

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

STT	MSSV	Họ và tên	Phân công
1	22521371	Nguyễn Thành Thọ	BT2 + BT1 ôn
2	22521190	Tô Công Quân	BT1 + BT1 ôn
3	22521448	Trần Văn Thuận	BT3
4	22521153	Lâm Hoàng Phước	BT4

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	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Chụp hình minh chứng	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Giải thích kết quả	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

5.6. BÀI TẬP ÔN TẬP

	BT 1
Trình bày cách làm	<input type="checkbox"/>
Chụp hình minh chứng	<input type="checkbox"/>
Giải thích kết quả	<input type="checkbox"/>

Tư chấm điểm: 9

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

**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: $sells \leq products \leq sells + [4 \text{ số cuối của MSSV}]$

- Cách làm:
 - Khởi tạo semaphore:
 - sem1 được khởi tạo với giá trị ban đầu là 0, điều này ngăn cản ProcessA tăng sells cho đến khi products có thể bắt đầu ở mức ban đầu.
 - sem2 được khởi tạo với giá trị ban đầu là 1190, cho phép ProcessB tăng products cho đến mức giới hạn tối đa ban đầu là 1190.
 - Cấu trúc của mỗi tiến trình:
 - ProcessA (tăng sells): Chờ đến khi có tín hiệu từ sem1, tăng sells lên 1 đơn vị, sau đó gửi tín hiệu cho sem2.
 - ProcessB (tăng products): Chờ đến khi có tín hiệu từ sem2, tăng products lên 1 đơn vị, sau đó gửi tín hiệu cho sem1
 - Quy tắc hoạt động của semaphore:
 - sem1 và sem2 đảm bảo rằng products không vượt quá $sells + 1190$ và sells không vượt quá products
- Minh chứng:

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

int sells = 0, products = 0;
sem_t sem1, sem2;

void *ProcessA(void *mess)
{
    while (1)
    {
        sem_wait(&sem1);
        sells++;
        printf("Sells = %d\n", sells);
        sem_post(&sem2);
    }
}

void *ProcessB(void *mess)
```

```
{
    while (1)
    {
        sem_wait(&sem2);
        products++;
        printf("Products = %d\n", products);
        sem_post(&sem1);
    }
}

int main()
{
    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, 1190);
    pthread_t pA, pB;

    pthread_create(&pA, NULL, &ProcessA, NULL);
    pthread_create(&pB, NULL, &ProcessB, NULL);

