Name: Huỳnh Viết Tuấn Kiệt

ID: 20521494

Class: IT007.M13.2

OPERATING SYSTEM
LAB 5'S REPORT

SUMMARY

Task		**Status**	**Page**
Question	Ex 5.4.1	Hoàn thành	2
	Ex 5.4.2	Hoàn thành	5
	Ex 5.4.3	Hoàn thành	9
	Ex 5.4.4	Hoàn thành	12
	Ex 5.5 (bonus)	Hoàn thành	14

Self-scrores: 10

1. Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: sells <= products <= sells + [2 số cuối của MSSV + 10]

a. Hiện thực

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

int sells = 5, products = 10;
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);
        }
}
```

Hình 1.1. Khai báo và hiện thực chương trình

```c
int main()
{
        sem_init(&sem1, 0, 5); // sem1: check out available PRODUCTS
        sem_init(&sem2, 0, 5+94+10-10); // sem2: check out PRODUCTS limit
        pthread_t pA, pB;
        pthread_create(&pA, NULL, &processA, NULL);
        pthread_create(&pB, NULL, &processB, NULL);

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

Hình 1.2. Hàm main

b. Ý tưởng

BÀI TOÁN YÊU CẦU CÓ 2 ĐIỀU KIỆN CẦN ĐỒNG BỘ:

- **sells <= products**
- **products <= sells + [MSSV + 10]**

DO ĐÓ SỬ DỤNG 2 SEMAPHORE ĐỂ ĐỒNG BỘ

- **sem1** kiểm tra điều kiện giới hạn dưới, lệnh `sem_wait(&sem1)` để đảm bảo khi **products > sells** thì sells mới có thể tăng lên, lệnh `sem_post(&sem1)` báo cho process A biết **products** đã được tăng lên
- **sem2** kiểm tra điều kiện giới hạn dưới, lệnh `sem_wait(&sem2)` để đảm bảo khi **products < sells + [MSSV + 10]** thì **products** mới có thể tăng lên, lệnh `sem_post(&sem2)` báo cho process B biết **sells** đã được tăng lên

KHỞI TẠO BAN ĐẦU:

- **sem1** khởi tạo = **products – sells**, thể hiện số lần sells được phép tăng thêm để đạt được giới hạn dưới
- **sem2** khởi tạo = **sells + MSSV + 10 – products**, thể hiện số lần products được phép tăng thêm để đạt được giới hạn trên

c. Kết quả

```
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ gcc bt1.c -o bt1 -lpthread -lrt
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ ./bt1
SELLs = 6
SELLs = 7
SELLs = 8
SELLs = 9
SELLs = 10
PRODUCTs = 11
PRODUCTs = 12
PRODUCTs = 13
PRODUCTs = 14
PRODUCTs = 15
PRODUCTs = 16
PRODUCTs = 17
```

```
SELLs = 110
SELLs = 111
SELLs = 112
SELLs = 113
SELLs = 114
SELLs = 115
SELLs = 116
SELLs = 117
SELLs = 118
SELLs = 119
SELLs = 120
SELLs = 121
SELLs = 122
SELLs = 123
PRODUCTs = 125
PRODUCTs = 126
PRODUCTs = 127
PRODUCTs = 128
PRODUCTs = 129
PRODUCTs = 130
PRODUCTs = 131
PRODUCTs = 132
PRODUCTs = 133
PRODUCTs = 134
PRODUCTs = 135
PRODUCTs = 136
PRODUCTs = 137
PRODUCTs = 138
```

Hình 1.3. Kết quả thực thi

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".**

a. Hiện thực

```
int n, i = 0, count = 0;
int a[10000];
sem_t sem1, sem2;
pthread_mutex_t mutex;
```

Hình 2.1. Khai báo & khởi tạo

```
void *processA(void* mess)
{
        while(1)
        {
                sem_wait(&sem1);
                pthread_mutex_lock(&mutex);
                a[i++] = rand() % (5*n);
                count++;
                printf("\n[PUSH] The number of elements in a: %d, they are: ",
count);
                for (int t = 0; t < count; t++)
                        printf("%d ", a[t]);
                printf("\n-------------------------------------------------");
                pthread_mutex_unlock(&mutex);
                sem_post(&sem2);
        }
}
```

