LAB 05 – GROUP 5

CLASS : IT007.019.1 && IT007.019.2

Members:

1. Hà Huy Hoàng | 22520460
2. Nguyễn Duy Hoàng | 22520467
3. Nguyễn Hoàng Hiệp | 22520452
4. Nguyễn Hoàng Phúc | 22521129

SUMMARY

Task		Status	Members
Thực hành	1	Hoàn thành	4
	2	Hoàn thành	2
	3	Hoàn thành	3
	4	Hoàn thành	1
Ôn tập	1	Hoàn thành	4

A. 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]

```c
#include <semaphore.h>
#include <stdio.h>
#include <pthread.h>
int sells = 0, products = 0;
sem_t sem, sem1;

void* processA(void* mes) {
    while (1) {
        sem_wait(&sem);
        sells++;
        printf("SELL = %d\n", sells);
        printf("SELL_1 = %d\n", sells + 1129);
        sem_post(&sem1);
    }
}
void* processB(void* mess) {
```

```
    while (1) {
        sem_wait(&sem1);
        products++;
        printf("PRODUCT = %d\n", products);
        sem_post(&sem);
    }
}
int main() {
    sem_init(&sem, 0, 0);
    sem_init(&sem1, 0, sells + 1129);
    pthread_t pA, pB;
    pthread_create(&pA, NULL, &processA, NULL);
    pthread_create(&pA, NULL, &processB, NULL);
    while (1) {}
    return 0;
}
```

- 2 semaphore để kiểm tra điều kiện gồm sem và sem1, 2 hàm process để thực hiện công việc sản xuất hàng và bán hàng
- Ta sẽ gán cho 2 biến là sells và products đều bằng 0, khai báo 2 semaphore với sem = 0 và sem1 = 1129 (4 số cuối mã số sinh viên 22521129)
- processA: bán hàng

+ sem_wait(&sem) để kiểm tra giá trị của sem, nếu sem bằng 0 thì block, nếu sem > 0 thì thực hiện lệnh sem = sem – 1. Sau đó thực hiện sells++ (muốn bán được hàng thì phải có hàng trước). Vậy hàm sem_wait() mục đích là để kiểm tra xem đã có hàng hay chưa,
+ sem_post(&sem1) tăng giá trị của sem1 lên làm điều kiện cho hàm processB

- processB: sản xuất hàng

+ sem_wait(&sem1) kiểm tra sem1 có bằng 0 (tương tự processA), mục đích kiểm tra products có bé hơn sells + 1129 hay không để thỏa mãn điều kiện

+Tăng biến products lên và sử dụng hàm sem_post(&sem) để thông báo cho processA biết là đã có hàng.

```
PRODUCT = 1
PRODUCT = 2
PRODUCT = 3
PRODUCT = 4
PRODUCT = 5
PRODUCT = 6
PRODUCT = 7
PRODUCT = 8
PRODUCT = 9
PRODUCT = 10
PRODUCT = 11
PRODUCT = 12
PRODUCT = 13
PRODUCT = 14
PRODUCT = 15
PRODUCT = 16
PRODUCT = 17
PRODUCT = 18
PRODUCT = 19
PRODUCT = 20
PRODUCT = 21
PRODUCT = 22
PRODUCT = 23
PRODUCT = 24
SELL = 1
SELL_1 = 1130
SELL = 2
SELL_1 = 1131
SELL = 3
SELL_1 = 1132
PRODUCT = 25
PRODUCT = 26
SELL_1 = 1150
SELL = 22
SELL_1 = 1151
SELL = 23
SELL_1 = 1152
SELL = 24
SELL_1 = 1153
SELL = 25
SELL_1 = 1154
SELL = 26
SELL_1 = 1155
SELL = 27
SELL_1 = 1156
SELL = 28
SELL_1 = 1157
SELL = 29
SELL_1 = 1158
SELL = 30
SELL_1 = 1159
SELL = 31
SELL_1 = 1160
SELL = 32
SELL_1 = 1161
SELL = 33
SELL_1 = 1162
SELL = 34
SELL_1 = 1163
SELL = 35
SELL_1 = 1164
SELL = 36
SELL_1 = 1165
SELL = 37
SELL_1 = 1166
SELL_1 = 1185
SELL = 57
SELL_1 = 1186
SELL = 58
SELL_1 = 1187
SELL = 59
SELL_1 = 1188
SELL = 60
SELL_1 = 1189
SELL = 61
SELL_1 = 1190
PRODUCT = 68
PRODUCT = 69
PRODUCT = 70
PRODUCT = 71
PRODUCT = 72
PRODUCT = 73
PRODUCT = 74
PRODUCT = 75
PRODUCT = 76
PRODUCT = 77
PRODUCT = 78
PRODUCT = 79
PRODUCT = 80
PRODUCT = 81
PRODUCT = 82
PRODUCT = 83
PRODUCT = 84
PRODUCT = 85
PRODUCT = 86
PRODUCT = 87
PRODUCT = 88
PRODUCT = 89
```

