Name: Lê Minh Nguyệt
ID: 21521211
Class: IT007.N11

OPERATING SYSTEM
LAB 05'S REPORT

SUMMARY

Self-scrores: 9,5

Section 5.4

Ex 1. Hiện thực hóa mô hình trong ví dụ 5.3.1.2, thay bằng điều kiện sells <= products <= sells + 21 (2 số cuối MSSV + 10 = 11 + 10 = 21)

- Process A mô tả số lượng hàng bán được, lưu trong biến toàn cục sells.
- Process B mô tả số lượng sản phẩm được sản xuất, lưu trong biến toàn cục products.
- Hai biến sells và products là dữ liệu được chia sẻ giữa hai process A và B.
- Ban đầu chưa có hàng, và cũng chưa bán được sản phẩm nào, ta đặt sells = products = 0.
- Với điều kiện sells <= products <= sells + 21, ta sử dụng hai semaphore sem1, sem2 để đồng bộ hai process A và B. Trong đó, sem1 đảm bảo điều kiện sells <= products, và sem2 đảm bảo điều kiện products <= sells + 21.
- Trong bài tập này, ta có thể xem giá trị của sem1 là số lượng sản phẩm có sẵn (được sản xuất ra) để bán, sem2 là số lượng sản phẩm có thể sản xuất thêm, bằng lượng đã bán + 21 (không vượt quá 21 so với số lượng đã bán).

