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

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 đảm bảo an toàn cho các biến `sells` và `products` trong môi trường đa luồng, `mutex` được sử dụng nhằm thực hiện đồng bộ hóa. Khi một *thread* cần truy cập vào *critical section*, nó sẽ sử dụng `pthread_mutex_lock` để khóa `mutex`, đảm bảo rằng chỉ có nó được phép thao tác trên dữ liệu trong thời gian này. Sau khi hoàn tất, *thread* sẽ gọi `pthread_mutex_unlock` để giải phóng khóa, cho phép các *thread* khác tiếp tục hoạt động. Cách tiếp cận này giúp duy trì tính nhất quán của dữ liệu chia sẻ và ngăn chặn xung đột.

2.1.3 Code

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdbool.h>

int products = 0;
int sells = 0;

sem_t sem;
pthread_mutex_t mutex;

void* producer(void* arg) {
    while (true) {
        sem_wait(&sem);
```

```
        pthread_mutex_lock(&mutex);
        if (sells < products) {
            sells++;
        }
        pthread_mutex_unlock(&mutex);
    }
}

void* consumer(void* arg) {
    while (true) {
        pthread_mutex_lock(&mutex);
        if (products < sells + 945) { //09450945
            products++;
        }
        pthread_mutex_unlock(&mutex);
        sem_post(&sem);
    }
}

int main() {
    pthread_t producer_thread, consumer_thread;
    pthread_mutex_init(&mutex, NULL);
    sem_init(&sem, 0, 0);

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    for (int i = 0; i < 5; i++) {
        printf("products is %d and sells is %d\n", products, sells);
        sleep(0.5);
    }
    sem_destroy(&sem);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

2.1.4 Kết quả

```
products is 0 and sells is 0
products is 10614 and sells is 10005
products is 20305 and sells is 20131
products is 30481 and sells is 30185
products is 41671 and sells is 40884
-----------------------------------
products is 0 and sells is 0
products is 10194 and sells is 8773
products is 20016 and sells is 18978
products is 30990 and sells is 30990
```

```
products is 42059 and sells is 41902
-------------------------------------
products is 0 and sells is 0
products is 9374 and sells is 8349
products is 22333 and sells is 20912
products is 34788 and sells is 33949
products is 48967 and sells is 48940
```

2.1.5 Giải thích kết quả

- Trong đoạn mã, **Consumer** luôn tăng giá trị của `products` trước. Cơ chế `semaphore` đảm bảo rằng **Producer** chỉ được phép bán khi đã có sản phẩm tồn tại.
- Giá trị của `products` luôn lớn hơn `sells` và nhỏ hơn hoặc bằng `sells + 0945`.
- Độ chênh lệch giữa `products` và `sells` được quyết định bởi tốc độ thực thi của các luồng trong hệ thống.
- Việc sử dụng `mutex` và `semaphore` đã đảm bảo tính đúng đắn của chương trình. Tuy nhiên, sự khác biệt về tốc độ xử lý giữa các luồng là đặc tính tự nhiên của 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

Một biến `semaphore` được khởi tạo với giá trị ban đầu là 0, nhằm quản lý đồng bộ giữa *Producer* và *Consumer*. Trước khi một *thread* đi vào *critical section*, nó sẽ gọi hàm `wait` của `semaphore`. Khi đó, chỉ một *thread* được phép thực thi trong vùng *critical section*, vì giá trị của `semaphore` giảm xuống 0. Nếu có *thread* khác cũng gọi `wait` trong thời gian này, nó sẽ bị chặn (*blocked*) cho đến khi `semaphore` được giải phóng. Sau khi hoàn thành công việc trong *critical section*, *thread* sẽ gọi hàm `post` để tăng giá trị của `semaphore`, từ đó cho phép *thread* tiếp theo được tiếp tục thực thi.