    while(1) {};
    return 0;
}
```

```
thaont@ubuntu: ~/Documents
thaont@ubuntu:~/Documents$ gcc Bai1.c -o Bai1 -lpthread -lrt
thaont@ubuntu:~/Documents$ ./Bai1
Products = 1
Products = 2
Products = 3
Products = 4
Products = 5
Sells = 1
Sells = 2
Sells = 3
Sells = 4
Sells = 5
Products = 6
Products = 7
Products = 8
Products = 9
Products = 10
Products = 11
Products = 12
Products = 13
Products = 14
Products = 15
Products = 16
Products = 17
```

- Giải thích:

- Ban đầu, ProcessB thực hiện liên tiếp việc tăng products từ 1 đến 5 mà không bị chặn bởi ProcessA vì sem2 có giá trị ban đầu đủ lớn.
- Khi ProcessB tăng products đến giá trị 5, ProcessA nhận được tín hiệu từ sem1 và bắt đầu thực hiện việc tăng sells từ 1 đến 5, mỗi lần tăng sells, ProcessA lại gửi tín hiệu cho sem2 để ProcessB có thể tiếp tục.
- Sau khi sells đạt giá trị 5, ProcessB lại tiếp tục tăng products từ 6 trở lên, do sem2 vẫn còn nhiều tín hiệu được khởi tạo ban đầu (tối đa 1190)

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.

- Cách làm:

- Khi chưa đồng bộ:
 - Chương trình tạo ra hai thread để thực hiện các thao tác trên mảng a: một thread thêm số nguyên ngẫu nhiên vào mảng và một thread lấy ra một phần tử ngẫu nhiên từ mảng. Tuy nhiên, chương trình này chưa được đồng bộ hóa, dẫn đến các vấn đề tiềm ẩn về truy cập đồng thời
- Khi được đồng bộ:
 - Khai báo và khởi tạo:

- `sem_t sem1, sem2`: Hai semaphore để điều phối việc thêm và lấy phần tử từ mảng `a`
- `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`: Mutex để bảo vệ các vùng mã truy cập vào mảng `a` và biến `size`
- Thread ProcessA (thêm phần tử vào mảng):
 - `sem_wait(&sem2)`: Đợi cho đến khi có tín hiệu từ `sem2`, đảm bảo rằng có chỗ trống để thêm phần tử vào mảng.
 - `pthread_mutex_lock(&mutex)`: Khóa mutex để đảm bảo rằng thao tác thêm phần tử vào mảng được thực hiện một cách an toàn.
 - Thêm một số ngẫu nhiên vào cuối mảng `a` và tăng biến `size`.
 - In ra số phần tử trong mảng sau khi thêm phần tử mới.
 - `sem_post(&sem1)`: Gửi tín hiệu cho `sem1` để báo rằng một phần tử đã được thêm vào mảng.
 - `pthread_mutex_unlock(&mutex)`: Mở khóa mutex
- Thread ProcessB (lấy phần tử ra từ mảng):
 - `sem_wait(&sem1)`: Đợi cho đến khi có tín hiệu từ `sem1`, đảm bảo rằng có ít nhất một phần tử trong mảng để lấy ra.
 - `pthread_mutex_lock(&mutex)`: Khóa mutex để đảm bảo rằng thao tác lấy phần tử ra khỏi mảng được thực hiện một cách an toàn.
 - Kiểm tra nếu mảng `a` rỗng, in ra "Nothing in array".
 - Nếu mảng không rỗng, lấy ra một phần tử ngẫu nhiên từ mảng.
 - Gọi hàm `Arrange` để xóa phần tử đó và điều chỉnh mảng.
 - In ra số phần tử trong mảng sau khi lấy phần tử ra.

- `sem_post(&sem2)`: Gửi tín hiệu cho `sem2` để báo rằng một phần tử đã được lấy ra và có chỗ trống để thêm phần tử mới.
- `pthread_mutex_unlock(&mutex)`: Mở khóa mutex

- Minh chứng:

○ Khi chưa đồng bộ:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

int *a;
int n, size = 0;

int rand_range(int min, int max)
{
    return min + (int)(rand()*(max-min+1.0)/(1.0+RAND_MAX));
}

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

void *ProcessA(void *mess)
{
    while(1)
    {
        a[size] = rand_range(0, 1000);
        size++;
        printf("So phan tu cua a sau khi them so nguyen %d: %d\n", a[size-1], size);
    }
}

void *ProcessB(void *mess)
```

```
{
    while(1)
    {
        if (size == 0)
            printf("Nothing in array \n");
        else
        {
            int r = rand_range(0, 1000) % size;
            printf("So phan tu cua a sau khi lay ra so nguyen %d: ", a[r]);
            Arrange(a, r);
            printf("%d\n", size);
        }
    }
}

int main()
{
    srand((int)time(0));

    printf("Nhap so phan tu: ");
    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;
}
```



```
thaont@ubuntu: ~/Documents
thaont@ubuntu:~/Documents$ gcc Bai2a.c -o Bai2a -lpthread
thaont@ubuntu:~/Documents$ ./Bai2a
Nhap so phan tu: 3
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
Nothing in array
So phan tu cua a sau khi them so nguyen 988: 1
So phan tu cua a sau khi them so nguyen 902: 2
So phan tu cua a sau khi them so nguyen 826: 3
So phan tu cua a sau khi them so nguyen 895: 4
So phan tu cua a sau khi them so nguyen 142: 5
So phan tu cua a sau khi them so nguyen 374: 6

So phan tu cua a sau khi them so nguyen 895: 4
So phan tu cua a sau khi them so nguyen 142: 5
So phan tu cua a sau khi them so nguyen 374: 6
So phan tu cua a sau khi lay ra so nguyen 988: So phan tu cua a sau khi them so
nguyen 720: 7
So phan tu cua a sau khi them so nguyen 243: 7
So phan tu cua a sau khi them so nguyen 860: 8
So phan tu cua a sau khi them so nguyen 131: 9
So phan tu cua a sau khi them so nguyen 871: 10
So phan tu cua a sau khi them so nguyen 855: 11
So phan tu cua a sau khi them so nguyen 575: 12
So phan tu cua a sau khi them so nguyen 221: 13
So phan tu cua a sau khi them so nguyen 822: 14
So phan tu cua a sau khi them so nguyen 101: 15
So phan tu cua a sau khi them so nguyen 657: 16
So phan tu cua a sau khi them so nguyen 100: 17
So phan tu cua a sau khi them so nguyen 568: 18
So phan tu cua a sau khi them so nguyen 832: 19
So phan tu cua a sau khi them so nguyen 93: 20
So phan tu cua a sau khi them so nguyen 176: 21
So phan tu cua a sau khi them so nguyen 669: 22
6
So phan tu cua a sau khi lay ra so nguyen 902: 22
So phan tu cua a sau khi lay ra so nguyen 93: 21
```

