Báo cáo thực hành môn Hệ điều hành - Giảng viên: Phạm Quốc Hùng.

Họ và tên: Lê Như Thực Mã số sinh viên: 24521747 Lớp: IT007.Q112.1

HỆ ĐIỀU HÀNH
BÁO CÁO LAB 5

CHECKLIST

5.5. BÀI TẬP THỰC HÀNH

	BT 1	BT 2	BT 3	BT 4
Trình bày cách làm	☒	☒	☒	☒
Chụp hình minh chứng	☒	☒	☒	☒
Giải thích kết quả	☒	☒	☒	☒

5.6. BÀI TẬP ÔN TẬP

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

Tự chấm điểm: 10/10

**Lưu ý: Xuất báo cáo theo định dạng PDF, đặt tên theo cú pháp:*
<Tên nhóm>_LAB5.pdf

5.5. BÀI TẬP THỰC HÀNH

1. 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]

- Sử dụng cơ chế **Semaphore** để giải quyết bài toán Bounded-Buffer (Bộ đệm giới hạn).

- Khai báo 2 semaphore: `sem_full` (số sản phẩm hiện có, khởi tạo = 0) và `sem_empty` (số chỗ trống, khởi tạo = 1747 - *là 4 số cuối MSSV*).

- **Thread Producer:** Chờ `sem_empty` (chờ kho còn chỗ), tăng `products`, báo hiệu `sem_full`.

- **Thread Consumer:** Chờ `sem_full` (chờ có hàng), tăng `sells`, báo hiệu `sem_empty`.

- Đảm bảo điều kiện: `sells <= products <= sells + 1747`

```c
#define MAX_ITEMS 1747

int sells = 0;
int products = 0;

sem_t sem_full;
sem_t sem_empty;

void* processA(void* arg) {
    while (1) {
        sem_wait(&sem_full);
        sells++;
        printf("Sells: %d, Products: %d\n", sells, products);
        sem_post(&sem_empty);
    }
}

void* processB(void* arg) {
    while (1) {
        sem_wait(&sem_empty);
        products++;
        printf("Produced! Sells: %d, Products: %d\n", sells, products);
        sem_post(&sem_full);
    }
}
```

```c
int main() {
    sem_init(&sem_full, 0, 0);
    sem_init(&sem_empty, 0, MAX_ITEMS);

    pthread_t tA, tB;

    pthread_create(&tA, NULL, processA, NULL);
    pthread_create(&tB, NULL, processB, NULL);

    pthread_join(tA, NULL);
    pthread_join(tB, NULL);

    return 0;
}
```

```
Produced! Sells: 5851032, Products: 5851045
Produced! Sells: 5851032, Products: 5851046
Produced! Sells: 5851032, Products: 5851047
Produced! Sells: 5851032, Products: 5851048
Produced! Sells: 5851032, Products: 5851049
Produced! Sells: 5851032, Products: 5851050
Produced! Sells: 5851032, Products: 5851051
Produced! Sells: 5851032, Products: 5851052
Produced! Sells: 5851032, Products: 5851053
Produced! Sells: 5851032, Products: 5851054
Produced! Sells: 5851032, Products: 5851055
```

- Chương trình hoạt động theo mô hình sản xuất - tiêu thụ.

- Biến products luôn lớn hơn hoặc bằng sells (nhờ sem_wait ở consumer).

- Hiệu số products - sells không bao giờ vượt quá 1747 (nhờ sem_wait ở producer chặn khi kho đầy).

- Hai tiến trình hoạt động đồng bộ, không xảy ra xung đột dữ liệu.

2. 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.

- Sử dụng một Mutex tên là sem_lock khởi tạo giá trị 1 để bảo vệ Critical Section.
- **Thread 1 (Thêm):** sem_wait(&sem_lock) → thêm số vào mảng a, tăng count → sem_post(&sem_lock).
- **Thread 2 (Lấy):** sem_wait(&sem_lock) → kiểm tra count > 0 thì lấy ra, giảm count → sem_post(&sem_lock).
- Nếu không có sem_lock, dữ liệu biến count sẽ bị sai lệch (race condition)

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

#define N 100
int a[N];
int count = 0;
sem_t sem_lock;

void* thread_add(void* arg) {
    while (1) {
        sem_wait(&sem_lock);

        if (count < N) {
            int val = rand() % 100;
            a[count] = val;
            count++;
            printf("Added %d. Count: %d\n", val, count);
        }

        sem_post(&sem_lock);
        sleep(1);
    }
}
void* thread_remove(void* arg) {
    while (1) {
        sem_wait(&sem_lock);

        if (count > 0) {
            count--;
            printf("Removed element. Count: %d\n", count);
        } else {
            printf("Nothing in array a\n");
        }
        sem_post(&sem_lock);
        sleep(2);
    }
}
```