2.2.3 Code

2.2.4 Kết quả

```
Producer: Count is 1
Producer: Count is 1
Producer: Count is 2
Producer: Count is 3
Producer: Count is 4
Producer: Count is 5
Producer: Count is 6
Producer: Count is 7
Producer: Count is 8
Producer: Count is 9
Producer: Count is 10
Producer: Count is 11
Producer: Count is 12
Producer: Count is 13
Producer: Count is 14
Producer: Count is 15
Consumer: Count is 0
Consumer: Count is 15
Consumer: Count is 14
Consumer: Count is 13
---------------------
Producer: Count is 1
Producer: Count is 1
Producer: Count is 2
Producer: Count is 3
Producer: Count is 4
Producer: Count is 5
Producer: Count is 6
Producer: Count is 7
Producer: Count is 8
Producer: Count is 9
Producer: Count is 10
Producer: Count is 11
Producer: Count is 12
Producer: Count is 13
Producer: Count is 14
Producer: Count is 15
Producer: Count is 16
Producer: Count is 17
Producer: Count is 18
Consumer: Count is 0
Consumer: Count is 18
Consumer: Count is 17
```

Full kết quả có ở đây và ở đây

2.2.5 Giải thích kết quả

- **Khi không sử dụng `semaphore`**: Giá trị của `count` có thể là 18 trên một *thread*, trong khi *thread* khác lại đọc được giá trị là 0, rồi sau đó là 18. Nguyên nhân của hiện tượng này là do cả hai *thread* truy cập và thao tác đồng thời trên biến `count`, dẫn đến hiện tượng *race condition*, khi việc đọc và ghi biến không được đồng bộ.
- **Khi sử dụng `semaphore`**: Kết quả thu được hoàn toàn chính xác như kỳ vọng. Hai *thread* phối hợp chặt chẽ bằng cách lần lượt chờ nhau để truy cập vào *critical section*. Điều này đảm bảo rằng biến `count` được tăng và giảm theo thứ tự, đồng thời việc đọc và ghi dữ liệu diễn ra chính xác và đồng bộ.

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

Code bài 3 ở đây

2.3.4 Kết quả

```
1
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

2.3.5 Giải thích kết quả

- Giá trị của x sẽ không tăng tuần tự.
- Vì hai tiến trình ghi đè xx nên giá trị x biến đổi không theo thứ tự

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 đồng bộ hóa truy cập vào biến `x`, nhằm đảm bảo rằng không có hai *thread* nào có thể thao tác đồng thời trên biến này. Trước khi thực hiện đoạn mã trong vùng tranh chấp (*critical section*), gọi hàm `pthread_mutex_lock` để khóa `mutex`. Sau khi hoàn tất công việc, sử dụng `pthread_mutex_unlock` để mở khóa, cho phép các *thread* khác tiếp tục truy cập.

2.4.3 Code

Code bài 4 ở đây

2.4.4 Kết quả

```
16
17
18
19
0
1
2
3
4
5
6
7
8
9
```

2.4.5 Giải thích kết quả

- Khi sử dụng `mutex`, mỗi thời điểm chỉ có một *thread* được phép truy cập vào biến `x`. Cơ chế này đảm bảo rằng không có hai *thread* nào đồng thời thực hiện thao tác đọc và ghi trên cùng một biến, qua đó loại bỏ nguy cơ xảy ra lỗi hoặc xung đột dữ liệu (*race condition*).

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:

$$
\begin{aligned}
w &= x1 \cdot x2; \quad \text{(a)} \\
v &= x3 \cdot x4; \quad \text{(b)} \\
y &= v \cdot x5; \quad \text{(c)} \\
z &= v \cdot x6; \quad \text{(d)} \\
y &= w \cdot y; \quad \text{(e)} \\
z &= w \cdot z; \quad \text{(f)} \\
ans &= y + z; \quad \text{(g)}
\end{aligned}
$$

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 được thực thi 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), chúng 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 được tính xong, 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. Để đảm bảo điều này, 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) được xử lý theo cách tương tự như (e), đảm bảo chỉ được thực hiện sau khi (e) và (f) hoàn tất. Điều này được thực hiện bằng cách khởi tạo một `semaphore` bổ sung và sử dụng nó để đồng bộ hóa thứ tự thực thi.

3.1.3 Code

Code bài tập về nhà ở đây

3.1.4 Kết quả

```
Calulated w = 2 (a)
Calulated v = 12 (b)
Calulated y = 60 (c)
Calulated z = 72 (d)
Calulated y = 120 (e)
Calulated z = 144 (f)
Calulated ans = 264 (g)
```

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