- Khi được đồng bộ:

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

int *a;
int n, size = 0;
sem_t sem1, sem2;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int rand_range(int min, int max)
{
    return min + (int)(rand()*(max-min+1.0)/(1.0+RAND_MAX));
}

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

void *ProcessA(void *mess)
{
    while(1)
    {
        sem_wait(&sem2);
        pthread_mutex_lock(&mutex);

        a[size] = rand_range(0, 1000);
        size++;
        printf("So phan tu cua a sau khi them so nguyen %d: %d\n", a[size-1], size);

        sem_post(&sem1);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
}

void *ProcessB(void *mess)
{
    while(1)
    {
        sem_wait(&sem1);
        pthread_mutex_lock(&mutex);

        if (size == 0)
            printf("Nothing in array \n");
        else
        {
            int r = rand_range(0, 1000) % size;
            printf("So phan tu cua a sau khi lay ra so nguyen %d: ", a[r]);
            Arrange(a, r);
            printf("%d\n", size);

            sem_post(&sem2);
            pthread_mutex_unlock(&mutex);
        }
    }
}

int main()
{
    srand((int)time(0));

    printf("Nhap so phan tu: ");
    scanf("%d", &n);
    a = (int *)malloc(n*sizeof(int));

    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, n);
    pthread_mutex_init(&mutex, NULL);

    pthread_t pA, pB;
    pthread_create(&pA, NULL, &ProcessA, NULL);
    pthread_create(&pB, NULL, &ProcessB, NULL);

    while(1) {};
    return 0;
}
```

```
thaont@ubuntu: ~/Documents
thaont@ubuntu:~/Documents$ gcc Bai2b.c -o Bai2b -lpthread -lrt
thaont@ubuntu:~/Documents$ ./Bai2b
Nhap so phan tu: 3
So phan tu cua a sau khi them so nguyen 205: 1
So phan tu cua a sau khi them so nguyen 256: 2
So phan tu cua a sau khi them so nguyen 98: 3
So phan tu cua a sau khi lay ra so nguyen 205: 2
So phan tu cua a sau khi lay ra so nguyen 256: 1
So phan tu cua a sau khi lay ra so nguyen 98: 0
So phan tu cua a sau khi them so nguyen 83: 1
So phan tu cua a sau khi them so nguyen 168: 2
So phan tu cua a sau khi them so nguyen 88: 3
So phan tu cua a sau khi lay ra so nguyen 83: 2
So phan tu cua a sau khi lay ra so nguyen 168: 1
So phan tu cua a sau khi lay ra so nguyen 88: 0
So phan tu cua a sau khi them so nguyen 948: 1
So phan tu cua a sau khi them so nguyen 618: 2
So phan tu cua a sau khi them so nguyen 906: 3
So phan tu cua a sau khi lay ra so nguyen 618: 2
So phan tu cua a sau khi lay ra so nguyen 948: 1
So phan tu cua a sau khi lay ra so nguyen 906: 0
So phan tu cua a sau khi them so nguyen 836: 1
So phan tu cua a sau khi them so nguyen 626: 2
So phan tu cua a sau khi them so nguyen 919: 3
```

- Giải thích:

- Khi chưa đồng bộ:
 - Truy cập và thay đổi biến size đồng thời, dẫn đến giá trị không chính xác hoặc gây ra lỗi bộ nhớ.
 - Truy cập và thay đổi mảng a đồng thời, dẫn đến việc đọc hoặc ghi dữ liệu không chính xác, có thể gây ra lỗi chương trình
- Khi được đồng bộ:
 - Kết quả trên cho thấy việc sử dụng semaphore và mutex đã giúp chương trình hoạt động đồng bộ và chính xác, tránh được các lỗi liên quan đến truy cập đồng thời vào dữ liệu. Mỗi lần ProcessA thêm một phần tử vào mảng, số lượng phần tử trong mảng tăng lên, và mỗi lần ProcessB lấy ra một phần tử, số lượng phần tử giảm xuống. Semaphore điều phối hoạt động thêm và lấy phần tử, trong khi mutex bảo vệ dữ liệu khỏi các xung đột truy cập

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

int x = 0;	
PROCESS A	PROCESS B
<pre>processA() { while(1){ x = x + 1; if (x == 20) x = 0; print(x); } }</pre>	<pre>processB() { while(1){ x = x + 1; if (x == 20) x = 0; print(x); } }</pre>

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

- Cách làm:

- Biến x được khởi tạo bằng 0 và được chia sẻ giữa hai luồng
- Hàm ProcessA:
 - Tăng giá trị của x lên 1.
 - Nếu x đạt giá trị 20, đặt lại x bằng 0.
 - In ra giá trị của x kèm theo chuỗi "Process A: "
- Hàm ProcessB:
 - Tăng giá trị của x lên 1.
 - Nếu x đạt giá trị 20, đặt lại x bằng 0.
 - In ra giá trị của x kèm theo chuỗi "Process B: "

- Minh chứng:

```
#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: %d\n", x);
    }
}

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

int main()
{
    pthread_t pA, pB;
    pthread_create(&pA, NULL, &ProcessA, NULL);
    pthread_create(&pB, NULL, &ProcessB, NULL);

    while(1) {}
    return 0;
}
```

```
thaont@ubuntu: ~/Documents
thaont@ubuntu:~/Documents$ gcc Bai3.c -o Bai3 -lpthread -lrt
thaont@ubuntu:~/Documents$ ./Bai3
Process A: 1
Process A: 2
Process A: 3
Process A: 4
Process A: 5
Process A: 6
Process A: 8
Process B: 7
Process B: 10
Process B: 11
Process B: 12
Process B: 13
Process B: 14
Process B: 15
Process B: 16
Process B: 17
Process B: 18
Process B: 19
Process B: 0
Process B: 1
Process B: 2
Process B: 3
```

- Giải thích:

- Khi hai luồng chạy đồng thời mà không có bất kỳ cơ chế đồng bộ nào (như mutex), có thể xảy ra xung đột truy cập (race condition).
- Điều này dẫn đến việc giá trị của x có thể bị ghi đè một cách không đồng bộ, dẫn đến các giá trị không liên tục và xen kẽ giữa ProcessA và ProcessB trong kết quả in ra.
- Ví dụ: Process A: 6 và Process B: 7 gần như liên tiếp, điều này cho thấy ProcessA và ProcessB đang chạy gần như cùng lúc

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

- Cách làm:

- Khai báo và khởi tạo mutex:
 - pthread_mutex_t mutex: Khai báo một biến mutex toàn cục để sử dụng cho việc đồng bộ hóa.
 - PTHREAD_MUTEX_INITIALIZER: Khởi tạo mutex với giá trị mặc định
- Bảo vệ đoạn mã quan trọng bằng mutex:

- Trong cả hai hàm ProcessA và ProcessB, chúng ta sử dụng `pthread_mutex_lock` để bắt đầu vùng mã quan trọng và `pthread_mutex_unlock` để kết thúc vùng mã quan trọng

- Minh chứng:

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

int x = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

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

void *ProcessB(void *mess)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        x++;
        if (x == 20)
            x = 0;
        printf("Process B: %d\n", x);
        pthread_mutex_unlock(&mutex);
    }
}

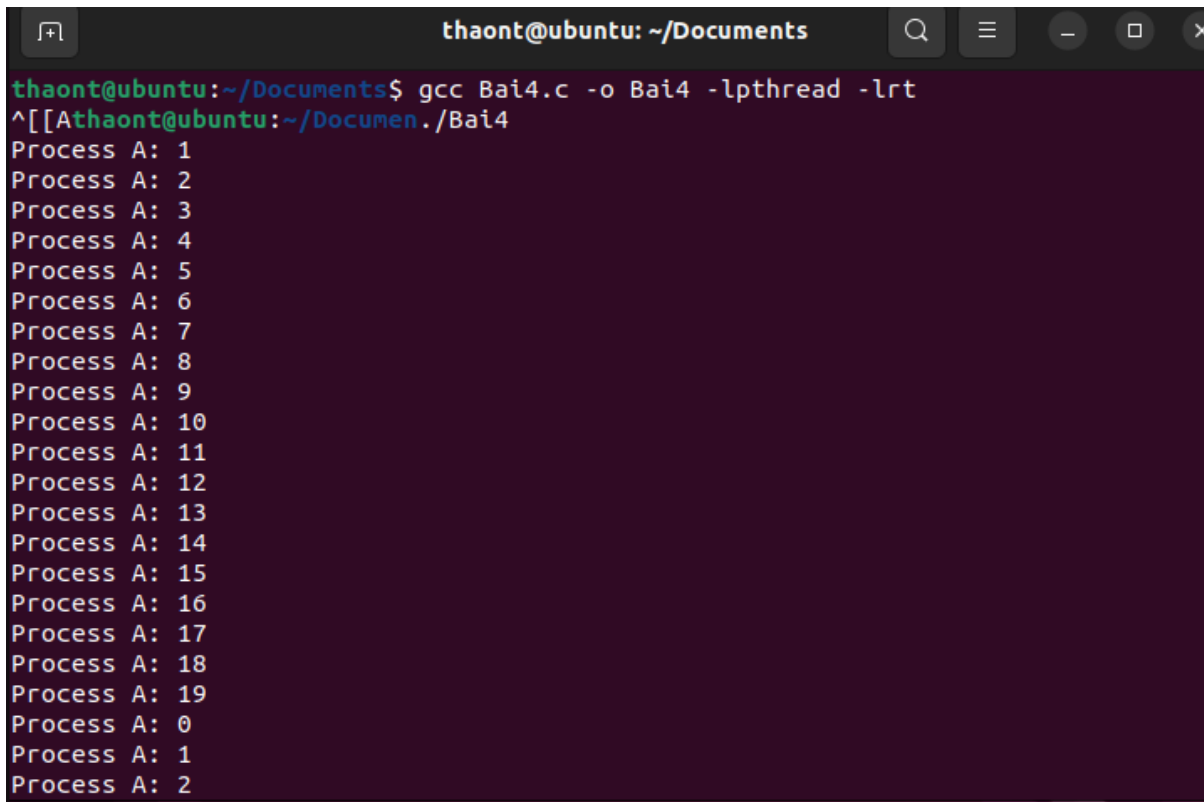
int main()
{
    pthread_mutex_init(&mutex, NULL);

    pthread_t pA, pB;
```