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

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

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

void *processB(void* mess) {
        while(1) {
                sem_wait(&sem2);
                products++;
                printf("Process B | products = %d\n", products);
                sem_post(&sem1);
        }
}
```

Hình 1. Sử dụng semaphore trong hai process để đồng bộ chúng

- Đặt hàm *sem_wait(&sem1)* trước khi thực hiện tăng biến sells, để mỗi khi process A muốn tăng biến sells, nó sẽ kiểm tra xem giá trị của sem1 liệu có lớn hơn 0 (có sản phẩm để bán).
  + Nếu giá trị của sem1 = 0 (tức không có sản phẩm được sản xuất), process A bị block (không thực hiện bán hàng).

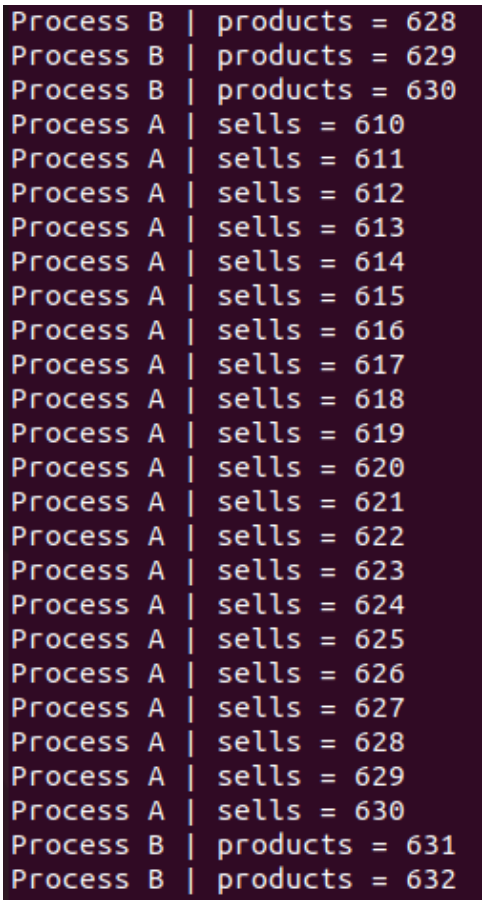
+ Nếu giá trị của sem1 > 0 (tức có sản phẩm để bán), process A sẽ được phép thực hiện tăng sells và giá trị của sem1 sẽ giảm đi 1 (số lượng sản phẩm sẵn có giảm đi sau khi bán).

- Sau khi thực hiện bán hàng, process A gặp hàm *sem_post(&sem2)* nên giá trị của sem2 sẽ được tăng lên 1 (đã bán thêm 1 nên có thể sản xuất thêm 1).
- Sau 1 khoảng thời gian, process A bị dừng và process B sẽ được chạy. Process B gặp hàm *sem_wait(&sem2)* sẽ kiểm tra liệu giá trị của sem2 liệu có lớn hơn 0 (có thể sản xuất thêm).
  + Nếu giá trị của sem2 = 0 (tức không được sản xuất thêm hay products hiện đang bằng sells + 21), process B bị block, không thực hiện tăng products nữa.
  + Nếu giá trị của sem2 > 0, process B được phép tăng products (sản xuất thêm 1 sản phẩm), sau đó qua hàm *sem_post(&sem1)* sẽ tăng giá trị của sem1 lên 1 (số sản phẩm có thể được bán tăng lên 1). Khi đó, nếu process A đang bị block do hàm *sem_wait(&sem1)* sẽ được mở ra và sẵn sàng để chạy, chỉ chờ đến khi process B hết thời gian của nó và nhường lại.
- Qua mỗi lần lặp, process B đều gặp hàm *sem_wait(&sem2)*, kiểm tra giá trị của sem2 và trừ giá trị đó đi 1 nếu nó lớn hơn 0, nếu nó bằng 0 trước khi process B hết thời gian, nó sẽ nhường lại cho process A và phải chờ process A chạy rồi tăng sem2 lên.

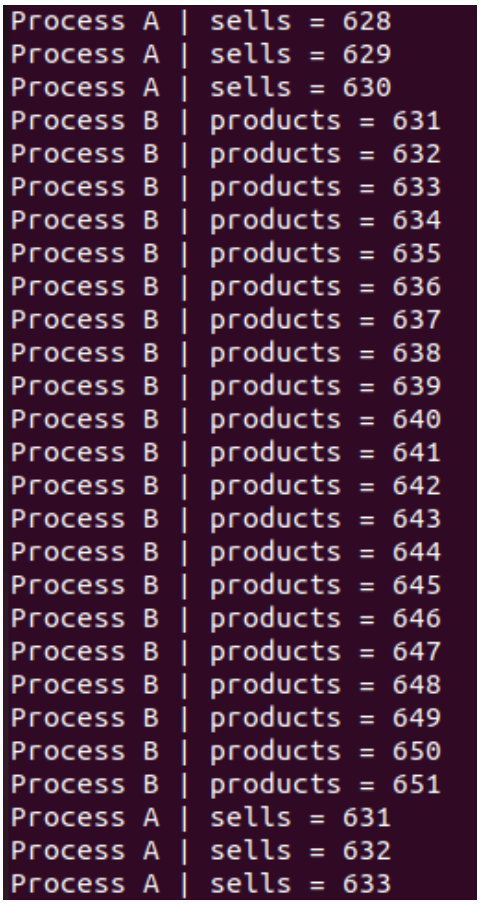
```
int main() {
        sem_init(&sem1, 0, 0);
        sem_init(&sem2, 0, 21);
        pthread_t pA, pB;
        pthread_create(&pA, NULL, &processA, NULL);
        pthread_create(&pB, NULL, &processB, NULL);
        while(1){}
        return 0;
}
```

Hình 2. Hàm main khởi tạo các semaphore và thread

- Giá trị ban đầu được khởi tạo của sem1 = 0 (chưa thể bán do không có sản phẩm).
- Giá trị ban đầu được khởi tạo của sem2 = 21 (sells ban đầu = 0 + 21, có thể sản xuất thêm 21 sản phẩm).
- Khi process B bị block (không thể sản xuất thêm), sẽ nhường cho process A chạy và tăng sells cùng giá trị sem2 lên (bán bớt sản phẩm đi để có thể sản xuất tiếp). Nếu process A không thể bán nữa (hết sản phẩm), sẽ nhường cho process B tăng products và sem1 lên (sản xuất thêm).



Hình 3. Kết quả chạy chương trình, thỏa mãn điều kiện sells <= products



Hình 4. Kết quả chạy chương trình, thỏa mãn điều kiện products <= sells + 21

- Ta thấy trong hình 3, process A chỉ tăng sells lên bằng 630 rồi dừng và nhường cho process B, do trước đó products đã được tính trong process B bằng 630. Thỏa mãn điều kiện: sells <= products
- Trong hình 4, ta thấy sau khi được nhường và chạy, process B tiếp tục tăng products lên, nhưng chỉ tăng cho đến khi bằng 651 (sells trước đó được tính bằng 630, + 21 = 651) rồi lại nhường cho process A. Thỏa mãn điều kiện: products <= sells + 21

Ex 2. Đồng bộ hóa 2 thread thêm và xóa phần tử của cùng 1 mảng a

2.1. Chạy thử và tìm lỗi khi chưa được đồng bộ

- Biến toàn cục a là một mảng số nguyên, tối đa n phần tử. Ta đặt n là một số cố định (ở đây n được đặt bằng 100000).
- Biến toàn cục size chứa số phần tử của mảng a. Ban đầu mảng a chưa có phần tử nào, size = 0.
- Mảng a và biến size là vùng dữ liệu được chia sẻ giữa hai process A và B.

- Process A thực hiện thêm một phần tử là một số ngẫu nhiên vào mảng a. Nếu mảng đầy (số phần tử bằng số phần tử tối đa) thì in ra “Array a is full.” Nếu không, in ra số phần tử hiện tại của a.
- Process B thực hiện xóa phần tử cuối cùng của a ra khỏi mảng và tính lại số phần tử. Nếu sau khi xóa, mảng rỗng thì in ra “Nothing in array a.” Nếu không, in ra số phần tử hiện tại của a.

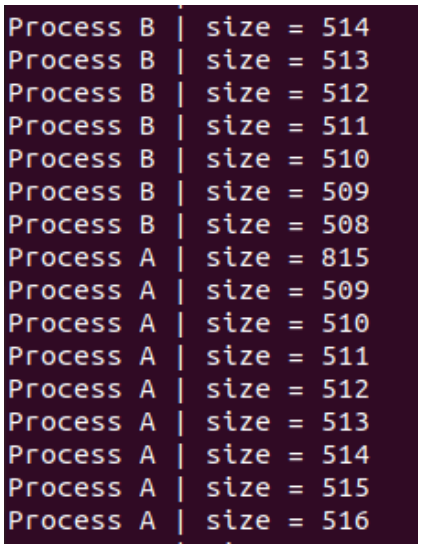
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define n 100000
#define NULLNUMBER -100000

int a[n];
int size=0;

void *processA(void* mess) {
        while(1) {
                a[size++]=rand();
                if(size==n) printf("Process A | Array a is full.\n");
                else printf("Process A | size = %d\n", size);
        }
}

void *processB(void* mess) {
        while(1) {
                if(size>0) {
                        a[size-1]=NULLNUMBER;
                        size--;
                        if(size==0)
                                printf("Process B | Nothing in array a.\n");
                        else printf("Process B | size = %d\n", size);
                }
        }
```