Nhìn vào kết quả chạy, products luôn luôn lớn hơn sells và luôn luôn nhỏ hơn sell_1 (sells + 1129), thỏa mãn được điều kiện mà bài tập đã nêu ra.

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:
a. 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.

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

int* a;
int n;
int iNum = 0;

void Arrange(int* a, int x) {
    if (x == iNum) {
        iNum--;
    } else {
        for (int i = x; i < iNum - 1; ++i) {
            a[i] = a[i + 1];
        }
        iNum--;
    }
}

void* ProcessA(void* mess) {
    while (1) {
        srand((int)time(0));
        a[iNum] = rand();
        iNum++;
        printf("So phan tu trong mang a: %d\n", iNum);
    }
}

void* ProcessB(void* mess) {
    while (1) {
        srand((int)time(0));
        if (iNum == 0) {
            printf("Nothing in array\n");
        } else {
            int r = rand() % iNum;
            Arrange(a, r);
            printf("So phan tu trong mang a sau khi lay ra la: %d\n", iNum);
```

```
        }
    }
}

int main() {
    printf("Nhap so phan tu: \n");
    scanf("%d", &n);
    a = (int*)malloc(n * sizeof(int));
    pthread_t PA, PB;
    pthread_create(&PA, NULL, &ProcessA, NULL);
    pthread_create(&PB, NULL, &ProcessB, NULL);
    while (1) {}
    return 0;
}
```

Race Condition: Hai thread ProcessA và ProcessB có thể cùng thao tác vào mảng a mà không có sự đồng bộ hóa. Nếu ProcessA thực hiện việc tăng iNum và ProcessB cùng lúc thực hiện việc giảm iNum, có thể xảy ra tình huống không lường trước được và dẫn đến kết quả không chính xác hoặc lỗi.

Thiếu bảo vệ đối với biến iNum: iNum được truy cập và thay đổi bởi cả hai thread mà không có bất kỳ cơ chế bảo vệ nào. Điều này có thể dẫn đến xung đột giữa các thao tác đọc và ghi trên iNum.