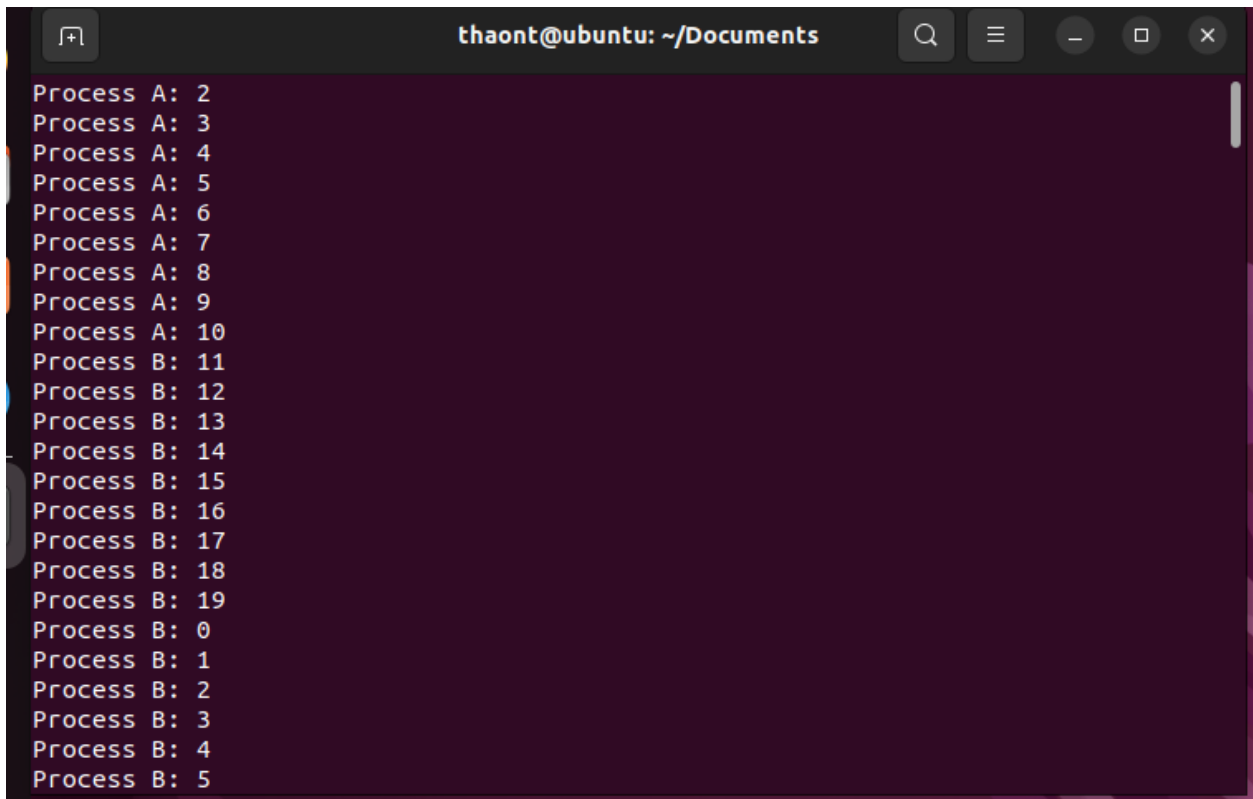


```
pthread_create(&pA, NULL, &ProcessA, NULL);
pthread_create(&pB, NULL, &ProcessB, NULL);

while(1) {}
return 0;
}
```



```
thaont@ubuntu:~/Documents$ gcc Bai4.c -o Bai4 -lpthread -lrt
^[[Athaont@ubuntu:~/Documents ./Bai4
Process A: 1
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 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 A: 2
```



```
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 B: 11
Process B: 12
Process B: 13
Process B: 14
Process B: 15
Process B: 16
Process B: 17
Process B: 18
Process B: 19
Process B: 0
Process B: 1
Process B: 2
Process B: 3
Process B: 4
Process B: 5
```

