Báo cáo thực hành môn Hệ điều hành - Giảng viên: Trần Hoàng Lộc.

Họ và tên: Lý Tuấn Khoa Mã số sinh viên: 21522225 Lớp:

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

**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]**

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
sem_t sem;
int sells=0, products=0;
void* PROCESSA()
{
    while(1)
    {
        sem_wait(&sem);
        sells++;
        printf("sells = %d\n", sells);
        sleep(2);
    }
}
void* PROCESSB()
{
    while(1)
    {
        if (products <= sells + 25 + 10) {
            products++;
            printf("products = %d\n", products);
            sem_post(&sem);
            sleep(1);

        }

    }
}
void main()
{
    sem_init(&sem,0,0);
    pthread_t th1, th2;
    pthread_create(&th1, NULL, &PROCESSA, NULL);
    pthread_create(&th2, NULL, &PROCESSB, NULL);
    while(1);
}
```

```
        sem_wait(&sem);
        sells++;
        printf("sells = %d\n", sells);
        sleep(2);
    }
}
void* PROCESSB()
{
    while(1)
    {
        if (products <= sells + 25 + 10) {
            products++;
            printf("products = %d\n", products);
            sem_post(&sem);
            sleep(1);

        }

    }
}
void main()
{
    sem_init(&sem,0,0);
    pthread_t th1, th2;
    pthread_create(&th1, NULL, &PROCESSA, NULL);
    pthread_create(&th2, NULL, &PROCESSB, NULL);
    while(1);
}
```

Result:

```
lytuankhoa_21522225/ $ gcc -o bai1 bai1.c
lytuankhoa_21522225/ $ ./bai1
products = 1
sells = 1
products = 2
sells = 2
products = 3
products = 4
sells = 3
products = 5
products = 6
sells = 4
products = 7
products = 8
sells = 5
products = 9
products = 10
sells = 6
products = 11
products = 12
```

Explain the result and the code run:

- `sem_t sem;`: Declaration of a semaphore variable `sem`.
- `sem_init(&sem, 0, 0);`: Initialization of the semaphore with an initial value of 0.
- `int sells`: Counter for the number of sells.
- `int products`: Counter for the number of products.
- It runs in an infinite loop.
- Checks if the number of products is less than or equal to the sum of sells, 25, and 10.
- If the condition is true, it produces a product, prints the number of products, signals the semaphore (`sem_post`) to indicate a product is available, and then sleeps for 1 second.
- It runs in an infinite loop.
- Waits for the semaphore (`sem_wait`) to be signaled, indicating that a product is available.
- Consumes a product (increments `sells`), prints the number of sells, and then sleeps for 2 seconds.
- Initializes the semaphore.
- Creates two threads (`th1` and `th2`) for `PROCESSA` and `PROCESSB`.

- The main function runs in an infinite loop to keep the program running.

Now, let's analyze the expected behavior:

- `PROCESSB` produces products, and each time it produces a product, it signals the semaphore (`sem_post`), allowing `PROCESSA` to consume a product.

- `PROCESSA` consumes products, and each time it consumes a product, it signals the semaphore (`sem_post`), allowing `PROCESSB` to produce more products.

- The sleep times in both threads (`sleep(1)` in `PROCESSB` and `sleep(2)` in `PROCESSA`) introduce delays to simulate some processing time.

The program will run indefinitely, and you should see interleaved output of sells and products on the console as they are produced and consumed by the two threads. The sleep times ensure that the producer and consumer threads take turns executing and avoid conflicts.

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.

Code:

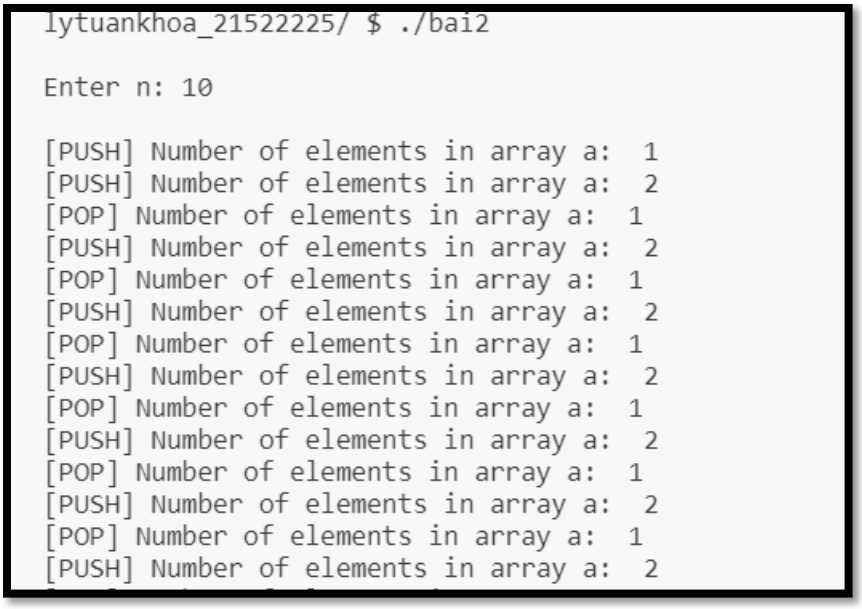
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t sem1, sem2;
int n;
int i = 0;
static int dem = 0;
int a[100000];
void* PROCESS1() {
    while (1) {
        if (dem < n) {
            a[i++] = rand() % (n - 1);
            dem++;
            printf("\n[PUSH] Number of elements in array a: %2d", dem);
        }
        int time_sleep = rand() % 2 + 1;
        sleep(time_sleep);

        sem_post(&sem1);
    }
}
void* PROCESS2() {
    int j, b;
    while (1) {
        sem_wait(&sem1);
```

```c
void* PROCESS2() {
    int j, b;
    while (1) {
        sem_wait(&sem1);
        if (dem == 0) {
            printf("\n[POP] Nothing in array a");
        } else {
            dem--;
            b = a[0];
            for (j = 0; j < dem; j++) {
                a[j] = a[j + 1];
            }
            printf("\n[POP] Number of elements in array a: %2d", dem);
        }

        int time_sleep = rand() % 2 + 1;
        sleep(time_sleep);
    }
}
int main() {
    sem_init(&sem1, 1, 0);
    sem_init(&sem2, 0, 0);
    printf("\nEnter n: ");
    scanf("%d", &n);
    pthread_t th1, th2;
    pthread_create(&th1, NULL, PROCESS1, NULL);
    pthread_create(&th2, NULL, PROCESS2, NULL);
    while (1);
    return 0;
}
```