Hình 5. Process A thêm phần tử vào mảng a, process B xóa phần tử khỏi mảng a và in ra màn hình

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

Hình 6. Tạo hai thread chạy đồng thời

- Sau khi viết chương trình, ta tiến hành chạy thử và tìm lỗi.



Hình 7. Phát hiện lỗi khi thực hiện chương trình chưa đồng bộ

- Quan sát kết quả số lượng phần tử được in ra bởi process A và process B. Ta thấy có trường hợp như hình 7. Sau khi process B thực hiện xóa, mảng a chỉ còn 508 phần tử. Lúc này, process B hết time slice nên nhường cho process A chạy. Thay vì thêm 1 phần tử và số lượng tăng lên 509, ta thấy size lúc này lại bằng 815, sau đó mới bằng 509.

2.2. Thực hiện đồng bộ hóa với semaphore

- Ta thực hiện đồng bộ hóa 2 process này bằng 2 semaphore sem1, sem2 và 1 mutex. Trong đó, sem1 đảm bảo điều kiện size <= n, sem2 đảm bảo điều kiện size >= 0 và mutex đảm bảo tại 1 thời điểm, chỉ một trong hai process truy cập vào và làm thay đổi mảng a cũng như giá trị của biến size.
- Trong bài tập này, ta có thể xem sem1 là số lượng phần tử có thể thêm vào a mà size vẫn không vượt quá n, sem2 là số lượng phần tử đang có trong a.
- Ban đầu, mảng rỗng, nên size = 0, sem2 khởi tạo bằng 0, sem1 khởi tạo bằng n (a có thể chứa thêm n phần tử).

