

ĐẠI HỌC QUỐC GIA TP HCM

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN



KHOA KỸ THUẬT MÁY TÍNH

HỆ ĐIỀU HÀNH

---

## Báo cáo thực hành LAB 5

---

### Môn học: Hệ điều hành

Sinh viên thực hiện:

- Đặng Quốc Cường
- MSSV : 23520192
- Lớp : IT007.P11.CTTN

Giảng viên hướng dẫn:

Thân Thế Tùng

Ngày 20 tháng 11 năm 2024

## Mục lục

<b>1 TỔNG QUAN</b>	<b>2</b>
<b>2 BÀI 1</b>	<b>2</b>
<b>3 BÀI 2</b>	<b>4</b>
3.1 Chương trình chưa được đồng bộ . . . . .	5
3.2 Chương trình đã được đồng bộ . . . . .	7
<b>4 BÀI 3</b>	<b>9</b>
<b>5 BÀI 4</b>	<b>11</b>
<b>6 BÀI 5 (ÔN TẬP)</b>	<b>13</b>
<b>7 Phụ lục</b>	<b>16</b>

# 1 TỔNG QUAN

Tiêu chí	1	2	3	4	5
Trình bày cách làm	✓	✓	✓	✓	✓
Chụp hình minh chứng	✓	✓	✓	✓	✓
Giải thích kết quả	✓	✓	✓	✓	✓

TỰ CHẤM ĐIỂM : 10

## 2 BÀI 1

**Yêu cầu :** Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau:

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

```
1  /*#####
2  # University of Information Technology
3  # IT007 Operating System
4  #
5  # <Dang Quoc Cuong>, <23520192>
6  # File: BAI1.cpp
7  #
8  #####*/
9  #include <pthread.h>
10 #include <semaphore.h>
11 #include <stdio.h>
12 #include <unistd.h>
13
14 int sells = 0, products = 0; // Bien toan cuc
15 sem_t sem, sem1;           // Semaphore de dong bo hoa
16
17 // Luong A: Xy ly sells
18 void *processA(void *mess) {
19     while (1) { // Vong lap vo han
20         sem_wait(&sem); // Doi semaphore "sem" duoc mo
21         sells++;        // Tang gia tri cua sells
22         printf("SELL = %d\n", sells); // In gia tri cua sells
23         printf("SELL_1 (SELL + 192) = %d\n", sells + 192); // In SELL + 192
24         sem_post(&sem1); // Mo semaphore "sem1" de luong B chay
25         //usleep(100000); // Nghi 0.1 giay de giam tai CPU
26     }
27     return NULL;
28 }
29
30 // Luong B: Xy ly products
31 void *processB(void *mess) {
32     while (1) { // Vong lap vo han
33         sem_wait(&sem1); // Doi semaphore "sem1" duoc mo
34         products++;      // Tang gia tri cua products
```

```
35     printf("PRODUCT = %d\n", products); // In gia tri cua products
36     sem_post(&sem); // Mo semaphore "sem" de luong A chay
37     //usleep(100000); // Nghi 0.1 giay de giam tai CPU
38 }
39 return NULL;
40 }
41
42 int main() {
43     // Khoi tao semaphore
44     sem_init(&sem, 0, 0); // Khoi tao semaphore "sem", gia tri ban dau
45     = 0
46     sem_init(&sem1, 0, sells + 192); // Khoi tao semaphore "sem1", gia tri ban
47     dau = sells + 192
48
49     pthread_t pA, pB; // Khai bao hai luong
50
51     // Tao hai luong
52     pthread_create(&pA, NULL, processA, NULL); // Tao luong A
53     pthread_create(&pB, NULL, processB, NULL); // Tao luong B
54
55     // Vong lap chinh chay vo han
56     while (1) {
57         // Chuong trinh van hoat dong de cac luong chay
58     }
59
60     // Huy semaphore khi chuong trinh ket thuc
61     sem_destroy(&sem);
62     sem_destroy(&sem1);
63
64     return 0;
65 }
```

Chương trình sử dụng **semaphore** để đồng bộ hai luồng **processA** và **processB**. Semaphore đảm bảo rằng hai luồng thực thi luân phiên, không bị xung đột.

Luồng **processA** xử lý biến **sells**, chờ semaphore **sem** để chạy, tăng giá trị **sells**, in kết quả, sau đó mở semaphore **sem1** cho luồng **processB**. Ngược lại, luồng **processB** xử lý biến **products**, chờ semaphore **sem1**, tăng giá trị **products**, in kết quả, rồi mở semaphore **sem** để quay lại luồng **processA**.

Hàm **main** khởi tạo các semaphore ban đầu:

- **sem** là 0 (luồng A chờ).
- **sem1** là 192 (luồng B chạy trước).