Hình 2.2. Process A

```c
void *processB(void* mess)
{
        while(1)
        {
                sem_wait(&sem2);
                pthread_mutex_lock(&mutex);
                int k, j;
                if (count == 0) printf("No thing in array a!\n");
                else
                {
                        printf("\nRemove element has index = ");
                        count--;
                        if (count == 0) printf("0");
                        else
                        {
                                k = rand() % count;
                                printf("%d", k);
                        }
                        i--;
                        for (j = k; j < count; j++) a[j] = a[j + 1];
                        printf("\n[POP] The number of elements in a: %d, they a
re: ", count);
                        for (int t = 0; t < count; t++) printf("%d ", a[t]);
                }
                pthread_mutex_unlock(&mutex);
                sem_post(&sem1);
        }
}
```

Hình 2.3. Process B

```c
int main()
{
        printf("Enter n: ");
        scanf("%d", &n);
        pthread_mutex_init(&mutex, NULL);
        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 2.4. Hàm main

b. Ý tưởng

KHỞI TẠO:

- Mảng `a[]` chứa các phần tử số nguyên `(int)`
- Biến `i` ban đầu bằng 0 để biểu thị vị trí hiện tại trong mảng
- Biến `count` ban đầu bằng 0 chứa số lượng phần tử đang có trong mảng

BÀI TOÁN YÊU CẦU CÓ 2 ĐIỀU KIỆN CẦN ĐỒNG BỘ

- Không thể xóa phần tử trong mảng có kích thước = 0 (`count == 0`)
- Không thể thêm phần tử khi trong mảng đang có `n` phần tử

DO ĐÓ SỬ DỤNG 2 SEMAPHORE ĐỂ ĐỒNG BỘ:

- **sem1** kiểm tra điều kiện trong mảng có ít hơn `n` phần tử để thêm hay không, `sem_wait(&sem1)` ở process A để block tiến trình khi có `n` phần tử trong mảng và không được phép thêm phần tử mới vào mảng `a`. `sem_post(&sem1)` ở cuối process B để báo hiệu đã xóa phần tử trong mảng `a` và process A có quyền được thêm phần tử mới
- **sem2** kiểm tra điều kiện trong mảng có phần tử để xóa hay không, `sem_wait(&sem2)` bị block và xuất ra `"Nothing in array a!"` nếu không có phần tử trong mảng, ngược lại nếu có phần tử sẽ random 1 vị trí trong mảng và xóa phần tử ở vị trí đó, `sem_post(&sem2)` ở cuối process A để báo hiệu đã thêm phần tử mới vào mảng `a` và process B có quyền được xóa phần tử.

KHỞI TẠO SEMAPHORE:

- **sem1** = `n`, biểu thị cho mảng ban đầu có `n` vị trí trống và process A được phép thêm liên tục n phần tử mới vào mảng
- **sem2** = `0`, biểu thị mảng ban đầu không chứa bất kì phần tử nào (tương tự như mảng ban đầu có n vị trí trống ở câu trên) và process B không được phép thực hiện xóa phần tử

SỬ DỤNG MUTEX:

- Do trong bài toán có sử dụng biến `count` chung cho cả 2 process. Để tránh 2 process không tranh chấp lẫn nhau và làm thay đổi biến `count`, có thể đưa đoạn mã xử lý vào trong cặp `mutex_lock` và `mutex_unlock`

c. Kết quả

```
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ ./bt2
Enter n: 5

[PUSH] The number of elements in a: 1, they are: 14
-------------------------------------------------
[PUSH] The number of elements in a: 2, they are: 14 14
-------------------------------------------------
[PUSH] The number of elements in a: 3, they are: 14 14 19
-------------------------------------------------
[PUSH] The number of elements in a: 4, they are: 14 14 19 7
-------------------------------------------------
[PUSH] The number of elements in a: 5, they are: 14 14 19 7 15
-------------------------------------------------
Remove element has index = 2
[POP] The number of elements in a: 4, they are: 14 14 7 15
-------------------------------------------------
Remove element has index = 2
[POP] The number of elements in a: 3, they are: 14 14 15
-------------------------------------------------
Remove element has index = 0
[POP] The number of elements in a: 2, they are: 14 15
-------------------------------------------------
Remove element has index = 0
[POP] The number of elements in a: 1, they are: 15
-------------------------------------------------
Remove element has index = 0
[POP] The number of elements in a: 0, they are:
```

Hình 2.5. Kết quả thực thi

3. Cho 2 Process A và B chạy song song. Hiện thực mô hình trên C và nhận xét kết quả

a. Hiện thực

```c
#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("PA = %d\n", x);
        }
}
void* processB()
{
        while(1)
        {
                x = x + 1;
                if (x == 20)
                        x = 0;
                printf("PB = %d\n", x);
        }
}
```

Hình 3.1. Khai báo và hiện thực chương trình

b. Kết quả