- Giải thích:

- ProcessA tăng và in giá trị của x nhiều lần trước khi ProcessB có cơ hội thực thi. Điều này có thể là do lịch trình của bộ lập lịch của hệ điều hành, nơi luồng ProcessA có thể nhận được nhiều thời gian CPU hơn
- Sau khi ProcessA thực hiện vài lần, ProcessB bắt đầu thực thi và tăng giá trị của x. Chúng ta có thể thấy rằng các giá trị in ra từ ProcessB xen kẽ với các giá trị in ra từ ProcessA
- Cả hai luồng đều kiểm tra và đặt lại giá trị của x khi x đạt giá trị 20. Điều này hoạt động chính xác nhờ vào mutex, đảm bảo rằng không có sự xung đột khi hai luồng kiểm tra và đặt lại x

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) \rightarrow (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

- Cách làm:

- Sử dụng các mutex để đảm bảo thứ tự thực thi các đoạn mã như yêu cầu. Mỗi đoạn mã sẽ có một mutex tương ứng và mutex này chỉ được mở khóa khi đoạn mã trước đó đã hoàn thành
- Biến mutex[7]: Dùng để đồng bộ hóa giữa các thread. Mỗi mutex kiểm soát một phần của công việc cần được hoàn thành trước khi tiếp tục phần tiếp theo.
- Hàm ProcessA: Tính $w = x1 * x2$ và mở khóa mutex[1] để cho phép ProcessE và ProcessF có thể chạy khi cần.
- Hàm ProcessB: Tính $v = x3 * x4$ và mở khóa mutex[2] để cho phép ProcessC và ProcessD chạy.
- Hàm ProcessC: Đợi mutex[2] (tức là đợi ProcessB hoàn thành), sau đó tính $y = v * x5$ và mở khóa mutex[3] để cho phép ProcessE chạy.

