DISTRIBUTED SYSTEMS – TRAN HAI ANH

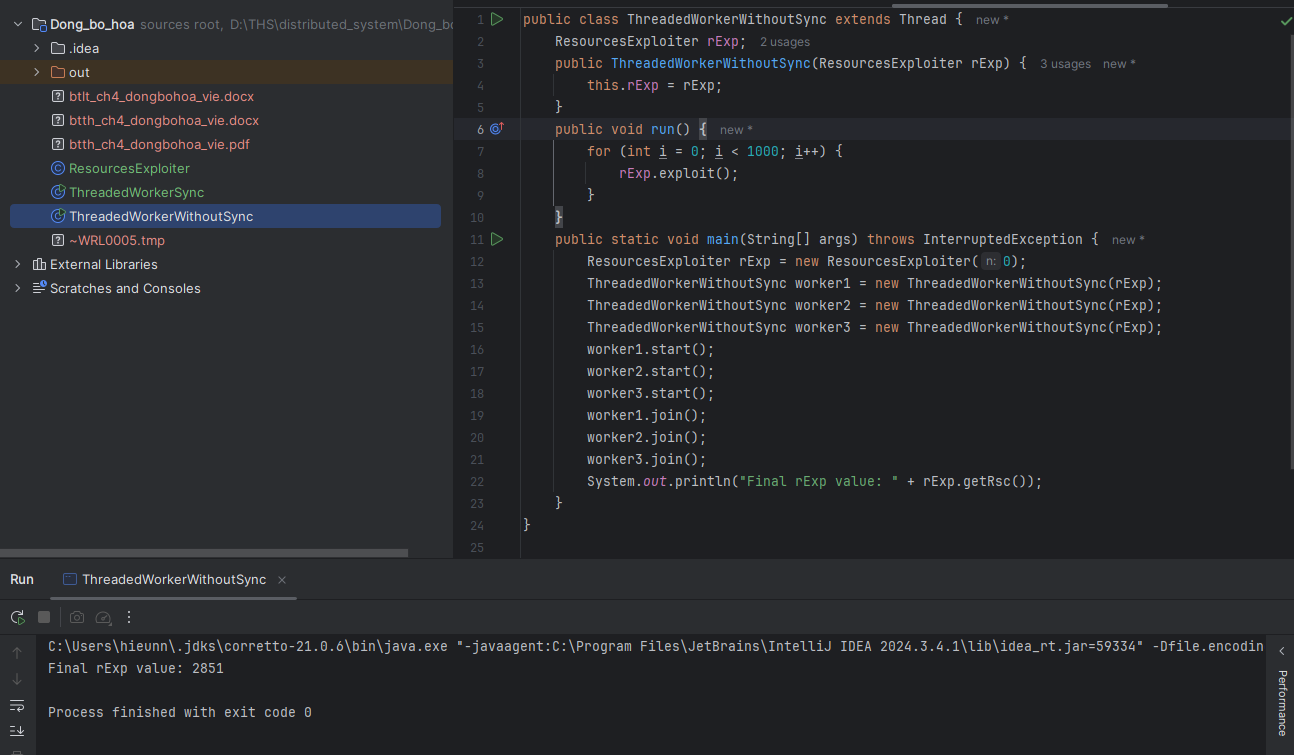
**BÀI THỰC HÀNH**

**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

Sự khác biệt này xuất hiện do Race Condition - một vấn đề nghiêm trọng trong lập trình đa luồng khi không có cơ chế đồng bộ hóa (synchronization).