```c
int main() {
    srand(time(NULL));
    sem_init(&sem_lock, 0, 1);

    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_add, NULL);
    pthread_create(&t2, NULL, thread_remove, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    sem_destroy(&sem_lock);
    return 0;
}
```

```
lenhuthuc@LAPTOP-SLH7F0A4:~$ ./l
Added 22. Count: 1
Removed element. Count: 0
Nothing in array a
Nothing in array a
Added 95. Count: 1
Removed element. Count: 0
Nothing in array a
Nothing in array a
```

- Kết quả cho thấy mảng a và biến đếm count được truy cập an toàn.
- Không xảy ra tình trạng hai thread cùng thay đổi biến count một lúc (gây sai số).
- Khi mảng rỗng, thread lấy phần tử sẽ thông báo "Nothing in array a" đúng như yêu cầu logic.

3. Cho 2 process A và B chạy song song như sau:

int x = 0;	
PROCESS A	**PROCESS B**

processA()
{
while(1){
x = x + 1;
	processB()
	{
	while(1){
	x = x + 1;

if (x == 20)
x = 0;
print(x);
}
}
	if (x == 20)
	x = 0;
	print(x);
	}
	}

Hiện thực mô hình trên C trong hệ điều hành Linux và nhận xét kết quả.

- Tạo biến toàn cục x = 0.
- Tạo 2 thread (Process A và B) cùng chạy một vòng lặp vô tận:
    + Tăng x lên 1.
    + Kiểm tra if (x == 20) x = 0.
    + In giá trị x.
- Không sử dụng cơ chế đồng bộ nào để quan sát lỗi tranh chấp tài nguyên

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int x = 0;

void* process(void* arg) {
    while (1) {
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("%s: %d\n", (char*)arg, x);
    }
}

int main() {
    pthread_t pA, pB;
    pthread_create(&pA, NULL, process, "Process A");
    pthread_create(&pB, NULL, process, "Process B");

    pthread_join(pA, NULL);
    pthread_join(pB, NULL);
    return 0;
}
```

```
Process A: 2
Process A: 3
Process A: 4
Process A: 5
Process A: 6
Process A: 7
Process A: 8
Process A: 9
Process A: 10
Process A: 11
Process B: 14
Process B: 13
Process B: 14
Process B: 15
```

- **Kết quả bị sai :** Xuất hiện các giá trị bất thường (nhảy từ 11 lên 14).

- **Nguyên nhân:** Do xảy ra **Race Condition** (Điều kiện đua). Lệnh x = x + 1 và lệnh kiểm tra if không phải là nguyên tử (atomic). Thread này có thể bị ngắt giữa chừng khi đang tăng x, thread kia nhảy vào thay đổi x tiếp, dẫn đến logic reset về 0 bị bỏ qua hoặc tính sai.

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

- Khai báo biến `pthread_mutex_t mutex`.

- Khởi tạo mutex bằng `pthread_mutex_init`.

- Trong vòng lặp của cả Process A và Process B:

    + Gọi `pthread_mutex_lock(&mutex)` trước khi truy cập biến `x`.

    + Thực hiện tăng `x` và kiểm tra điều kiện reset.

    + Gọi `pthread_mutex_unlock(&mutex)` sau khi in xong.

- Hủy mutex khi chương trình kết thúc

```c
int x = 0;
pthread_mutex_t mutex;
void* process(void* arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("%s: %d\n", (char*)arg, x);

        pthread_mutex_unlock(&mutex);
        usleep(100000);
    }
}

int main() {
    pthread_mutex_init(&mutex, NULL);
    pthread_t pA, pB;
    pthread_create(&pA, NULL, process, "Process A");
    pthread_create(&pB, NULL, process, "Process B");

    pthread_join(pA, NULL);
    pthread_join(pB, NULL);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