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

#define n 100000
#define NULLNUMBER -100000

int a[n];
int size=0;
sem_t sem1, sem2;
pthread_mutex_t mutex;

void *processA(void* mess) {
        while(1) {
                sem_wait(&sem1);
                pthread_mutex_lock(&mutex);
                a[size++]=rand();
                if(size==n) printf("Process A | Array A is full.\n");
                else printf("Process A | size = %d\n", size);
                sem_post(&sem2);
                pthread_mutex_unlock(&mutex);
        }
}
```

Hình 8. Thực hiện đồng bộ hóa process A bằng semaphore và mutex

- Đặt hàm *sem_wait(&sem1)* trước khi process A thực hiện thêm phần tử để kiểm tra liệu sem1 có lớn hơn 0 (có thể thêm phần tử vào a).
  + Nếu sem1 = 0 (tức không thể thêm phần tử vào a hay a đã đầy), process A sẽ bị block và chờ đến khi sem1 được tăng lên lớn hơn 0 (tức a còn trống chỗ để thêm phần tử mới).
  + Nếu sem1 > 0, process A xin phép khóa mutex bằng hàm *pthread_mutex_lock(&mutex)* để truy cập vào CS.
    - Nếu process B đang khóa mutex, process A phải chờ đến khi không còn process nào khóa mutex mới được phép khóa và truy cập vào CS để thực hiện các lệnh bên dưới.
    - Nếu không có process nào đang khóa mutex, process A sẽ được khóa mutex và chạy, thêm 1 số ngẫu nhiên vào a tại vị trí size và tăng size lên 1. Trong khi đó, process khác không được truy cập vào CS, nếu yêu cầu khóa mutex cũng phải chờ process A unlock mutex. Sau khi thực hiện thêm phần tử, process A qua hàm *sem_post(&sem2)* sẽ tăng giá trị sem2 lên 1 (số lượng phần tử đang có trong mảng tăng thêm 1), và process A sẽ mở khóa mutex bằng hàm *pthread_mutex_unlock(&mutex)* để process tiếp theo yêu cầu khóa được thực hiện.

```
void *processB(void* mess) {
        while(1) {
                sem_wait(&sem2);
                pthread_mutex_lock(&mutex);
                a[size-1]=NULLNUMBER;
                size--;
                if(size==0)
                        printf("Process B | Nothing in array a.\n");
                else printf("Process B | size = %d\n", size);
                sem_post(&sem1);
                pthread_mutex_unlock(&mutex);
        }
}

int main() {
        sem_init(&sem1, 0, n);
        sem_init(&sem2, 0, 0);
        pthread_t pA, pB;
        pthread_create(&pA, NULL, &processA, NULL);
        pthread_create(&pB, NULL, &processB, NULL);
        while(1) {}
        return 0;
}
```

Hình 9. Thực hiện đồng bộ hóa process B bằng semaphore và mutex

- Đặt hàm *sem_wait(&sem2)* trước khi process B thực hiện xóa phần tử để kiểm tra liệu sem2 có lớn hơn 0 (trong mảng còn phần tử để xóa hay không).
  + Nếu sem2 = 0 (tức không còn phần tử nào trong a hay a rỗng), process B sẽ bị block và chờ đến khi sem2 được tăng lên lớn hơn 0 (tức a có phần tử để xóa).
  + Nếu sem2 > 0, process B xin phép khóa mutex bằng hàm *pthread_mutex_lock(&mutex)* để truy cập vào CS.
    - Nếu process A đang khóa mutex, process B phải chờ đến khi không còn process nào khóa mutex mới được phép khóa và truy cập vào CS để thực hiện các lệnh bên dưới.
    - Nếu không có process nào đang khóa mutex, process B sẽ được khóa mutex và chạy, xóa phần tử cuối của a giảm size đi 1. Trong khi đó, process khác không được truy cập vào CS, nếu yêu cầu khóa mutex cũng phải chờ process B mở khóa mutex. Sau khi thực hiện xóa phần tử, process B qua hàm *sem_post(&sem1)* sẽ tăng giá trị sem1 lên 1 (số lượng chỗ trống trong mảng tăng thêm 1), và process B sẽ mở khóa mutex bằng hàm *pthread_mutex_unlock(&mutex)* để process tiếp theo yêu cầu khóa được thực hiện.
- Thực hiện đồng bộ bằng semaphore giải quyết được trường hợp khi process A làm mảng đầy (size == n) hay process B làm mảng rỗng (size == 0) nhưng chưa hết time slice, nên sẽ liên tục in ra màn hình “Array is full.” hay “Nothing in array a.” cho đến khi hết time slice và phải nhường cho process còn lại. Hai semaphore sẽ đảm bảo hai process chỉ được thực hiện tiếp nếu vẫn thỏa mãn điều kiện size <= n (process A) và size >= 0 (process B).