Result:



Explain:

The program runs two threads, `PROCESS1` and `PROCESS2`, with shared access to an array and a counter (`dem`). Here's a concise summary:

PROCESS1:

- Generates random numbers, adds them to the array if space is available.
- Increments and prints the counter (`dem`).
- Sleeps for 1-2 seconds.
- Signals a semaphore (`sem1`).

PROCESS2:

- Waits for the semaphore (`sem1`).
- If there are elements in the array, pops the first, shifts, and prints.
- Decrements the counter (`dem`).
- Sleeps for 1-2 seconds.

Main:

- Takes user input for array size (`n`).
- Creates threads for `PROCESS1` and `PROCESS2`.
- Runs an infinite loop.

The threads introduce delays for dynamic array modifications, and the program runs indefinitely, showcasing multithreading behavior..

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

int x = 0;	
PROCESS A	**PROCESS B**
processA()	processB()

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

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

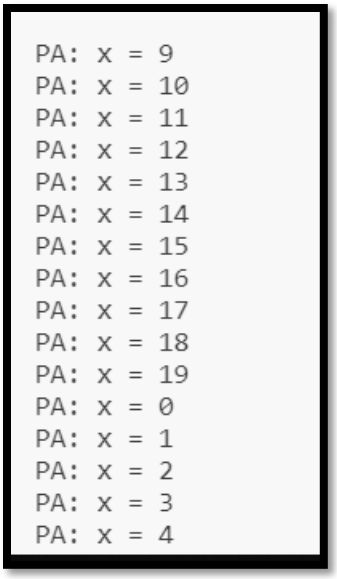
Hiện thực mô hình trên C trong hệ điều hành Linux và nhận xét kết quả.

Trả lời…

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
int x = 0;
void* A() {
    while (1) {
        x = x + 1;
        if (x == 20) {
            x = 0;
        }
        printf("PA: x = %d\n", x);
    }
}
void* B() {
    while (1) {
        x = x + 1;
        if (x == 20) {
            x = 0;
        }
        printf("PB: x = %d\n", x);
        sleep(1); // Move inside the loop for a 1-second delay
    }
}

int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, &A, NULL);
    pthread_create(&th2, NULL, &B, NULL);
    while (1);
    return 0;
}
```

Reutls:



Explain the result and code:

-int x = 0 : A shared variable manipulated by both threads.

- Runs in an infinite loop.
- Increments the value of `x`.
- Resets `x` to 0 if it reaches 20.
- Prints the current value of `x`.
- Runs in an infinite loop (same as Thread A).
- Increments the value of `x`.
- Resets `x` to 0 if it reaches 20.
- Prints the current value of `x`.
- Sleeps for 1 second (outside the loop).

In summary, both threads `A` and `B` perform the same actions, and the only difference is that thread `B` has a sleep statement outside the loop, causing it to sleep for 1 second after completing the loop.

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

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
sem_t sem_1, sem_2;
int x = 0;
pthread_mutex_t mutex;
void* PROCESS1()
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        x++;
        if (x == 20)
        {
            x = 0;
        }
        printf("PA: x = %d\n", x);
        pthread_mutex_unlock(&mutex);

    }
}
void* PROCESS2()
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        x++;
        if (x == 20)
        {
            x = 0;
        }
        printf("PB: x = %d\n", x);
        pthread_mutex_unlock(&mutex);
    }
}
void main()
```

```
21      }
22  }
23  void* PROCESS2()
24  {
25      while (1)
26      {
27          pthread_mutex_lock(&mutex);
28          x++;
29          if (x == 20)
30          {
31              x = 0;
32          }
33          printf("PB: x = %d\n", x);
34          pthread_mutex_unlock(&mutex);
35      }
36  }
37  void main()
38  {
39      pthread_mutex_init(&mutex, NULL);
40      pthread_t th1, th2;
41      pthread_create(&th1, NULL, &PROCESS1, NULL);
42      pthread_create(&th2, NULL, &PROCESS2, NULL);
43      while (1);
44  }
45
46
```