```
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ gcc bt3.c -o bt3 -lpthread -lrt
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ ./bt3
PB = 1
PB = 2
PB = 3
PB = 4
PB = 5
PB = 6
PB = 7
PB = 8
PB = 9
PB = 10
PB = 11
PB = 12
PB = 13
PB = 14
PB = 15
PB = 16
PB = 17
PB = 18
PB = 19
PB = 0
PB = 1
PB = 2
PB = 3
PB = 4
PB = 5
```

Hình 3.2. Kết quả thực thi

```
PB = 17
PB = 18
PB = 19
PB = 0
PB = 1
PB = 2
PB = 3
PA = 7
PA = 4
PA = 5
PA = 6
PA = 7
PA = 8
PA = 9
PA = 10
```

Hình 3.3. Kết quả bất hợp lí của chương trình

c. Nhận xét

Với đoạn mã chương trình trên (*Hình 3.1*) và in ra kết quả bất hợp lí (*Hình 3.3*), có thể thấy được process A và process B đang tranh chấp biến x lẫn nhau như trên *hình 3.3* thay vì xuất ra `PA = 4` ngay lập tức thì chương trình lại xuất ra `PA = 7`, lí do là process A đang giữ lại giá trị `x` cũ khi đưa vào thanh ghi và khi process B thực hiện thì giá trị `x` đó không được cập nhật.

Một vấn đề nữa, chương trình với đoạn mã trên không đảm bảo được giá trị `x` sẽ không vượt quá 20. Giả sử lúc `x` đang có giá trị 20, khi thực hiện lệnh `if` thì hết quantum time, sau khi được cấp CPU và thực thi lại thì tiến trình thực thi đúng ngay đoạn `x = x + 1` và tăng lên 21 dẫn đến `x` vượt quá 20.

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

a. Hiện thực

```
int x = 0;
pthread_mutex_t mutex;

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

Hình 4.1. Khai báo và hiện thực chương trình

b. Ý tưởng

Giải quyết hai vấn đề nêu ra ở bài 3 bằng cách đưa đoạn mã xử lý vào trong cặp `mutex_lock` và `mutex_unlock` để các tiến trình không tranh chấp biến `x` lẫn nhau và đảm bảo khi tiến trình thực thi xong, mở khóa thì tiến trình kia mới được phép thực thi.

Kết quả *hình 4.2* và *hình 4.3* xuất ra giá trị `x` ở cả 2 tiến trình một cách có thứ tự và không vượt quá giá trị 20.

c. Kết quả

Hình 4.2. Kết quả (1)

Hình 4.3. Kết quả (2)

5. **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)

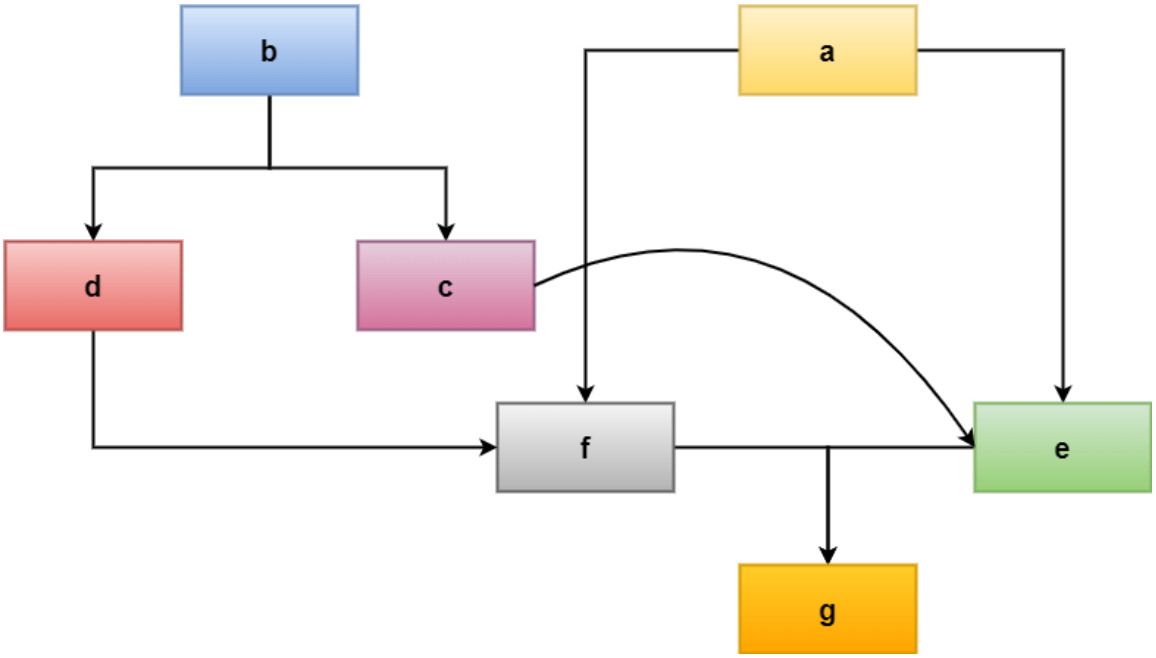
Giả sử các lệnh từ (a) → (g) nằm trên các thread chạy song

