



VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY

IT007.P11.CTTN - HĐH

LÀM VIỆC VỚI TIỂU TRÌNH VÀ ĐỒNG BỘ HÓA TIỂU TRÌNH

Sinh Viên :
Nguyễn Văn Minh

Giảng viên :
Vũ Đức Lung
Thân Thế Tùng

Ngày 20 tháng 11 năm 2024



Mục lục

1	TỔNG QUAN	3
1.1	BÀI TẬP THỰC HÀNH	3
1.2	BÀI TẬP ÔN TẬP	3
2	BÀI TẬP THỰC HÀNH	4
2.1	Bài 1	4
2.1.1	Đề bài	4
2.1.2	Cách làm	4
2.1.3	Code	4
2.1.4	Kết quả	5
2.1.5	Giải thích kết quả	6
2.2	Bài 2	6
2.2.1	Đề bài	6
2.2.2	Cách làm	6
2.2.3	Code	7
2.2.4	Kết quả	7
2.2.5	Giải thích kết quả	8
2.3	Bài 3	8
2.3.1	Đề bài	8
2.3.2	Cách làm	8
2.3.3	Code	8
2.3.4	Kết quả	8
2.3.5	Giải thích kết quả	8
2.4	Bài 4	8
2.4.1	Đề bài	8
2.4.2	Cách làm	8
2.4.3	Code	9
2.4.4	Kết quả	9
2.4.5	Giải thích kết quả	9
3	BÀI TẬP ÔN TẬP	10
3.1	Bài 1	10
3.1.1	Đề bài	10
3.1.2	Cách làm	10
3.1.3	Code	11
3.1.4	Kết quả	11



3.1.5	Giải thích kết quả	11
4	Source code	12

Chương 1

TỔNG QUAN

CHECKLIST (Đánh dấu x khi hoàn thành)

Lưu ý mỗi câu phải làm đủ 3 yêu cầu

1.1 BÀI TẬP THỰC HÀNH

	BT 1	BT 2	BT3
Trình bày cách làm	X	X	X
Chụp hình minh chứng	X	X	X
Giải thích kết quả	X	X	X

1.2 BÀI TẬP ÔN TẬP

	BT 1
Trình bày cách làm	X
Chụp hình minh chứng	X
Giải thích kết quả	X

Tự chấm điểm: 10

Lưu ý: Xuất báo cáo theo định dạng PDF, đặt tên theo cú pháp:

<MSSV>_LABx.pdf

Chương 2

BÀI TẬP THỰC HÀNH

2.1 Bài 1

2.1.1 Đề bài

Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: `sells <= products <= sells + [4 số cuối của MSSV]`

2.1.2 Cách làm

Để bảo vệ các biến `sells` và `products` khỏi việc bị truy cập đồng thời bởi nhiều *thread*, `mutex` được sử dụng để đồng bộ hóa. Trước khi một *thread* truy cập vào vùng mã quan trọng (*critical section*), nó sẽ gọi `pthread_mutex_lock` để khóa `mutex`, đảm bảo chỉ mình nó có quyền thao tác. Sau khi hoàn thành, *thread* sẽ gọi `pthread_mutex_unlock` để mở khóa, cho phép các *thread* khác tiếp tục. Cơ chế này đảm bảo dữ liệu chia sẻ luôn nhất quán và tránh xung đột.

2.1.3 Code

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <unistd.h>
6 #include <stdbool.h>
7
8 int products = 0;
9 int sells = 0;
10
11 sem_t sem;
12 pthread_mutex_t mutex;
13
14 void* producer(void* arg) {
15     while (true) {
16         sem_wait(&sem);
17         pthread_mutex_lock(&mutex);
```



```
18     if (sells < products) {
19         sells++;
20     }
21     pthread_mutex_unlock(&mutex);
22 }
23 }
24
25 void* consumer(void* arg) {
26     while (true) {
27         pthread_mutex_lock(&mutex);
28         if (products < sells + 1421) { //4 so cuoi MSSV la 1421
29             products++;
30         }
31         pthread_mutex_unlock(&mutex);
32         sem_post(&sem);
33     }
34 }
35
36 int main() {
37     pthread_t producer_thread, consumer_thread;
38     pthread_mutex_init(&mutex, NULL);
39     sem_init(&sem, 0, 0);
40
41     pthread_create(&producer_thread, NULL, producer, NULL);
42     pthread_create(&consumer_thread, NULL, consumer, NULL);
43
44     for (int i = 0; i < 5; i++) {
45         printf("products is %d and sells is %d\n", products, sells);
46         sleep(0.5);
47     }
48     sem_destroy(&sem);
49     pthread_mutex_destroy(&mutex);
50
51     return 0;
52 }
```

2.1.4 Kết quả