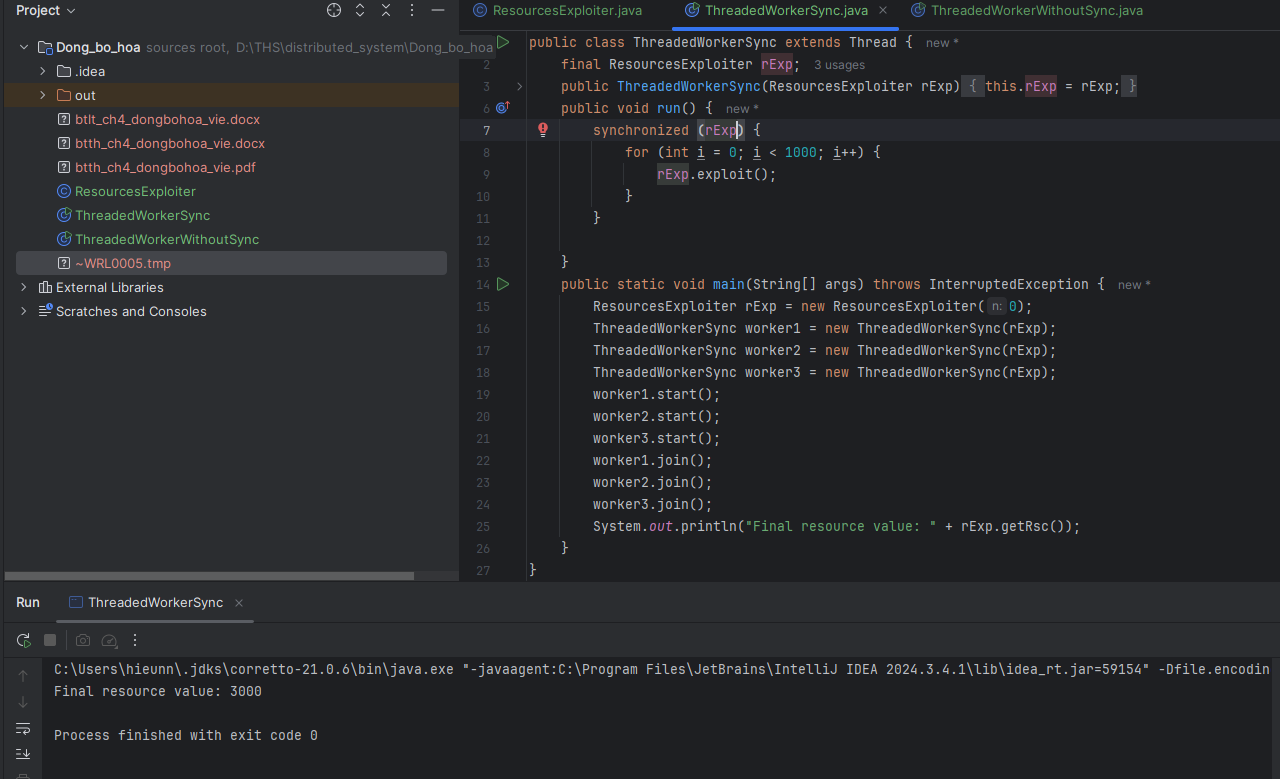
Logic xử lý của các hàm exploit là:

* 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.

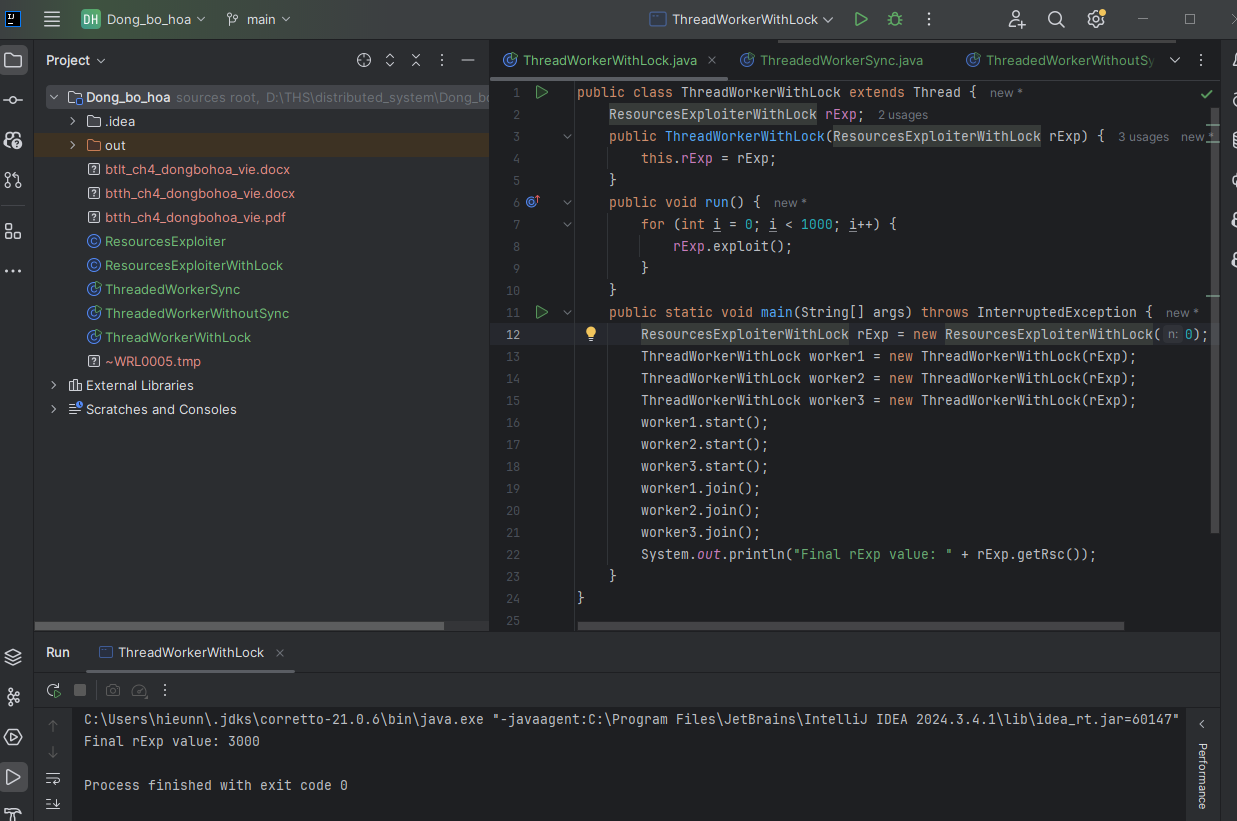
|  |
| --- |
| Câu hỏi 2: Thay đổi đoạn mã trong chương trình chạy (phương thức *main*), thay đổi kiểu của 3 thực thể worker1-3 thành *ThreadedWorkerWithSync.* Bạn nhận thấy sự thay đổi gì ở đầu ra khi chạy chương trình đó khi so sánh với câu hỏi 1? Giải thích! |