- Hàm ProcessD: Đợi mutex[2] (tức là đợi ProcessB hoàn thành), sau đó tính $z = v * x6$ và mở khóa mutex[4] để cho phép ProcessF chạy.
- Hàm ProcessE: Đợi mutex[1] và mutex[3] (tức là đợi ProcessA và ProcessC hoàn thành), sau đó tính $y = w * y$ và mở khóa mutex[5] để cho phép ProcessG chạy.
- Hàm ProcessF: Đợi mutex[1] và mutex[4] (tức là đợi ProcessA và ProcessD hoàn thành), sau đó tính $z = w * z$ và mở khóa mutex[6] để cho phép ProcessG chạy.
- Hàm ProcessG: Đợi mutex[5] và mutex[6] (tức là đợi ProcessE và ProcessF hoàn thành), sau đó tính $res = y + z$.
- Phần main: Khởi tạo mutex, khóa chúng ban đầu. Nhận đầu vào, khởi tạo các giá trị, tạo các thread, và cuối cùng đợi các thread kết thúc. Sau khi các thread kết thúc, phá hủy các mutex

- Minh chứng:

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

pthread_mutex_t mutex[7];
int x1, x2, x3, x4, x5, x6;
int w, v, y, z, res;

void *ProcessA(void *mess)
{
    w = x1 * x2;
    printf("Process A: w = x1 * x2 = %d\n", w);
    pthread_mutex_unlock(&mutex[1]);
}

void *ProcessB(void *mess)
{
    v = x3 * x4;
    printf("Process B: v = x3 * x4 = %d\n", v);
    pthread_mutex_unlock(&mutex[2]);
}

void *ProcessC(void *mess)
{
```

```
pthread_mutex_lock(&mutex[2]);
pthread_mutex_unlock(&mutex[2]);

y = v * x5;
printf("Process C: y = v * x5 = %d\n", y);
pthread_mutex_unlock(&mutex[3]);
}

void *ProcessD(void *mess)
{
    pthread_mutex_lock(&mutex[2]);
    pthread_mutex_unlock(&mutex[2]);

    z = v * x6;
    printf("Process D: z = v * x6 = %d\n", z);
    pthread_mutex_unlock(&mutex[4]);
}

void *ProcessE(void *mess)
{
    pthread_mutex_lock(&mutex[1]);
    pthread_mutex_lock(&mutex[3]);
    pthread_mutex_unlock(&mutex[1]);

    y = w * y;
    printf("Process E: y = w * y = %d\n", y);
    pthread_mutex_unlock(&mutex[5]);
}

void *ProcessF(void *mess)
{
    pthread_mutex_lock(&mutex[1]);
    pthread_mutex_lock(&mutex[4]);
    pthread_mutex_unlock(&mutex[1]);

    z = w * z;
    printf("Process F: z = w * z = %d\n", z);
    pthread_mutex_unlock(&mutex[6]);
}

void *ProcessG(void *mess)
{
    pthread_mutex_lock(&mutex[6]);
    pthread_mutex_lock(&mutex[5]);
```

```
    res = y + z;
    printf("Process G: res = y + z = %d\n", res);
}

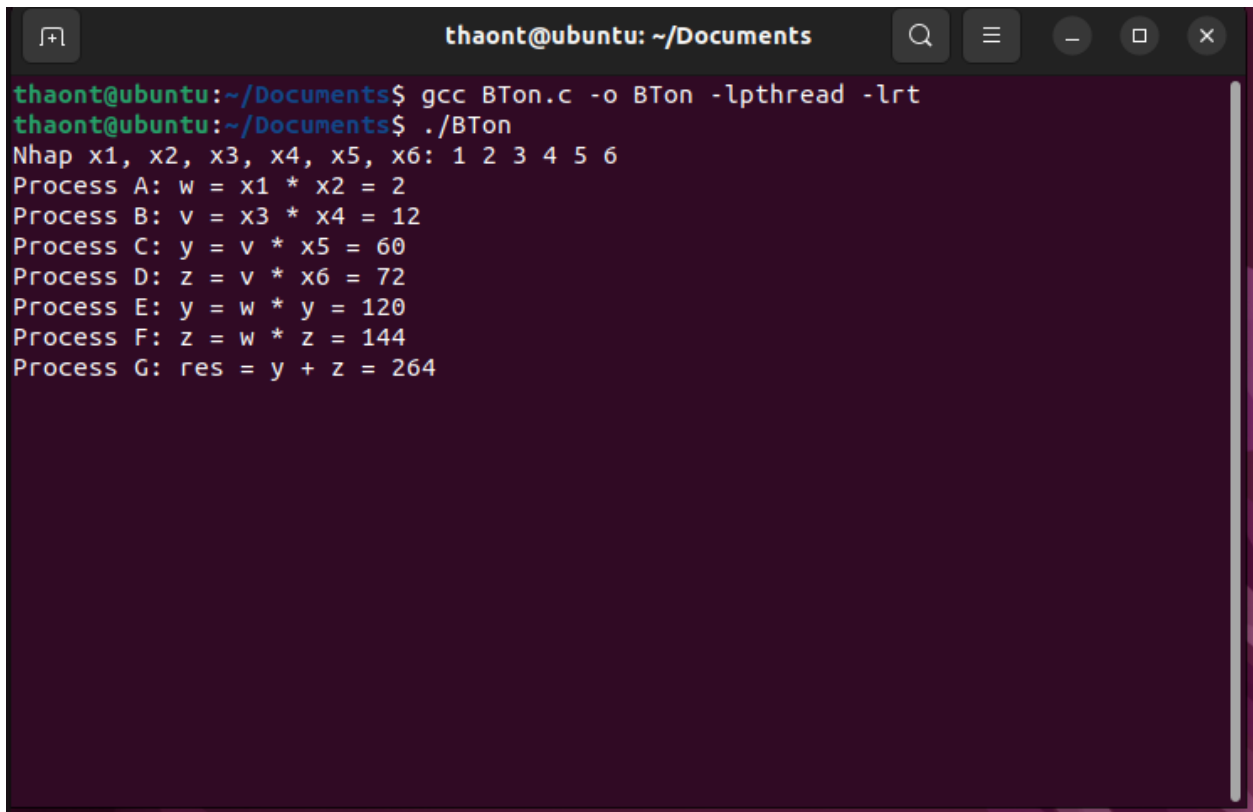
int main()
{
    for (int i = 1; i <= 6; i++)
    {
        pthread_mutex_init(&mutex[i], NULL);
        pthread_mutex_lock(&mutex[i]);
    }