```
1 products is 0 and sells is 0
2 products is 10614 and sells is 10005
3 products is 20305 and sells is 20131
4 products is 30481 and sells is 30185
5 products is 41671 and sells is 40884
6 -----
7 products is 0 and sells is 0
8 products is 10194 and sells is 8773
9 products is 20016 and sells is 18978
10 products is 30990 and sells is 30990
11 products is 42059 and sells is 41902
```



```
12 -----
13 products is 0 and sells is 0
14 products is 9374 and sells is 8349
15 products is 22333 and sells is 20912
16 products is 34788 and sells is 33949
17 products is 48967 and sells is 48940
```

2.1.5 Giải thích kết quả

- Consumer luôn tăng products trước, và semaphore đảm bảo chỉ khi có sản phẩm thì Producer mới được phép bán.
- Products lúc nào cũng lớn hơn sells và nhỏ hơn sells + 1418
- Chênh lệch giữa products và sells phụ thuộc vào tốc độ xử lý của các luồng.
- Mutex và semaphore hoạt động đúng, nhưng sự chênh lệch tốc độ là yếu tố tự nhiên trong hệ thống đa luồng.

2.2 Bài 2

2.2.1 Đề bài

Cho một mảng *a* được khai báo như một mảng số nguyên có thể chứa *n* phần tử, *a* được khai báo như một biến toàn cục. Viết chương trình bao gồm 2 *thread* chạy song song:

- Một *Thread* làm nhiệm vụ sinh ra một số nguyên ngẫu nhiên sau đó bỏ vào *a*. Sau đó đếm và xuất ra số phần tử của *a* có được ngay sau khi thêm vào.
- *Thread* còn lại lấy ra một phần tử trong *a* (phần tử bất kỳ, phụ thuộc vào người lập trình). Sau đó đếm và xuất ra số phần tử của *a* có được ngay sau khi lấy ra, nếu không có phần tử nào trong *a* thì xuất ra màn hình “Nothing in array *a*”.

Chạy thử và tìm ra lỗi khi chạy chương trình trên khi chưa được đồng bộ. Thực hiện đồng bộ hóa với *semaphore*.

2.2.2 Cách làm

- Khởi tạo một biến *semaphore* với giá trị ban đầu là 0, sử dụng để quản lý cả *Producer* và *Consumer*. Trước khi một *thread* truy cập vào *critical section*, nó sẽ gọi hàm *wait* của *semaphore*. Tại thời điểm này, chỉ một *thread* được phép thực thi trong vùng *critical section*, vì giá trị của *semaphore* trở về 0. Nếu có *thread* khác gọi *wait*, nó sẽ bị chặn (*blocked*) cho đến khi *semaphore* được giải phóng. Sau khi hoàn tất công việc trong *critical section*, *thread* sẽ gọi hàm *post* để tăng giá trị *semaphore*, cho phép *thread* tiếp theo được thực thi.



2.2.3 Code

2.2.4 Kết quả