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.

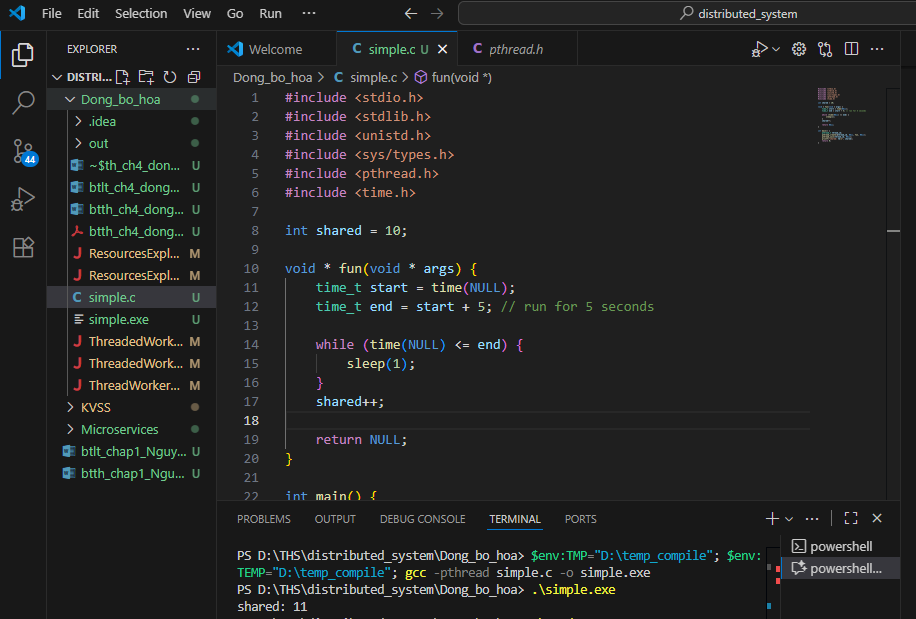
|  |
| --- |
| Câu hỏi 3: Thay đổi đoạn code của chương trình chạy chính bằng cách thay thế kiểu của 3 thực thể worker1-3 thành *ThreadedWorkerWithLock.* Có khác nhau gì so với đầu ra của câu hỏi 1. Giải thích! |