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**

a. Sơ đồ mô phỏng



Hình 5.1. Sơ đồ mô phỏng

Sơ đồ trên biểu thị "Điều kiện tiên quyết" để một tiến trình thực thi **MÔ TẢ:** Mũi tên trỏ từ khối **[A]** sang khối **[B]** thể hiện process A thực thi xong thì process B mới được phép thực thi.

b. Hiện thực

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

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

sem_t semA, semB, semC, semD, semEF;
```

Hình 5.2. Khai báo & khởi tạo

```c
void* processA()
{
        w = x1 * x2;
        printf("[Process A] w = %d\n", w);
        sem_post(&semA);
        sem_post(&semA);
}
```

Hình 5.3. Process A

```c
void* processB()
{
        v = x3 * x4;
        printf("[Process B] v = %d\n", v);
        sem_post(&semB);
        sem_post(&semB);
}
```

Hình 5.4. Process B

```c
void* processC()
{
        sem_wait(&semB);
        y = v * x5;
        printf("[Process C] y = %d\n", y);
        sem_post(&semC);
}
```

Hình 5.5. Process C

```
void* processD()
{
        sem_wait(&semB);
        z = v * x6;
        printf("[Process D] z = %d\n", z);
        sem_post(&semD);
}
```

Hình 5.6. Process D

```
void* processE()
{
        sem_wait(&semA);
        sem_wait(&semC);
        y = w * y;
        printf("[Process E] y = %d\n", y);
        sem_post(&semEF);
}
```

Hình 5.7. Process E

```
void* processF()
{
        sem_wait(&semA);
        sem_wait(&semD);
        z = w * z;
        printf("[Process F] z = %d\n", z);
        sem_post(&semEF);
}
```

Hình 5.8. Process F

```
void* processG()
{
        sem_wait(&semEF);
        sem_wait(&semEF);
        ans = y + z;
        printf("[Process G] ans = %d\n", ans);
}
```

Hình 5.9. Process G

```c
int main()
{
        sem_init(&semA, 0, 0);
        sem_init(&semB, 0, 0);
        sem_init(&semC, 0, 0);
        sem_init(&semD, 0, 0);
        sem_init(&semEF, 0, 0);

        printf("Enter x1 value: ");
        scanf("%d", &x1);
        printf("Enter x2 value: ");
        scanf("%d", &x2);
        printf("Enter x3 value: ");
        scanf("%d", &x3);
        printf("Enter x4 value: ");
        scanf("%d", &x4);
        printf("Enter x5 value: ");
        scanf("%d", &x5);
        printf("Enter x6 value: ");
        scanf("%d", &x6);

        printf("\n-----------------------------------------\n");

        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 5.10. Hàm main

c. Ý tưởng

Bài toán trên có nhiều cách giải quyết khác nhau phụ thuộc vào số lượng semaphore được lựa chọn sử dụng. Cách tối đa và đơn giản nhất để giải quyết là sử dụng 8 semaphore cho mỗi điều kiện dựa vào *hình 5.1*.

Trong đoạn chương trình trên, chỉ cần dùng 5 semaphore cho bài toán (có thể chưa tối ưu nhất):

