

Name: Nguyễn Trần Trung Nguyên

ID: 21522393

Class: IT007.N22.ATCL.1

OPERATING SYSTEM LAB X'S REPORT

SUMMARY

Task		Status	Page
Section 1.5	Ex 1	Hoàn thành	1
	Ex 2	Done	3
	Ex 3	Undone	
	Task name 4	Undone	
	Task name 5	Undone	
	Task name 6	Done	4
	Task name 7	Done	7
...	...		
	...		

Self-scores:

Note: Export file to **PDF and name the file by following format:
Student ID_LABx.pdf*

Section 5.4

Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: $\text{sells} \leq \text{products} \leq \text{sells} + [2 \text{ số cuối của MSSV} + 10]$

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

int sells=0, products=0;
sem_t sem,sem2;
int req = 93 + 10;      //2 so cuoi MSSV + 10

void* processA(void* mess)
{
    while(1)
    {
        sem_wait(&sem);
        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(&sem);
    }
}

int main()
{
    sem_init(&sem,0,0);
    sem_init(&sem2,0,req);
    pthread_t pA,pB;
    pthread_create(&pA,NULL,&processA,NULL);
    pthread_create(&pB,NULL,&processB,NULL);
    while(1){}
    return 0;
}
```

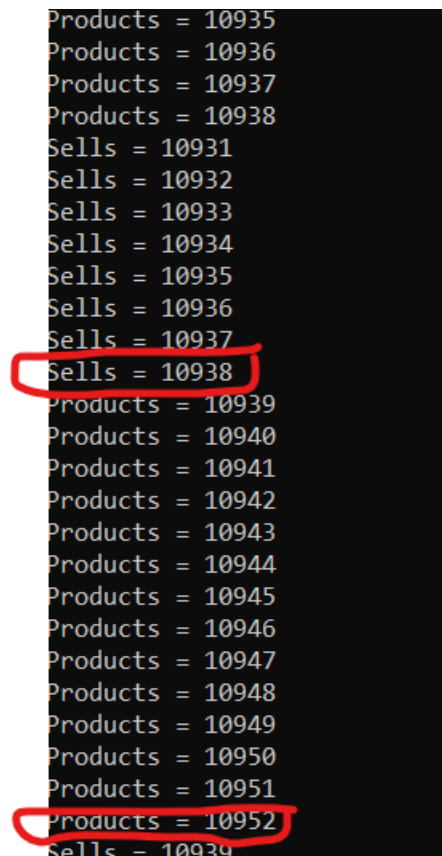
Sử dụng 2 biến semaphore để quản lý các cận của dữ liệu.

Sem để quản lý $\text{sells} \leq \text{products}$

Sem2 để quản lý $\text{products} \leq \text{sells} + [2 \text{ số cuối MSSV} + 10]$ (ở đây là $93 + 10 = 103$)

Ta gán $\text{sem} = 0$ để sells luôn $\leq \text{products}$ vì processA thực hiện lệnh $\text{sem_wait}(\&\text{sem})$ sẽ giảm giá trị của sem đi 1 và khi thực hiện lệnh này mà $\text{sem} < 0$ thì process A sẽ sleep. Còn processB thì sau khi thực hiện xong sẽ wake up processA bằng $\text{sem_post}(\&\text{sem})$. Lệnh $\text{sem_post}(\&\text{sem})$ sẽ tăng sem lên 1 đơn vị và wake up processA nếu A đang sleep.

Ta gán $\text{sem2} = 103$ để products luôn $\leq \text{sells} + 103$ vì processB thực hiện lệnh $\text{sem_wait}(\&\text{sem2})$ sẽ giảm giá trị của sem2 đi 1 và khi thực hiện lệnh này mà $\text{sem2} < 0$ thì process B sẽ sleep. Còn processA thì sau khi thực hiện xong sẽ wake up processB bằng $\text{sem_post}(\&\text{sem2})$. Lệnh $\text{sem_post}(\&\text{sem2})$ sẽ tăng sem2 lên 1 đơn vị và wake up processB nếu B đang sleep. Do giá trị ban đầu của sem2 là 103 nên số lượng products sẽ luôn $\leq \text{sells} + 103$.