```
1 Producer: Count is 1
2 Producer: Count is 1
3 Producer: Count is 2
4 Producer: Count is 3
5 Producer: Count is 4
6 Producer: Count is 5
7 Producer: Count is 6
8 Producer: Count is 7
9 Producer: Count is 8
10 Producer: Count is 9
11 Producer: Count is 10
12 Producer: Count is 11
13 Producer: Count is 12
14 Producer: Count is 13
15 Producer: Count is 14
16 Producer: Count is 15
17 Consumer: Count is 0
18 Consumer: Count is 15
19 Consumer: Count is 14
20 Consumer: Count is 13
21 -----
22 Producer: Count is 1
23 Producer: Count is 1
24 Producer: Count is 2
25 Producer: Count is 3
26 Producer: Count is 4
27 Producer: Count is 5
28 Producer: Count is 6
29 Producer: Count is 7
30 Producer: Count is 8
31 Producer: Count is 9
32 Producer: Count is 10
33 Producer: Count is 11
34 Producer: Count is 12
35 Producer: Count is 13
36 Producer: Count is 14
37 Producer: Count is 15
38 Producer: Count is 16
39 Producer: Count is 17
40 Producer: Count is 18
41 Consumer: Count is 0
42 Consumer: Count is 18
43 Consumer: Count is 17
```

Full kết quả có ở đây và ở đây



2.2.5 Giải thích kết quả

- Khi không có **semaphore**: Giá trị của **count** là 199 ở một *thread*, nhưng *thread* khác lại đọc được giá trị là 0, sau đó mới đọc được 198. Nguyên nhân của hiện tượng này là do cả hai *thread* đồng thời truy cập và thao tác trên biến **count**, dẫn đến *race condition* khi đọc và ghi biến không được đồng bộ hóa.
- Khi có **semaphore**: Kết quả thu được hoàn toàn đúng như mong đợi, hai *thread* phối hợp nhịp nhàng bằng cách chờ nhau để truy cập vào *critical section*. Nhờ đó, biến **count** được tăng và giảm một cách tuần tự, đảm bảo việc đọc và ghi dữ liệu diễn ra đúng thứ tự.

2.3 Bài 3

2.3.1 Đề bài

Cho 2 process A và B chạy song song

2.3.2 Cách làm

- Hiện thực hai tiến trình **processA** và **processB** chạy song song, cùng chia sẻ biến **x**. Cả hai tiến trình thực hiện thao tác tăng giá trị biến **x**, kiểm tra giới hạn giá trị (reset khi **x == 20**), và in giá trị **x**.

2.3.3 Code

2.3.4 Kết quả

2.3.5 Giải thích kết quả

- Giá trị của **x** sẽ không tăng tuần tự.
- Hiện tượng **x** nhảy không theo thứ tự xảy ra vì hai tiến trình ghi đè giá trị **x**.

2.4 Bài 4

2.4.1 Đề bài

Đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3.

2.4.2 Cách làm

- Sử dụng **mutex** để bảo vệ biến **x**, đảm bảo rằng hai *thread* không truy cập đồng thời vào biến này. Áp dụng hàm **pthread_mutex_lock** để khóa **mutex** trước khi thực thi đoạn mã trong vùng tranh chấp (*critical section*) và dùng **pthread_mutex_unlock** để mở khóa sau khi đoạn mã hoàn tất.



2.4.3 Code

2.4.4 Kết quả

2.4.5 Giải thích kết quả

- Khi sử dụng mutex, chỉ một thread được phép truy cập biến x tại một thời điểm. Điều này đảm bảo không có hai thread đồng thời thực hiện đọc và ghi lên cùng một biến, từ đó loại bỏ khả năng xảy ra lỗi.

Chương 3

BÀI TẬP ÔN TẬP

3.1 Bài 1

3.1.1 Đề bài

Biến `ans` được tính từ các biến `x1`, `x2`, `x3`, `x4`, `x5`, `x6` như sau:

$$w = x1 \cdot x2; \quad (a)$$

$$v = x3 \cdot x4; \quad (b)$$

$$y = v \cdot x5; \quad (c)$$

$$z = v \cdot x6; \quad (d)$$

$$y = w \cdot y; \quad (e)$$

$$z = w \cdot z; \quad (f)$$

$$ans = y + z; \quad (g)$$

Giả sử các lệnh từ (a) đến (g) nằm trên các thread chạy song song với nhau. Hãy lập trình mô phỏng và đồng bộ trên C trong hệ điều hành Linux theo thứ tự sau:

- (c), (d) chỉ được thực hiện sau khi `v` được tính.
- (e) chỉ được thực hiện sau khi `w` và `y` được tính.
- (g) chỉ được thực hiện sau khi `y` và `z` được tính.

3.1.2 Cách làm

- Chúng ta sẽ khởi tạo 3 **semaphore** để đảm bảo các tiến trình thực thi theo đúng thứ tự. Đầu tiên, vì (c) và (d) chỉ được thực hiện sau khi tính xong `v` (tức là sau (b)), ta khởi tạo **semaphore** `sem_c_d` với giá trị ban đầu là 0. Trong các *thread* thực hiện (c) và (d), sẽ gọi `wait` trên `sem_c_d` trước khi bắt đầu tính toán. Sau khi (b) hoàn thành, ta gọi `sem_post` hai lần để kích hoạt cả (c) và (d). Vì (c) và (d) chỉ đọc giá trị `v` sau khi nó đã ổn định, hai tiến trình này có thể chạy song song.



- Tương tự, (e) chỉ được thực thi khi cả (a) và (c) hoàn thành, nghĩa là (e) cần đợi kết quả từ cả hai. Ta khởi tạo một **semaphore** khác với giá trị ban đầu là 0, và gọi **wait** hai lần trong (e). Sau khi (a) và (c) hoàn thành, ta lần lượt gọi **sem_post** để kích hoạt (e).
- Cuối cùng, (g) cũng được xử lý theo cách tương tự như (e), đảm bảo chỉ được thực hiện sau khi (e) và (f) hoàn tất.

3.1.3 Code

3.1.4 Kết quả

3.1.5 Giải thích kết quả

- Thứ tự thực thi chi tiết có thể thay đổi, nhưng các ràng buộc sau luôn được đảm bảo:
 - (c) và (d) luôn được tính sau (b).
 - (e) luôn được tính sau (a) và (c).
 - (g) luôn được tính sau (e) và (f).
- Sự khác biệt trong thứ tự thực thi chi tiết là do các thread chạy đồng thời.
- Ngoài những thread có quy định thứ tự rõ ràng, các thread còn lại có thể thực thi song song, và thread nào hoàn thành trước sẽ không tuân theo một quy luật cố định nào.

Chương 4

Source code

Code tham khảo có ở đây