```
nguyenduyhoang-22520467@HOANGND04:~$ gcc -o lab5_bai2a lab5_bai2a.
nguyenduyhoang-22520467@HOANGND04:~$ ./lab5_bai2a
Nhap so phan tu:
10
So phan tu trong mang a: 1
So phan tu trong mang a: 1
So phan tu trong mang a: 2
So phan tu trong mang a: 3
So phan tu trong mang a: 4
So phan tu trong mang a: 5
So phan tu trong mang a: 6
So phan tu trong mang a: 7
So phan tu trong mang a: 8
So phan tu trong mang a: 9
So phan tu trong mang a: 10
So phan tu trong mang a: 11
So phan tu trong mang a: 12
So phan tu trong mang a: 13
So phan tu trong mang a: 14
So phan tu trong mang a: 15
So phan tu trong mang a: 16
So phan tu trong mang a: 17
So phan tu trong mang a: 18
So phan tu trong mang a: 19
So phan tu trong mang a: 20
So phan tu trong mang a: 21
So phan tu trong mang a: 22
So phan tu trong mang a: 23
So phan tu trong mang a: 24
So phan tu trong mang a: 25
So phan tu trong mang a: 26
So phan tu trong mang a: 27
So phan tu trong mang a: 28
So phan tu trong mang a: 29
So phan tu trong mang a: 30
So phan tu trong mang a sau khi lay ra la: 0
So phan tu trong mang a sau khi lay ra la: 30
So phan tu trong mang a sau khi lay ra la: 29
So phan tu trong mang a sau khi lay ra la: 28
So phan tu trong mang a sau khi lay ra la: 27
So phan tu trong mang a sau khi lay ra la: 26
So phan tu trong mang a sau khi lay ra la: 25
So phan tu trong mang a sau khi lay ra la: 24
So phan tu trong mang a sau khi lay ra la: 23
So phan tu trong mang a sau khi lay ra la: 22
So phan tu trong mang a sau khi lay ra la: 21
```

b. 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”.

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

int* a;
int n;
int iNum = 0;
```

```c
sem_t sem1, sem2, busy;

void Arrange(int* a, int x) {
    if (x == iNum - 1) {
        iNum--;
    } else {
        for (int i = x; i < iNum - 1; ++i) {
            a[i] = a[i + 1];
        }
        iNum--;
    }
}

void* ProcessA(void* mess) {
    while (1) {
        sem_wait(&sem2);
        sem_wait(&busy);
        srand((unsigned int)time(NULL));
        a[iNum] = rand();
        iNum++;
        printf("So phan tu trong mang a: %d\n", iNum);
        sem_post(&sem1);
        sem_post(&busy);
    }
}

void* ProcessB(void* mess) {
    while (1) {
        sem_wait(&sem1);
        sem_wait(&busy);
        if (iNum == 0) {
            printf("Nothing in array\n");
        } else {
            int r = rand() % iNum;
            Arrange(a, r);
            printf("So phan tu trong mang a sau khi lay ra la: %d\n", iNum);
        }
        sem_post(&sem2);
        sem_post(&busy);

    }
}

int main() {
    printf("Nhap so phan tu: \n");
    scanf("%d", &n);
```

```
    a = (int*)malloc(n * sizeof(int));

    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, n);
    sem_init(&busy, 0, 1);

    pthread_t PA, PB;

    pthread_create(&PA, NULL, &ProcessA, NULL);
    pthread_create(&PB, NULL, &ProcessB, NULL);
    while (1) {
    }
    return 0;
}
```

```
nguyenduyhoang-22520467@HOANGND04:~$ ./lab5_bai2b
Nhap so phan tu:
10
So phan tu trong mang a: 1
So phan tu trong mang a: 2
So phan tu trong mang a: 3
So phan tu trong mang a: 4
So phan tu trong mang a: 5
So phan tu trong mang a: 6
So phan tu trong mang a: 7
So phan tu trong mang a: 8
So phan tu trong mang a: 9
So phan tu trong mang a: 10
So phan tu trong mang a sau khi lay ra la: 9
So phan tu trong mang a sau khi lay ra la: 8
So phan tu trong mang a sau khi lay ra la: 7
So phan tu trong mang a sau khi lay ra la: 6
So phan tu trong mang a sau khi lay ra la: 5
So phan tu trong mang a sau khi lay ra la: 4
So phan tu trong mang a sau khi lay ra la: 3
So phan tu trong mang a sau khi lay ra la: 2
So phan tu trong mang a sau khi lay ra la: 1
So phan tu trong mang a sau khi lay ra la: 0
```

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

PROCESS A	PROCESS B

processA()
{
while (1)
{
x = x + 1;
if (x == 20)
x = 0;
print(x);
}
}
	processB()
	{
	while (1)
	{
	x = x + 1;
	if (x == 20)
	x = 0;
	print(x);
	}
	}

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

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

int x = 0;

void *processA(void *mess)
{
    while (1)
    {
        x++;
        if (x == 20)
        {
            x = 0;
        }
        printf("Process A: x = %d\n", x);
    }
}

void *processB(void *mess)
{
    while (1)
    {
        x += 1;
        if (x == 20)
        {
            x = 0;
        }
        printf("Process B: x = %d\n", x);
    }
}
```

```c
int main()
{
    pthread_t pA, pB;

    pthread_create(&pA, NULL, &processA, NULL);
    pthread_create(&pB, NULL, &processB, NULL);

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

    return 0;
}
```

```
Process B: x = 10
Process B: x = 11
Process B: x = 12
Process B: x = 13
Process B: x = 14
Process B: x = 15
Process B: x = 16
Process B: x = 17
Process B: x = 18
Process B: x = 19
Process B: x = 0
Process A: x = 16
Process A: x = 2
Process A: x = 3
Process A: x = 4
Process A: x = 5
Process A: x = 6
Process A: x = 7
Process A: x = 8
Process A: x = 9
Process A: x = 10
Process A: x = 11
Process A: x = 12
Process A: x = 13
Process A: x = 14
Process A: x = 15
Process A: x = 16
Process A: x = 17
Process A: x = 18
Process A: x = 19
Process A: x = 0
Process A: x = 1
Process A: x = 2
```