- Process A thực hiện trước và cũng là điều kiện tiên quyết để Process E và Process F được thực hiện, do đó sẽ có `sem_post(&semA)` 2 lần cho mỗi Process
- Tương tự, Process B thực thi không vướng điều kiện và cũng là điều kiện tiên quyết cho 2 Process C và D, do đó sẽ có `sem_post(&semB)` 2 lần cho mỗi Process
- Process C thực hiện cần chờ tín hiệu từ Process B, do đó có lệnh `sem_wait(&semB)`. Ngoài ra Process C là điều kiện tiên quyết cho Process E, do đó có lệnh `sem_post(&semC)` báo hiệu cho phép Process E được phép thực thi
- Process D thực hiện cần chờ tín hiệu từ Process B, do đó có lệnh `sem_wait(&semB)`. Ngoài ra Process D là điều kiện tiên quyết cho Process F, do đó có lệnh `sem_post(&semD)` báo hiệu cho phép Process F được phép thực thi
- Process E thực hiện cần chờ 2 luồng tín hiệu từ Process A và Process C, do đó có 2 lệnh `sem_wait(&semA)` và `sem_wait(&semC)` trước khi thực hiện tính toán. Process E là điều kiện tiên quyết của Process G, do đó có lệnh `sem_post(&semEF)` báo hiệu Process G được phép thực thi
- Process F thực hiện cần chờ 2 luồng tín hiệu từ Process A và Process D, do đó có 2 lệnh `sem_wait(&semA)` và `sem_wait(&semD)` trước khi thực hiện tính toán. Process F là điều kiện tiên quyết của Process G, do đó có lệnh `sem_post(&semEF)` báo hiệu Process G được phép thực thi
- Process G là tiến trình in ra kết quả cuối cùng và chỉ được thực thi khi Process E và Process F đã hoàn thành, do đó có lệnh `sem_wait(&semEF)` 2 lần trước khi in ra màn hình kết quả cuối cùng. Vì nhiệm vụ của Process E và F tương tự nhau nên chỉ cần 1 semaphore chung cho cả hai tiến trình là được

d. Kết quả

TRƯỜNG HỢP 1: CÁC GIÁ TRỊ BAN ĐẦU CÓ THỨ TỰ:

$$x1 = 1$$
$$x2 = 2$$
$$x3 = 3$$
$$x4 = 4$$
$$x5 = 5$$
$$x6 = 6$$

CÁC TIẾN TRÌNH SẼ THỰC THI LẦN LƯỢT:

- Process A: $w = x1 \times x2 = 1 \times 2 = 2$
- Process B: $v = x3 \times x4 = 3 \times 4 = 12$
- Process C: $y = v \times x5 = 12 \times 5 = 60$
- Process D: $z = v \times x6 = 12 \times 6 = 72$
- Process E: $y = w \times y = 2 \times 60 = 120$
- Process F: $z = w \times z = 2 \times 72 = 144$
- Process G: $ans = y + z = 120 + 144 = 264$

Kết quả trùng khớp với kết quả chương trình (*hình 5.11*)

```
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ ./bonus
Enter x1 value: 1
Enter x2 value: 2
Enter x3 value: 3
Enter x4 value: 4
Enter x5 value: 5
Enter x6 value: 6

----------------------------------------
[Process A] w = 2
[Process B] v = 12
[Process C] y = 60
[Process D] z = 72
[Process E] y = 120
[Process F] z = 144
[Process G] ans = 264
```

Hình 5.11. Kết quả (1)

TRƯỜNG HỢP 2: CÁC GIÁ TRỊ BAN ĐẦU NGẪU NHIÊN:

$$x1 = 15$$
$$x2 = 4$$
$$x3 = 2$$
$$x4 = 31$$
$$x5 = 1037$$
$$x6 = 22$$

CÁC TIẾN TRÌNH SẼ THỰC THI LẦN LƯỢT:

- Process A: $w = x1 \times x2 = 15 \times 4 = 60$
- Process B: $v = x3 \times x4 = 2 \times 31 = 62$
- Process C: $y = v \times x5 = 62 \times 1037 = 64294$
- Process D: $z = v \times x6 = 62 \times 22 = 1364$
- Process E: $y = w \times y = 60 \times 64294 = 3857640$
- Process F: $z = w \times z = 60 \times 1364 = 81840$
- Process G: $ans = y + z = 3857640 + 81840 = 3939480$

Kết quả trùng khớp với kết quả chương trình (*hình 5.12*)

```
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ gcc bonus.c -o bonus -lpthread -l
rt
kiet-20521494@kiet20521494-VirtualBox:~/LAB5$ ./bonus
Enter x1 value: 15
Enter x2 value: 4
Enter x3 value: 2
Enter x4 value: 31
Enter x5 value: 1037
Enter x6 value: 22

----------------------------------------
[Process A] w = 60
[Process B] v = 62
[Process C] y = 64294
[Process D] z = 1364
[Process E] y = 3857640
[Process F] z = 81840
[Process G] ans = 3939480
```

Hình 5.12. Kết quả (2)

HẾT