```
Process B | size = 4
Process B | size = 3
Process B | size = 2
Process B | size = 1
Process B | Nothing in array a.
Process A | size = 1
Process A | size = 2
Process A | size = 3
Process A | size = 4
```

Hình 10. Kết quả chạy chương trình sau khi đồng bộ (trường hợp xóa phần tử đến khi mảng rỗng)

```
Process A | size = 293
Process A | size = 294
Process A | size = 295
Process A | size = 296
Process B | size = 295
Process B | size = 294
Process B | size = 293
Process B | size = 292
Process B | size = 291
```

Hình 11. Kết quả chạy chương trình sau khi đồng bộ (trường hợp hết time slice)

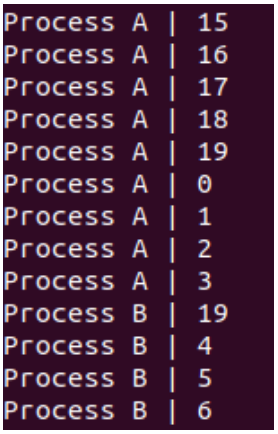
- Quan sát kết quả khi chạy chương trình đã được đồng bộ, ta thấy trong hình 10, process B xóa phần tử trong a đến khi mảng rỗng, đã in ra màn hình và nhường lại cho process A dù chưa hết time slice (giá trị của sem2 bằng 0).
- Trong hình 11, process A tăng số phần tử lên 296 và hết time slice nên nhường cho process B tiếp tục giảm số phần tử xuống 295.

Ex 3. Hiện thực mô hình 2 process A, B chạy đồng thời cùng thực hiện một công việc và nhận xét kết quả

- Ta viết chương trình như hình 12, hiện thực 2 process A và B cùng tăng giá trị của biến toàn cục x lên 1. Nếu x == 20 thì đặt lại x = 1. Sau đó, in giá trị của x ra màn hình (x là vùng dữ liệu được chia sẻ giữa 2 process A và B).

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

int x=0;

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

void *processB(void* mess){
        while(1) {
                x=x+1;
                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;
}
```

Hình 12. Hiện thực mô hình 2 process A, B chưa đồng bộ



Hình 13. Phát hiện lỗi trong kết quả chạy chương trình hiện thực mô hình 2 process A, B chưa đồng bộ

- Sau khi hiện thực mô hình bằng code, ta chạy thử chương trình, quan sát kết quả và nhận thấy có lỗi. Process A liên tiếp thực hiện tăng x, cho đến khi x == 20 thì đặt lại x = 0 và in ra màn hình, sau đó tiếp tục tăng x. Sau khi in ra x = 3, process A hết time slice nên nhường lại cho process B. Thay vì tiếp tục tăng giá trị của x lên 4, ta thấy process B lại in ra màn hình 19 rồi mới đến 4.
- Giải thích: Trước đó, khi process B đang chạy, giá trị hiện tại của x đã được nạp vào thanh ghi mà chưa kịp in ra màn hình thì process B hết time slice, nên phải nhường cho process A chạy. Lúc này, process A nạp lại x vào thanh ghi khác và thực hiện tính toán (in ra màn hình). Khi process B chạy lại sau đó, giá trị của x đã cũ nhưng nó vẫn thực hiện tiếp công việc in ra chưa hoàn thành của mình, rồi mới nạp giá trị mới của x vào và tính toán tiếp.