```
Process A: x = 15
Process A: x = 16
Process A: x = 17
Process A: x = 18
Process A: x = 19
Process A: x = 0
Process A: x = 1
Process A: x = 2
Process A: x = 3
Process A: x = 4
Process A: x = 5
Process A: x = 6
Process A: x = 7
Process A: x = 8
Process A: x = 9
Process A: x = 10
Process A: x = 11
Process A: x = 12
Process A: x = 13
Process A: x = 14
Process A: x = 15
Process B: x = 14
Process B: x = 17
Process B: x = 18
Process B: x = 19
Process B: x = 0
Process B: x = 1
Process B: x = 2
Process B: x = 3
Process B: x = 4
Process B: x = 5
Process B: x = 6
Process B: x = 7
```

Dựa vào kết quả chạy được cung cấp, có thể nhận thấy rằng giá trị của biến x không nhất quán trong suốt quá trình thực thi chương trình. Điều này là do hai luồng, Process A và Process B, đang truy cập và sửa đổi biến x đồng thời mà không có bất kỳ sự đồng bộ hóa nào.

Do đó, giá trị của x có thể không nhất quán giữa hai luồng và giá trị cuối cùng của x cũng không thể đoán trước được. Hành vi này là do điều kiện chạy đua xảy ra khi hai luồng cố gắng sửa đổi một tài nguyên được chia sẻ mà không có sự đồng bộ hóa thích hợp.

Kết quả chạy cho thấy rằng chương trình có thể dẫn đến kết quả không mong muốn nếu không có sự đồng bộ hóa. Để tránh điều này, cần phải sử dụng các cơ chế đồng bộ hóa để đảm bảo rằng các luồng chỉ truy cập và sửa đổi các biến chia sẻ một cách an toàn.

Trong trường hợp này, có thể sử dụng mutex để đồng bộ hóa quyền truy cập vào biến x. Cụ thể, mỗi khi một luồng cần truy cập hoặc sửa đổi x, nó sẽ khóa mutex trước. Sau khi hoàn tất, luồng sẽ mở khóa mutex để các luồng khác có thể truy cập x.

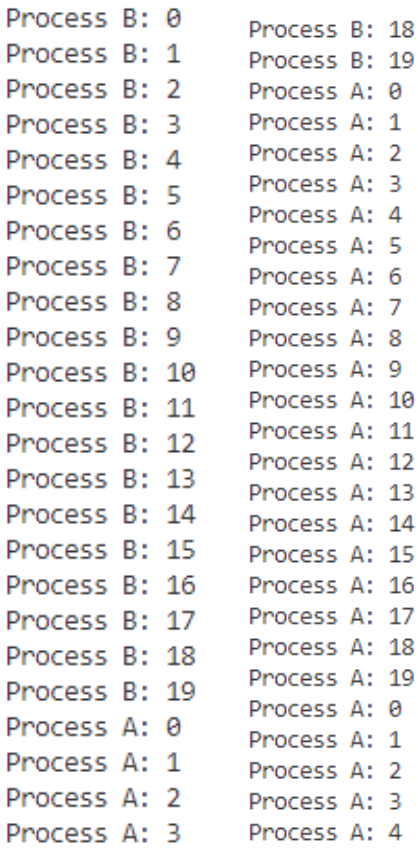
Việc thêm mutex vào chương trình sẽ đảm bảo rằng mỗi luồng chỉ có thể truy cập x khi nó được khóa. Điều này sẽ ngăn chặn các điều kiện chạy đua và đảm bảo rằng giá trị của x luôn nhất quán.

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

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

int x = 0; // Khởi tạo biến toàn cục x
pthread_mutex_t lock; // Khai báo mutex

void* processA(void* arg) // Hàm thực thi quy trình A
{
    while(1){ // Vòng lặp vô hạn
        pthread_mutex_lock(&lock); // Khóa mutex để đảm bảo rằng chỉ có một quy trình có thể truy cập và thay đổi biến x tại một thời điểm
        x = x + 1; // Tăng giá trị của x lên 1
        if (x == 20) // Nếu x đạt đến 20
            x = 0; // Đặt lại x về 0
        printf("Process A: %d\n", x); // In giá trị của x
        pthread_mutex_unlock(&lock); // Mở khóa mutex
    }
}

void* processB(void* arg) // Hàm thực thi quy trình B
{
```

```
    while(1){ // Vòng lặp vô hạn
        pthread_mutex_lock(&lock); // Khóa mutex
        x = x + 1; // Tăng giá trị của x lên 1
        if (x == 20) // Nếu x đạt đến 20
            x = 0; // Đặt lại x về 0
        printf("Process B: %d\n", x); // In giá trị của x
        pthread_mutex_unlock(&lock); // Mở khóa mutex
    }
}

int main()
{
    pthread_t thread_id1, thread_id2; // Khai báo hai thread
    pthread_mutex_init(&lock, NULL); // Khởi tạo mutex

    pthread_create(&thread_id1, NULL, processA, NULL); // Tạo thread cho quy
trình A
    pthread_create(&thread_id2, NULL, processB, NULL); // Tạo thread cho quy
trình B

    while(1){}

    return 0; // Kết thúc chương trình
}
```