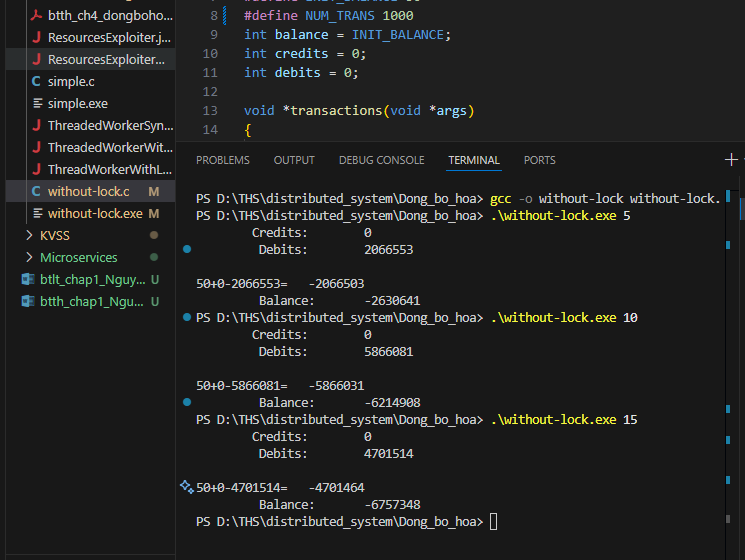
Tương tự với câu 2, các Thread khi gọi hàm exploit() đã được xử lý tuần tự

**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). |



|  |
| --- |
| 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. |



Tăng NUM\_TRANS lên 1000

|  |  |  |  |
| --- | --- | --- | --- |
| Thread num | Expected Balance | Actual Balance | Diff |
| 5 | -2,066,503 | -2,630,641 | -564,138 |
| 10 | -5,866,031 | -6,214,908 | -348,877 |
| 15 | -4,701,464 | -6,757,348 | -2,055,884 |

Giải thích:

Tương tự với câu hỏi 1. Trường hợp này cũng xuất hiên tình trạng Race Condition

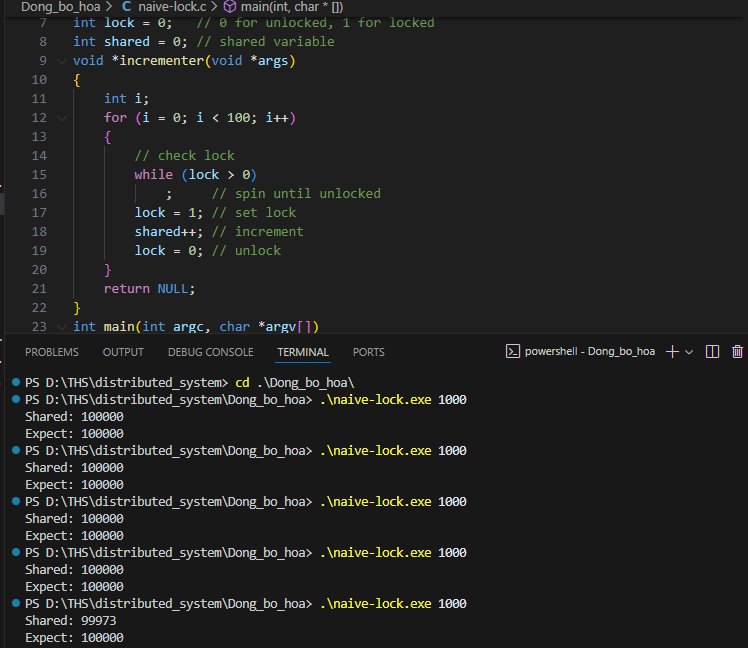
Logic xử lý của các hàm transactions là:

* Bước 1: Đọc giá trị hiện tại của balance
* Bước 2: Tính toán (balance + v hoặc balance - v)
* Bước 3: Ghi kết quả trở lại balance

Các worker1, worker2, worker3 xử lý song song.