```
Products = 10935
Products = 10936
Products = 10937
Products = 10938
Sells = 10931
Sells = 10932
Sells = 10933
Sells = 10934
Sells = 10935
Sells = 10936
Sells = 10937
Sells = 10938
Products = 10939
Products = 10940
Products = 10941
Products = 10942
Products = 10943
Products = 10944
Products = 10945
Products = 10946
Products = 10947
Products = 10948
Products = 10949
Products = 10950
Products = 10951
Products = 10952
Sells = 10939
```

Dựa theo kết quả khi chạy code ta có thể thấy rằng kết quả luôn thỏa điều kiện $\text{sells} \leq \text{products} \leq \text{sells} + 103$

Ví dụ $\text{sells} = 10938 \leq \text{products} = 10952 \leq \text{sells} + 103 = 11041$

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int a[100];
int n=0;
int i=0;
void* processA()
{
    while(1)
    {
        int randomNum= rand() % 10;    // random trong khoảng [0;9]
        a[i]=randomNum; // Them vao cuoi array
        printf("\nPhan tu them vao a[%d]: %d",i,a[i]);
        i++;
        printf("\nMang a co %d phan tu!\n",i); // So luong phan tu sau khi them vao
        for (int j=0; j<i;j++)
        {
            printf("a[%d]: %d\t",j,a[j]);
        }
    }
}
void* processB()
{
    while(1)
    {
        i--; //Lay ra tu cuoi array
        if(i<0)
        {
            printf("\nNothing in array a\n");
        }
        else
        {
            printf("\nPhan tu lay ra a[%d]: %d",i,a[i]);
            printf("\nMang a co %d phan tu!\n",i); // So luong phan tu sau khi lay ra
            for (int j=0; j<i;j++)
            {
                printf("a[%d]: %d\t",j,a[j]);
            }
        }
    }
}
```

```

int main()
{
    do
    {
        printf("Nhap so luong phan tu cua mang trong khoang [1;100]: ");
        scanf("%d",&n);
    }while(n<=0||n>100);

    pthread_t pA,pB;
    pthread_create(&pA,NULL,&processA,NULL);
    pthread_create(&pB,NULL,&processB,NULL);
    while(1){}
    return 0;
}

```

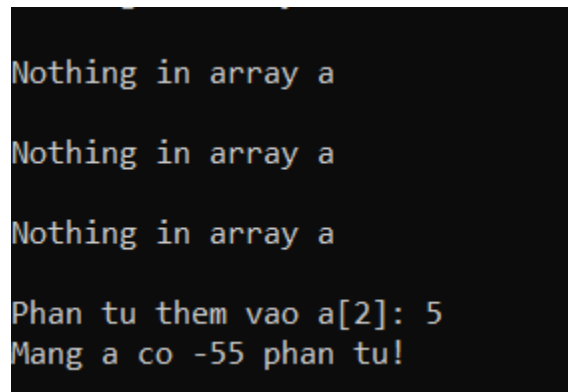
Ta có 2 process A và B chạy song song với nhau

Process A tạo 1 số nguyên ngẫu nhiên và thêm vào cuối array đồng thời xuất ra màn hình số phần tử có trong mảng.

Process B lấy 1 số nguyên từ cuối array và xuất ra màn hình số phần tử đang có trong mảng.

Chạy thử chương trình:

Chương trình khi chưa đồng bộ hóa xuất hiện vài lỗi:



```

Nothing in array a
Nothing in array a
Nothing in array a
Phan tu them vao a[2]: 5
Mang a co -55 phan tu!

```

Đầu tiên là trong array không có gì nhưng vẫn liên tục lấy 1 số nguyên từ array từ đó dẫn đến khi thêm vào sẽ thêm vào vị trí âm và có số phần tử là số âm.

```

Phan tu them vao a[1]: 7
Mang a co 2 phan tu!
a[0]: 6 a[1]: 7
Nothing in array a

```

Xuất hiện đụng độ giữa 2 tiểu trình đó là khi process A đang thực hiện chưa xong thì process B cũng thực hiện dẫn đến thay đổi dữ liệu chung từ đó khiến process A thực hiện sai.

Code sau khi đồng bộ:

```

#include <semaphore.h>
int a[100];
int n=0;          // n: so luong phan tu cua mang
int i=0;          // i: vi tri hien tai trong mang
sem_t sem,sem2;  // sem: control i>=0
                  // sem2: control i<=n
pthread_mutex_t mutex;
void* processA()
{
    while(1)
    {
        sem_wait(&sem2);          // Doi i<=n
        pthread_mutex_lock(&mutex);
        int randomNum= rand() % 10;    // random trong khoang [0;9]
        a[i]=randomNum; // Them vao cuoi array
        printf("\nPhan tu them vao a[%d]: %d",i,a[i]);
        i++;
        printf("\nMang a co %d phan tu!\n",i); // So luong phan tu sau khi them vao
        for (int j=0; j<i;j++)
        {
            printf("a[%d]: %d\t",j,a[j]);
        }
        pthread_mutex_unlock(&mutex);
        sem_post(&sem); // Thong bao them 1 phan tu vao mang
    }
}

void* processB()
{
    while(1)
    {
        if(i<=0)
        {
            printf("\nNothing in array a\n");
        }
        sem_wait(&sem); // Doi cho den khi co mot phan nao do co trong mang
        pthread_mutex_lock(&mutex);
        i--; //Lay ra tu cuoi array
        printf("\nPhan tu lay ra a[%d]: %d",i,a[i]);
        printf("\nMang a co %d phan tu!\n",i); // So luong phan tu sau khi lay ra
        for (int j=0; j<i;j++)
        {
            printf("a[%d]: %d\t",j,a[j]);
        }
        pthread_mutex_unlock(&mutex);
        sem_post(&sem2); // Thong bao da lay ra 1 phan tu
    }
}

```

```

int main()
{
    do
    {
        printf("Nhap so luong phan tu cua mang trong khoang [1;100]: ");
        scanf("%d",&n);
    }while(n<=0||n>100);
    sem_init(&sem,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;
}

```

Sử dụng 1 biến mutex và 2 biến semaphore.