Trong đoạn mã trên, chúng ta sử dụng mutex để đảm bảo rằng chỉ có một quy trình có thể truy cập và thay đổi biến x tại một thời điểm. Điều này giúp tránh các vấn đề liên quan đến việc hai quy trình cùng lúc truy cập và thay đổi biến x, dẫn đến kết quả không như mong đợi. Mutex giúp đảm bảo tính nhất quán của dữ liệu khi có nhiều quy trình hoặc luồng cùng lúc truy cập vào cùng một tài nguyên.



Kết quả thực thi bài 4

B. Bài tập ôn tập

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

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

int x1, x2, x3, x4, x5, x6;
int w, v, y, z, ans;

sem_t s15, s16, s23, s24, s35, s46, s57, s67;

void *process1(void *arg) {
    w = x1 * x2;
    sem_post(&s15);
    sem_post(&s16);
    pthread_exit(NULL);
}

void *process2(void *arg) {
    v = x3 * x4;
    sem_post(&s23);
    sem_post(&s24);
    pthread_exit(NULL);
}

void *process3(void *arg) {
    sem_wait(&s23);
    y = v * x5;
    sem_post(&s35);
    pthread_exit(NULL);
}

void *process4(void *arg) {
    sem_wait(&s24);
    z = v * x6;
```

```
    sem_post(&s46);
    pthread_exit(NULL);
}

void *process5(void *arg) {
    sem_wait(&s15);
    sem_wait(&s35);
    y = w * y;
    sem_post(&s57);
    pthread_exit(NULL);
}

void *process6(void *arg) {
    sem_wait(&s16);
    sem_wait(&s46);
    z = w * z;
    sem_post(&s67);
    pthread_exit(NULL);
}

void *process7(void *arg) {
    sem_wait(&s57);
    sem_wait(&s67);
    ans = y + z;
    pthread_exit(NULL);
}

int main() {
    printf("Enter the value for x1: ");
    scanf("%d", &x1);

    printf("Enter the value for x2: ");
    scanf("%d", &x2);

    printf("Enter the value for x3: ");
    scanf("%d", &x3);

    printf("Enter the value for x4: ");
    scanf("%d", &x4);

    printf("Enter the value for x5: ");
    scanf("%d", &x5);

    printf("Enter the value for x6: ");
    scanf("%d", &x6);
```

```
    pthread_t thread1, thread2, thread3, thread4, thread5, thread6, thread7;

    sem_init(&s15, 0, 0);
    sem_init(&s16, 0, 0);
    sem_init(&s23, 0, 0);
    sem_init(&s24, 0, 0);
    sem_init(&s35, 0, 0);
    sem_init(&s46, 0, 0);
    sem_init(&s57, 0, 0);
    sem_init(&s67, 0, 0);

    pthread_create(&thread1, NULL, &process1, NULL);
    pthread_create(&thread2, NULL, &process2, NULL);
    pthread_create(&thread3, NULL, &process3, NULL);
    pthread_create(&thread4, NULL, &process4, NULL);
    pthread_create(&thread5, NULL, &process5, NULL);
    pthread_create(&thread6, NULL, &process6, NULL);
    pthread_create(&thread7, NULL, &process7, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);
    pthread_join(thread5, NULL);
    pthread_join(thread6, NULL);
    pthread_join(thread7, NULL);

    // Use ans
    printf("Ans: %d\n", ans);

    return 0;
}
```

```
nguyenhoangphuc-22521129@LAPTOP-021S54DV:~$ ./m
Enter the value for x1: 1
Enter the value for x2: 2
Enter the value for x3: 3
Enter the value for x4: 4
Enter the value for x5: 5
Enter the value for x6: 6
Ans: 264
```