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

- Để sửa lỗi bất hợp lý trong kết quả của Ex 3, ta sử dụng mutex đồng bộ giữa hai process A và B.
- Ta đặt hàm *pthread_mutex_lock(&mutex)* trước khi mỗi process truy cập vào CS (thực hiện tính toán thay đổi giá trị của x) và hàm *pthread_mutex_unlock(&mutex)* sau khi mỗi process thực hiện xong CS.
- Process A trước khi thay đổi giá trị của x, sẽ phải yêu cầu khóa CS bằng hàm lock. Khi đó:
  + Nếu process B đang khóa mutex, process A sẽ phải chờ đến khi process B mở khóa.
  + Nếu process B không khóa mutex (không process nào đang khóa mutex), process A sẽ được phép khóa và thực hiện thay đổi giá trị của x rồi in ra màn hình. Trong quá trình này, process A sẽ không bị ngắt và cũng không có process nào khác được truy cập vào CS làm thay đổi giá trị của x.
- Tương tự với process B, sẽ phải chờ nếu process A đang khóa mutex, và sẽ được thực hiện tính toán trên x mà không bị ngắt nếu không có process khác đang khóa mutex.

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

int x=0;
pthread_mutex_t mutex;

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

Hình 14. Đồng bộ 2 process A, B bằng mutex

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

Hình 15. Khởi tạo thread khi đồng bộ bằng mutex

```
Process A | 10
Process A | 11
Process A | 12
Process A | 13
Process A | 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
```

Hình 16. Kết quả sau khi đã đồng bộ bằng mutex

- Quan sát kết quả sau khi chạy chương trình đã đồng bộ, ta thấy khi process A tăng x lên bằng 14, nó bị hết time slice nên nhường cho process B chạy. Lúc này, process B tiếp tục tăng x từ 14 lên đúng bằng 15, và tiếp tục tăng đến khi bằng 20 thì đặt lại bằng 0.

Section 5.5

Ex 1. Lập trình mô phỏng và đồng bộ các thread chạy đồng thời chứa các lệnh (a) -> (g) theo các thứ tự:

- (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

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)

- Ta có thứ tự thực hiện các process như sau:

Process	Biểu thức	Thứ tự
A	*w = x1 * x2*	Sau G (nếu G đã được tính ít nhất 1 lần trước đó)
B	*v = x3 * x4*	
C	*y = v * x5*	Sau B, trước E
D	*z = v * x6*	Sau B, trước F
E	*y = w * y*	Sau C, trước G
F	*z = w * z*	Sau D, trước G
G	*ans = y + z*	Sau E và F

- Để đảm bảo thứ tự này, ta sử dụng các semaphore: semA, semB, semBC, semBD, semCE, semDF, semEG, semFG.
- Trong đó, semA và semB có giá trị khởi tạo bằng 1, các semaphore còn lại khởi tạo bằng 0 (vì semA và semB đảm bảo process A và process B sẽ được thực hiện 1 lần trước tiên, và chỉ được thực hiện tiếp sau khi process G tính xong 1 giá trị của ans).
- Công việc của các semaphore còn lại (trừ semA, semB):
  + semAE sẽ được tăng giá trị thêm 1 sau khi process A chạy xong, process E có thể chạy (sau khi đợi process A thông qua semAE).
  + semBC sẽ được tăng giá trị thêm 1 sau khi process B chạy xong, process C có thể chạy (sau khi đợi process B thông qua semBC).
  + Ttương tự với các semaphore còn lại.
- Bên cạnh đó, ta sử dụng các mutex mutex1, mutex2, mutex3, mutex4 để đảm bảo các process làm thay đổi giá trị của cùng 1 biến sẽ không truy cập vào biến đó đồng thời. Cụ thể:

+ mutex1 và mutex4 tránh cho process A truy cập vào w cùng lúc với process E và F, nhưng E và F vẫn có thể truy cập vào w cùng lúc với nhau (vì hai process này không làm thay đổi w).
+ mutex2 cùng mutex3 tránh cho process B truy cập vào v cùng lúc với process C và D, nhưng C và D vẫn có thể truy cập vào v cùng lúc với nhau (vì hai process này không làm thay đổi v).
+ mutex2 còn tránh process C truy cập vào y cùng lúc với process E (vì 2 process này cùng làm thay đổi y).
+ mutex3 còn tránh process D truy cập vào z cùng lúc với process F (vì 2 process này cùng làm thay đổi z).
+ mutex2 cùng mutex3 tránh process C, D, E, F (làm thay đổi y, z) truy cập vào y, z cùng lúc với process G (dùng y, z để tính toán).

- Ta đặt cho x1 -> x6 nhận giá trị là các số nguyên ngẫu nhiên trong khoảng [1;10] khác nhau tại mỗi lần tính toán (để dễ quan sát kết quả).

