thay đổi đoạn mã trong chương trình chạy (phương thức *main*), thay đổi kiểu của 3 thực thể worker1-3 thành *ThreadedWorkerWithSync.* Bạn nhận thấy sự thay đổi gì ở đầu ra khi chạy chương trình đó khi so sánh với câu hỏi 1? Giải thích! |



Khi sử dụng synchronized trên thực thể rExp kết quả trả về 3000.

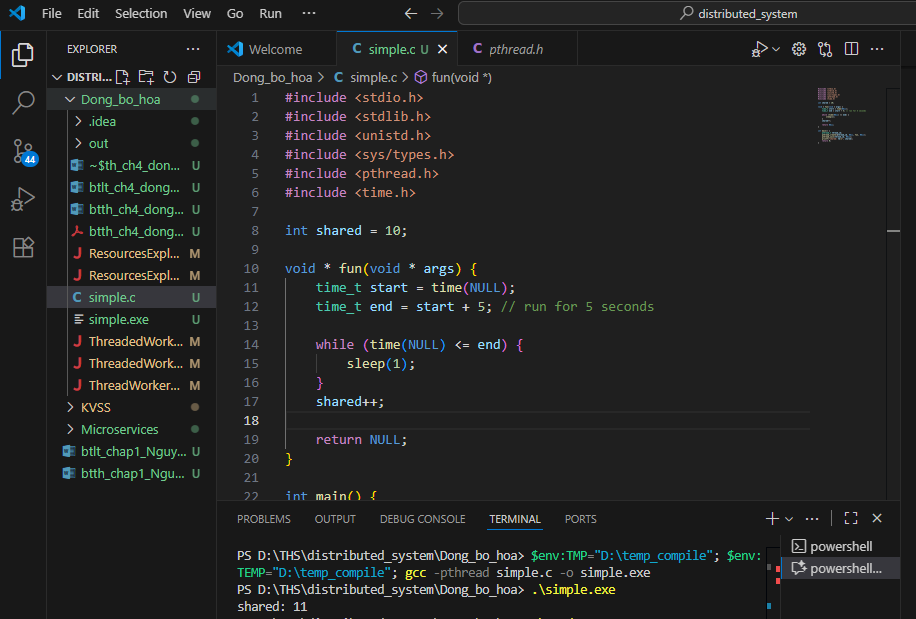
Giải thích: các Thread khi gọi hàm exploit() trong thực thể rExp được xử lý một cách tuần tự.

|  |
| --- |
| 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).**

|  |
| --- |
| 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;

5

DISTRIBUTED SYSTEMS – TRAN HAI ANH

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);

}

6

DISTRIBUTED SYSTEMS – TRAN HAI ANH

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

7

DISTRIBUTED SYSTEMS – TRAN HAI ANH

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:

8

DISTRIBUTED SYSTEMS – TRAN HAI ANH

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*.

9

DISTRIBUTED SYSTEMS – TRAN HAI ANH

|  |
| --- |
| 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;

10

DISTRIBUTED SYSTEMS – TRAN HAI ANH

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. |

11