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

|  |
| --- |
| 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. |

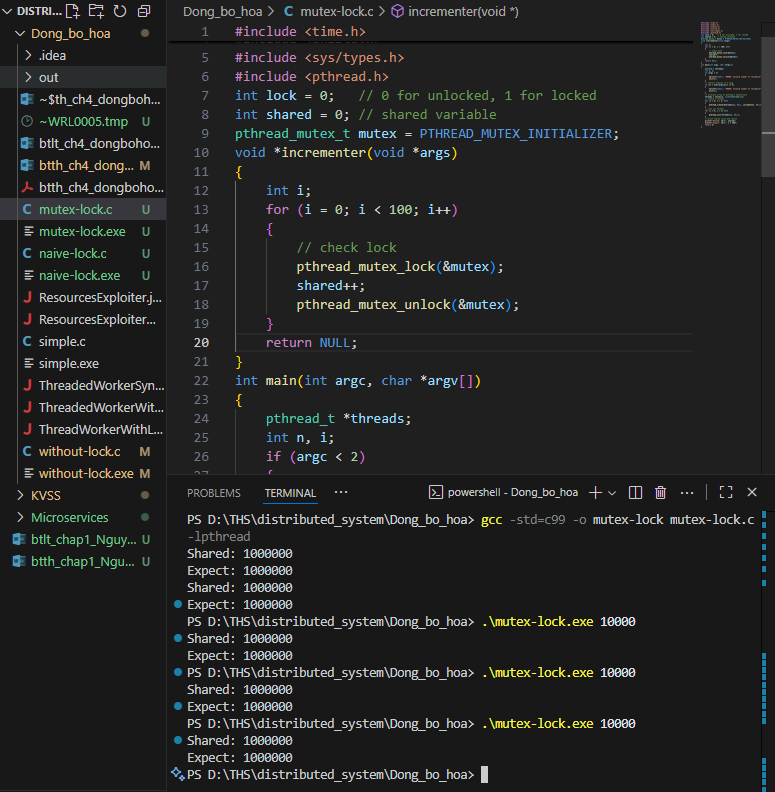


Đoạn code có sự khác biệt giữa Shared 99973 và Expect: 100000

Giải thích:

* Trong khoảng thời gian nhỏ giữa lock=1; và lock=0; nhiều Thread có thể lấy ra giá trị lock=0, ghi đè biến shared và ghi đè lock=1;
* Việc check-then-set với naive-lock không phải là atomic operation.

|  |
| --- |
| 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? |



Sau khi sử dụng mutex-lock đã thấy Shared và Expect không bị lệch nhau

Sử dụng fine-locking, Expected balance và Actual Balance đã khớp nhau

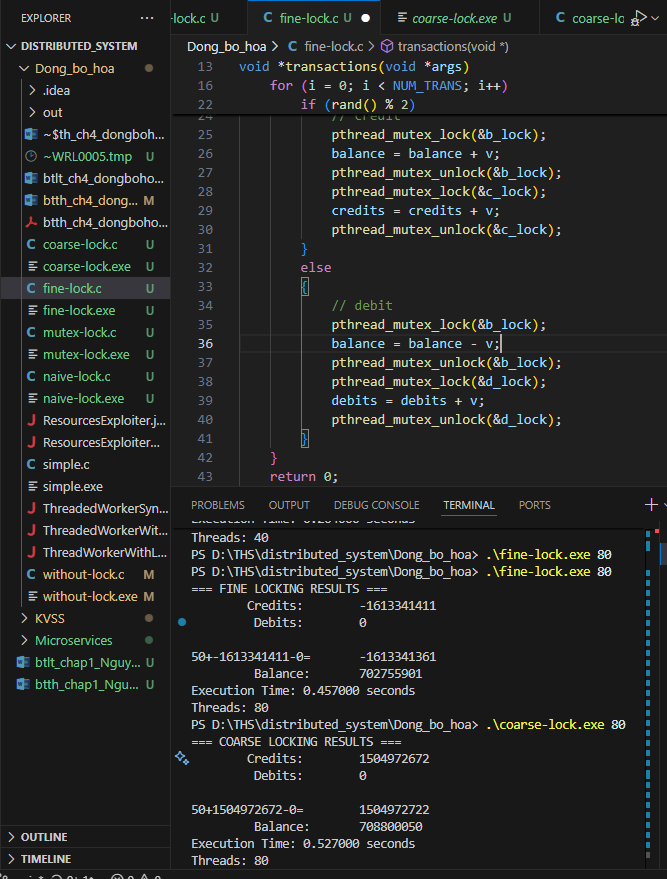
9

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. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test | Threads | Fine Locking (giây) | Coarse Locking (giây) | Tỉ lệ (Coarse/Fine) | Winner |
| 1 | 10 | 0.062 | 0.067 | 1.08x | Fine |
| 2 | 20 | 0.118 | 0.132 | 1.12x | Fine |
| 3 | 40 | 0.240 | 0.264 | 1.10x | Fine |
| 4 | 80 | 0.457 | 0.527 | 1.15x | Fine |

Hiệu suất cải thiện Fine Locking: 8-15% so với 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