Results:

```
PA: x = 12
PA: x = 13
PA: x = 14
PA: x = 15
PA: x = 16
PA: x = 17
PA: x = 18
PB: x = 19
PB: x = 0
PB: x = 1
PB: x = 2
PB: x = 3
PB: x = 4
PB: x = 5
PB: x = 6
PB: x = 7
```

Explain result:

The program consists of two threads, `PROCESS1` and `PROCESS2`, sharing a variable `x` protected by a mutex. Both threads increment `x` and print its value in an infinite loop. If `x` reaches 20, it is reset to 0. The mutex ensures that only one thread can access the critical section at a time.

The output will likely

PA: x = 1

PB: x = 2

PA: x = 3

PB: x = 4

The key points are:

- Threads are executing concurrently.

- Mutex ensures that only one thread modifies `x` at a time, preventing data corruption.

- `x` is incremented and printed in sequence, with resets to 0 when it reaches 20.

This output showcases the synchronized access to the shared variable due to the mutex, preventing race conditions and ensuring consistent results in a multithreaded environment.

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

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
sem_t p1_5, p1_6, p2_3, p2_4, p3_5, p4_6, p5_7, p6_7;
int x1 = 1;
int x2 = 2;
int x3 = 3;
int x4 = 4;
int x5 = 5;
int x6 = 6;
int w, v, z, y, x;
int ans = 0;
void* PROCESS1()
{
    w = x1 * x2;
    printf("w = %d\n", w);
    sem_post(&p1_5);
    sem_post(&p1_6);
    sleep(1);
}
```

```
31 void* PROCESS3()
32 {
33     sem_wait(&p2_3);
34     printf("y = %d\n", y);
35     y = v * x5;
36     sem_post(&p3_5);
37     sleep(1);
38 }
39 void *PROCESS4()
40 {
41     sem_wait(&p2_4);
42     printf("z = %d\n", z);
43     z = v * x6;
44     sem_post(&p4_6);
45     sleep(1);
46 }
47 void *PROCESS5()
48 {
49     sem_wait(&p1_5);
50     sem_wait(&p3_5);
51     y = w * y;
52     printf("y = %d\n", y);
53     sem_post(&p5_7);
54     sleep(1);
55 }
56 void *PROCESS6()
57 {
58     sem_wait(&p1_6);
59     sem_wait(&p4_6);
60     z = w * z;
61     printf("z = %d\n", z);
62     sem_post(&p6_7);
63     sleep(1);
64 }
65 void* PROCESS7()
66 {
67     sem_wait(&p5_7);
```

```c
void* PROCESS7()
{
    sem_wait(&p5_7);
    sem_wait(&p6_7);
    ans = y + z;
    printf("ans = %d\n", ans);
    sleep(1);
}
void main()
{
    sem_init(&p1_5, 0, 1);
    sem_init(&p1_6, 0, 0);
    sem_init(&p2_3, 0, 0);
    sem_init(&p2_4, 0, 0);
    sem_init(&p3_5, 0, 0);
    sem_init(&p4_6, 0, 0);
    sem_init(&p5_7, 0, 0);
    sem_init(&p6_7, 0, 0);
    pthread_t th1, th2, th3, th4, th5, th6, th7;
    pthread_create(&th1, NULL, &PROCESS1, NULL);
    pthread_create(&th2, NULL, &PROCESS2, NULL);
    pthread_create(&th3, NULL, &PROCESS3, NULL);
    pthread_create(&th4, NULL, &PROCESS4, NULL);
    pthread_create(&th5, NULL, &PROCESS5, NULL);
    pthread_create(&th6, NULL, &PROCESS6, NULL);
    pthread_create(&th7, NULL, &PROCESS7, NULL);
    while (1);
}
```

Results:

```
lytuankhoa_21522225/ $ gcc -o bai5 bai5.c
lytuankhoa_21522225/ $ ./bai5
w = 2
v = 12
z = 0
z = 144
y = 0
y = 120
ans = 264
```

explain:

It demonstrates the synchronization of processes through semaphores to ensure the correct order of computations. The result ans is calculated based on the intermediate values computed by different processes. The infinite loop at the end keeps the program