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

int w,v,y,z,ans;
int x1,x2,x3,x4,x5,x6;
sem_t semAE, semAF, semBC, semBD, semCE, semDF, semEG, semFG;
pthread_mutex_t mutex1, mutex2, mutex3, mutex4;
sem_t semA, semB;

void *processA(void* mess) {
        while(1) {
                sem_wait(&semA);
                pthread_mutex_lock(&mutex1);
                pthread_mutex_lock(&mutex4);
                x1=rand() % 10 + 1;
                x2=rand() % 10 + 1;
                w=x1*x2;
                printf("Process A | w = %d * %d = %d\n", x1, x2, w);
                sem_post(&semAE);
                sem_post(&semAF);
                pthread_mutex_unlock(&mutex1);
                pthread_mutex_unlock(&mutex4);
        }
}
```

Hình 17. Đồng bộ process A bằng semA, semAE, semAF và mutex1, mutex4

```
void *processB(void* mess) {
        while(1) {
                sem_wait(&semB);
                pthread_mutex_lock(&mutex2);
                pthread_mutex_lock(&mutex3);
                x3=rand() % 10 + 1;
                x4=rand() % 10 + 1;
                v=x3*x4;
                printf("Process B | v = %d * %d = %d\n", x3, x4, v);
                sem_post(&semBC);
                sem_post(&semBD);
                pthread_mutex_unlock(&mutex2);
                pthread_mutex_unlock(&mutex3);
        }
}
void *processC(void* mess) {
        while(1) {
                sem_wait(&semBC);
                pthread_mutex_lock(&mutex2);
                x5=rand() % 10 + 1;
                y=v*x5;
                printf("Process C | y = v * %d = %d\n", x5, y);
                sem_post(&semCE);
                pthread_mutex_unlock(&mutex2);
        }
}
```

Hình 18. Đồng bộ process B bằng semB, semBC, semBD và mutex2, mutex3. Đồng bộ process C bằng semBC, semCE và mutex2

```
void *processD(void* mess) {
        while(1) {
                sem_wait(&semBD);
                pthread_mutex_lock(&mutex3);
                x6=rand() % 10 + 1;
                z=v*x6;
                printf("Process D | z = v * %d = %d\n", x6, z);
                sem_post(&semDF);
                pthread_mutex_unlock(&mutex3);
        }
}
void *processE(void* mess) {
        while(1) {
                sem_wait(&semAE);
                sem_wait(&semCE);
                pthread_mutex_lock(&mutex1);
                pthread_mutex_lock(&mutex2);
                y=w*y;
                printf("Process E | y = w * y = %d\n", y);
                sem_post(&semEG);
                pthread_mutex_unlock(&mutex1);
                pthread_mutex_unlock(&mutex2);
        }
}
```

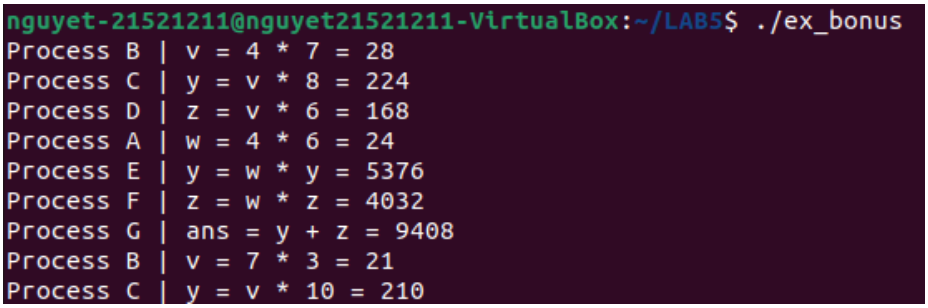
Hình 19. Đồng bộ process D bằng semBD, semDF và mutex3. Đồng bộ process E bằng semAE, semCE, semEG và mutex1, mutex2