    printf("Nhap x1, x2, x3, x4, x5, x6: ");
    scanf("%d %d %d %d %d %d", &x1, &x2, &x3, &x4, &x5, &x6);
    w = v = y = z = 0;

    pthread_t pA, pB, pC, pD, pE, pF, pG;

    pthread_create(&pF, NULL, &ProcessA, NULL);
    pthread_create(&pG, NULL, &ProcessB, NULL);
    pthread_create(&pE, NULL, &ProcessC, NULL);
    pthread_create(&pA, NULL, &ProcessD, NULL);
    pthread_create(&pB, NULL, &ProcessE, NULL);
    pthread_create(&pC, NULL, &ProcessF, NULL);
    pthread_create(&pD, NULL, &ProcessG, NULL);

    while(1) {};
    return 0;
}
```



```
thaont@ubuntu: ~/Documents
thaont@ubuntu:~/Documents$ gcc BTon.c -o BTon -lpthread -lrt
thaont@ubuntu:~/Documents$ ./BTon
Nhap x1, x2, x3, x4, x5, x6: 1 2 3 4 5 6
Process A: w = x1 * x2 = 2
Process B: v = x3 * x4 = 12
Process C: y = v * x5 = 60
Process D: z = v * x6 = 72
Process E: y = w * y = 120
Process F: z = w * z = 144
Process G: res = y + z = 264
```

- Giải thích:

- Input: Nhập x1, x2, x3, x4, x5, x6 lần lượt bằng 1, 2, 3, 4, 5, 6
- Tính toán và đồng bộ hóa:
- Process A: $w = x1 * x2$

Giá trị w được tính là 2. Mutex[1] được mở khóa để các bước tiếp theo có thể tiến hành

- Process B: $v = x3 * x4$

Giá trị v được tính là 12. Mutex[2] được mở khóa để các bước tiếp theo có thể tiến hành

- Process C: $y = v * x5$

Giá trị y được tính là 60 (chỉ thực hiện sau khi v được tính). Mutex[3] được mở khóa để Process E có thể tiến hành

- Process D: $z = v * x6$

Giá trị z được tính là 72 (chỉ thực hiện sau khi v được tính). Mutex[4] được mở khóa để Process F có thể tiến hành

- Process E: $y = w * y$

Giá trị y được tính lại là 120 (chỉ thực hiện sau khi w và y được tính).

Mutex[5] được mở khóa để Process G có thể tiến hành

- Process F: $z = w * z$

Giá trị z được tính lại là 144 (chỉ thực hiện sau khi w và z được tính).

Mutex[6] được mở khóa để Process G có thể tiến hành

- Process G: $res = y + z$

Giá trị res cuối cùng được tính là 264 (chỉ thực hiện sau khi y và z được tính)