2 biến semaphore dùng để quản lý biến i nằm trong khoảng [0;n]

Biến mutex dùng để đảm bảo rằng chỉ có 1 process A hoặc B được tác động lên biến i vào 1 thời điểm.

Process A tăng biến i lên → ta cần lệnh `sem_wait(&sem2)` để đảm bảo nó không > n, sau khi thực hiện xong process A cần thông báo cho process B biết là đã có phần tử trong mảng → sử dụng `sem_post(&sem)`

Process B giảm biến i xuống → ta cần lệnh `sem_wait(&sem)` để đảm bảo nó không < 0, sau khi thực hiện xong process B cần thông báo cho process A biết là đã lấy 1 phần tử ra khỏi mảng → sử dụng `sem_post(&sem2)`


```

nguyen8a@DESKTOP-BKCEOEA: /mnt/d/HOCTAP/HỆ ĐIỀU HÀNH/TH/Lab5/code
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7 a[6]: 2 a[7]: 3
Phan tu them vao a[8]: 6
Mang a co 9 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7 a[6]: 2 a[7]: 3 a[8]: 6
Phan tu them vao a[9]: 3
Mang a co 10 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7 a[6]: 2 a[7]: 3 a[8]: 6 a[9]: 3
Phan tu lay ra a[9]: 3
Mang a co 9 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7 a[6]: 2 a[7]: 3 a[8]: 6
Phan tu lay ra a[8]: 6
Mang a co 8 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7 a[6]: 2 a[7]: 3
Phan tu lay ra a[7]: 3
Mang a co 7 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7 a[6]: 2
Phan tu lay ra a[6]: 2
Mang a co 6 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6 a[5]: 7
Phan tu lay ra a[5]: 7
Mang a co 5 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5 a[4]: 6
Phan tu lay ra a[4]: 6
Mang a co 4 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1 a[3]: 5
Phan tu lay ra a[3]: 5
Mang a co 3 phan tu!
a[0]: 4 a[1]: 2 a[2]: 1
Phan tu lay ra a[2]: 1
Mang a co 2 phan tu!
a[0]: 4 a[1]: 2
Phan tu lay ra a[1]: 2
Mang a co 1 phan tu!
a[0]: 4
Phan tu lay ra a[0]: 4
Mang a co 0 phan tu!

Nothing in array a

```

Kết quả sau khi chạy code thỏa yêu cầu. Không được lấy ra khi không có phần tử nào trong mảng, không được thêm vào khi mảng đầy.

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

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

12



<pre>while(1){ x = x + 1; if (x == 20) x = 0; print(x); }</pre>	<pre>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

Code:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
int x=0;
void* processA()
{
    while(1)
    {
        x = x+1;
        if (x==20) x=0;
        printf("\nProcess a: %d",x);
    }
}

void* processB()
{
    while(1)
    {
        x = x+1;
        if (x==20) x=0;
        printf("\nProcess b: %d",x);
    }
}

int main()
{
    pthread_t pA,pB;
    pthread_create(&pA,NULL,&processA,NULL);
    pthread_create(&pB,NULL,&processB,NULL);
    while(1){}
    return 0;
}

```

Sau khi chạy:

```

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

```

Có thể thấy rõ là khi chuyển từ tiểu trình A sang tiểu trình B biến x bị thay đổi không theo ta mong muốn (biến x từ 1 chuyển thành 9 rồi sau đó lại thành 3)

Lí do: trong khi thread này thực hiện đợi I/O in ra màn hình thì thread khác thực hiện thay đổi biến → kết quả in ra không mong muốn vì cả 2 đều cùng vào critical section

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

Code:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
int x=0;
pthread_mutex_t mutex;
void* processA()
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        x = x+1;
        if (x==20) x=0;
        printf("\nProcess a: %d",x);
        pthread_mutex_unlock(&mutex);
    }
}

void* processB()
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        x = x+1;
        if (x==20) x=0;
        printf("\nProcess b: %d",x);
        pthread_mutex_unlock(&mutex);
    }
}

int main()
{
    pthread_t pA,pB;
    pthread_mutex_init(&mutex,NULL);
    pthread_create(&pA,NULL,&processA,NULL);
    pthread_create(&pB,NULL,&processB,NULL);
    while(1){}
    return 0;
}

```

Dùng mutex để đảm bảo tại 1 thời điểm chỉ có 1 process được vào critical section.

Kết quả:

```
Process a: 5  
Process a: 4  
Process a: 5  
Process b: 6  
Process b: 7  
Process b: 8
```

Kết quả chạy đúng như mong đợi.