```
Process A: 12
Process A: 13
Process A: 14
Process A: 15
Process A: 16
Process A: 17
Process A: 18
Process A: 19
Process A: 0
Process A: 1
Process B: 9
Process B: 3
Process B: 4
Process B: 5
Process B: 6
Process B: 7
```

- Kết quả chính xác: Biến x tăng từ 0 đến 19 rồi quay về 0 đúng chu kỳ.

- Giải thích**:** Mutex đảm bảo tính Mutual Exclusion. Tại một thời điểm, chỉ có duy nhất 1 thread được phép vào Critical Section để sửa đổi biến x. Thread kia phải chờ đến khi khóa được mở mới được thực thi.

5.6. BÀI TẬP ÔN TẬP

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

w = x1 * x2; (a)

v = x3 * x4; (b)

y = v * x5; (c)

z = v * x6; (d)

y = w * y; (e)

z = w * z; (f)

ans = y + z; (g)

Giả sử các lệnh từ (a) → (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

- Phân tích sự phụ thuộc dữ liệu:
  + v cần cho y(c) và z(d).
  + w cần cho y(e) và z(f).
  + ans cần y và z cuối cùng.
- Sử dụng các Semaphore để điều phối thứ tự:
  + sem_v, sem_w: Báo hiệu tính xong bước 1.
  + sem_y_step1, sem_z_step1: Báo hiệu tính xong bước trung gian.
- Mỗi phép tính chạy trên 1 thread riêng, sử dụng sem_wait để chờ dữ liệu đầu vào và sem_post để báo hiệu dữ liệu đầu ra đã sẵn sàng.

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int x1=1, x2=2, x3=3, x4=4, x5=5, x6=6;
int w, v, y, z, ans;

sem_t sem_v, sem_w, sem_y_c, sem_z_d, sem_y_e, sem_z_f;

void* calc_w(void* arg) {
    w = x1 * x2;
    printf("Calculated w = %d\n", w);
    sem_post(&sem_w);
    sem_post(&sem_w);
    return NULL;
}

void* calc_v(void* arg) {
    v = x3 * x4;
    printf("Calculated v = %d\n", v);
    sem_post(&sem_v);
    sem_post(&sem_v);
    return NULL;
}
```

```c
void* calc_y_c(void* arg) {
    sem_wait(&sem_v);
    y = v * x5;
    printf("Calculated y(c) = %d\n", y);
    sem_post(&sem_y_c);
    return NULL;
}

void* calc_z_d(void* arg) {
    sem_wait(&sem_v);
    z = v * x6;
    printf("Calculated z(d) = %d\n", z);
    sem_post(&sem_z_d);
    return NULL;
}
```

```c
void* calc_y_e(void* arg) {
    sem_wait(&sem_w);
    sem_wait(&sem_y_c);
    y = w * y;
    printf("Calculated y(e) = %d\n", y);
    sem_post(&sem_y_e);
    return NULL;
}

void* calc_z_f(void* arg) {
    sem_wait(&sem_w);
    sem_wait(&sem_z_d);
    z = w * z;
    printf("Calculated z(f) = %d\n", z);
    sem_post(&sem_z_f);
    return NULL;
}

void* calc_ans(void* arg) {
    sem_wait(&sem_y_e);
    sem_wait(&sem_z_f);
    ans = y + z;
    printf("Final ans = %d\n", ans);
    return NULL;
}
```

```c
int main() {
    sem_init(&sem_v, 0, 0);
    sem_init(&sem_w, 0, 0);
    sem_init(&sem_y_c, 0, 0);
    sem_init(&sem_z_d, 0, 0);
    sem_init(&sem_y_e, 0, 0);
    sem_init(&sem_z_f, 0, 0);

    pthread_t t1, t2, t3, t4, t5, t6, t7;

    pthread_create(&t1, NULL, calc_w, NULL);
    pthread_create(&t2, NULL, calc_v, NULL);
    pthread_create(&t3, NULL, calc_y_c, NULL);
    pthread_create(&t4, NULL, calc_z_d, NULL);
    pthread_create(&t5, NULL, calc_y_e, NULL);
    pthread_create(&t6, NULL, calc_z_f, NULL);
    pthread_create(&t7, NULL, calc_ans, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_join(t5, NULL);
    pthread_join(t6, NULL);
    pthread_join(t7, NULL);

    return 0;
}
```

```
lenhuthuc@LAPTOP-SLH7F0A4:~$ ./lab05
Calculated w = 2
Calculated v = 12
Calculated y(c) = 60
Calculated z(d) = 72
Calculated y(e) = 120
Calculated z(f) = 144
Final ans = 264
```

- Các phép tính được thực hiện song song ở những nhánh không phụ thuộc nhau (ví dụ w và v có thể tính cùng lúc).
- Các phép tính phụ thuộc (như ans) luôn đợi đủ dữ liệu mới chạy.
- Kết quả cuối cùng ans được tính chính xác sau khi tất cả các bước trung gian đã hoàn tất, đảm bảo tính đúng đắn của luồng dữ liệu.