```c
void *processF(void* mess) {
        while(1) {
                sem_wait(&semAF);
                sem_wait(&semDF);
                pthread_mutex_lock(&mutex3);
                pthread_mutex_lock(&mutex4);
                z=w*z;
                printf("Process F | z = w * z = %d\n", z);
                sem_post(&semFG);
                pthread_mutex_unlock(&mutex3);
                pthread_mutex_unlock(&mutex4);
        }
}
void *processG(void* mess) {
        while(1) {
                sem_wait(&semEG);
                sem_wait(&semFG);
                pthread_mutex_lock(&mutex2);
                pthread_mutex_lock(&mutex3);
                ans=y+z;
                printf("Process G | ans = y + z = %d\n", ans);
                sem_post(&semA);
                sem_post(&semB);
                pthread_mutex_unlock(&mutex2);
                pthread_mutex_unlock(&mutex3);
        }
}
```

Hình 20. Đồng bộ process F bằng semAF, semDF, semFG và mutex3, mutex4. Đồng bộ process G bằng semEG, semFG, semA, semB và mutex2, mutex3

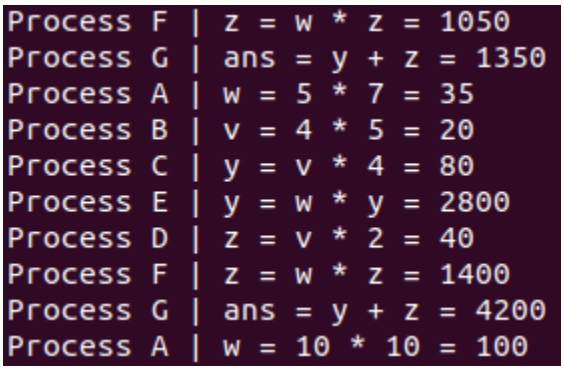
```c
int main() {
        sem_init(&semA, 0, 1);
        sem_init(&semB, 0, 1);
        sem_init(&semAE, 0, 0);
        sem_init(&semAF, 0, 0);
        sem_init(&semBC, 0, 0);
        sem_init(&semBD, 0, 0);
        sem_init(&semCE, 0, 0);
        sem_init(&semDF, 0, 0);
        sem_init(&semEG, 0, 0);
        sem_init(&semFG, 0, 0);
        pthread_t pA, pB, pC, pD, pE, pF, pG;
        pthread_create(&pA, NULL, &processA, NULL);
        pthread_create(&pB, NULL, &processB, NULL);
        pthread_create(&pC, NULL, &processC, NULL);
        pthread_create(&pD, NULL, &processD, NULL);
        pthread_create(&pE, NULL, &processE, NULL);
        pthread_create(&pF, NULL, &processF, NULL);
        pthread_create(&pG, NULL, &processG, NULL);
        while(1) {}
        return 0;
}
```

Hình 21. Khởi tạo giá trị ban đầu cho các semaphore và tạo các thread chạy đồng thời



Hình 22. Kết quả chạy chương trình sau khi đồng bộ (1)

- Quan sát kết quả (1), ta thấy thứ tự thực hiện các process là: B -> C -> D -> A -> E -> F -> G. Đảm bảo được yêu cầu:
  + Process B tính được v từ x3 = 4 và x4 = 7
  + Process C dùng v vừa tính trong process B và x5 = 8 tính được y
  + Process D dùng v vừa tính trong process B và x6 = 6 tính được z
  + Process A tính được w từ x1 = 4 và x2 = 6
  + Process E dùng w vừa tính trong process A và y vừa tính trong process C tính được y
  + Process F dùng w vừa tính trong process A và z vừa tính trong process D tính được z
  + Process G dùng y vừa tính trong process E và z vừa tính trong process F tính được ans = 9408



Hình 23. Kết quả chạy chương trình sau khi đồng bộ (2)

- Quan sát kết quả (2), ta thấy thứ tự thực hiện các process là: A -> B -> C -> E -> D -> F -> G. Đảm bảo được yêu cầu:
  + Process A tính được w từ x1 = 5 và x2 = 7
  + Process B tính được v từ x3 = 4 và x4 = 5
  + Process C dùng v vừa tính trong process B và x5 = 4 tính được y
  + Process E dùng w vừa tính trong process A và y vừa tính trong process C tính được y
  + Process D dùng v vừa tính trong process B và x6 = 2 tính được z
  + Process F dùng w vừa tính trong process A và z vừa tính trong process D tính được z

+ Process G dùng y vừa tính trong process E và z vừa tính trong process F tính được ans = 4200

Nhận xét: *Cả hai kết quả dù thứ tự thực hiện khác nhau nhưng vẫn đảm bảo được thứ tự đúng như yêu cầu đề bài.*