Sau đó, chương trình tạo hai luồng thực thi đồng thời. Hai luồng luân phiên chạy, đồng bộ qua việc mở và chờ semaphore. Chương trình chạy vô hạn với luồng A và B phối hợp chặt chẽ thông qua semaphore.

```
PRODUCT = 378258
SELL_1 (SELL + 192) = 378258
SELL = 378067
SELL_1 (SELL + 192) = 378259
SELL = 378068
SELL_1 (SELL + 192) = 378260
SELL = 378069
PRODUCT = 378254
PRODUCT = 378255
PRODUCT = 378256
PRODUCT = 378257
PRODUCT = 378258
PRODUCT = 378259
PRODUCT = 378260
SELL_1 (SELL + 192) = 378261
SELL = 378070
SELL_1 (SELL + 192) = 378262
SELL = 378071
SELL_1 (SELL + 192) = 378263
SELL = 378072
SELL_1 (SELL + 192) = 378264
SELL = 378073
PRODUCT = 378261
PRODUCT = 378262
PRODUCT = 378263
PRODUCT = 378264
SELL_1 (SELL + 192) = 378265
SELL = 378074
```

Hình 1: Kết quả bài 1

## Nhận xét

Kết quả cho thấy hai luồng hoạt động luân phiên, đảm bảo đồng bộ hóa:

- **Luồng A (SELL):** Tăng tuần tự giá trị `sells` và tính toán `sells + 192`.
- **Luồng B (PRODUCT):** Tăng tuần tự giá trị `products`.
- Semaphore đảm bảo thứ tự thực thi, tránh xung đột giữa hai luồng.

## 3 BÀI 2

Cho một mảng `a` được khai báo như một mảng số nguyên có thể chứa tối đa `n` phần tử, và 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 mảng. 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 bất kỳ từ mảng (phụ thuộc vào người lập trình). Sau đó đếm và xuất ra số phần tử của `a` còn lại sau khi lấy ra. Nếu không có phần tử nào 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.

### 3.1 Chương trình chưa được đồng bộ

```
1  /*#####
2  # University of Information Technology
3  # IT007 Operating System
4  #
5  # <Dang Quoc Cuong>, <23520192>
6  # File: BAI2.cpp not synchronized
7  #
8  #####*/
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <pthread.h>
12
13 int i = 0, a[100], n;
14
15 void *processA(void *arr) {
16     while (1) {
17         int temp = rand() % 100; // Sinh so ngau nhien
18         a[++i] = temp; // Them vao mang
19
20         printf("\nThem vao mang: %d", temp);
21         printf("\nKich thuoc mang a la: %d\n", i);
22
23         for (int k = 0; k < i; k++) {
24             printf("%d ", a[k]); // In cac phan tu trong mang
25         }
26         printf("\n");
27     }
28 }
29
30 void *processB(void *arr) {
31     while (1) {
32         if (i <= 0) {
33             printf("\nNothing in array a\n");
34             continue;
35         }
36
37         printf("\nLoai khoi mang: %d", a[i - 1]); // Lay phan tu cuoi
38         i--; // Giam chi so i
39         printf("\nKich thuoc mang a la: %d\n", i);
40
41         for (int k = 0; k < i; k++) {
42             printf("%d ", a[k]); // In cac phan tu trong mang
43         }
44         printf("\n");
45     }
46 }
47
48 int main() {
49     printf("Nhap so phan tu: ");
```

```
50     scanf("%d", &n);
51
52     pthread_t pA, pB;
53     pthread_create(&pA, NULL, &processA, NULL); // Tao luồng A
54     pthread_create(&pB, NULL, &processB, NULL); // Tao luồng B
55
56     while (1) {
57         // Chương trình chạy liên tục
58     }
59
60     return 0;
61 }
```

```
Kích thước mảng a là: 213
0 6 77 33 98 16 14 32 19 36 19 99 49 92 49 10 55 91 15 15 57 38 61 67 78 59 41 23 95 2 1 50 55 78 8 8
0 74 37 6 29 3 58 62 15 25 26 59 3 60 4 22 10 42 70 63 94 96 73 52 93 74 87 97 44 37 80 71 14 77 28 2
3 66 80 95 75 97 0 78 88 92 30 23 69 63 81 92 29 19 0 10 86 89 66 28 57 98 6 77 63 39 16 52 57 45 61
55 20 77 24 17 90 82 14 55 87 38 17 92 92 19 25 36 21 28 15 61 12 44 18 10 45 46 33 96 64 90 1 14 33
45 79 86 3 1 8 54 94 79 16 14 27 26 28 71 0 40 70 20 47 84 91 66 29 4 46 32 8 9 21 32 74 73 77 43 92
3 42 97 88 88 94 6 12 89 41 35 76 9 84 86 27 26 22 75 7 8 1 86 63 87 82 13 0 34 74 55 17 58 49 38 27
20 23

Loại khỏi mảng: 23
Kích thước mảng a là: 212
0 6 77 33 98 16 14 32 19 36 19 99 49 92 49 10 55 91 15 15 57 38 61 67 78 59 41 23 95 2 1 50 55 78 8 8
0 74 37 6 29 3 58 62 15 25 26 59 3 60 4 22 10 42 70 63 94 96 73 52 93 74 87 97 44 37 80 71 14 77 28 2
3 66 80 95 75 97 0 78 88 92 30 23 69 63 81 92 29 19 0 10 86 89 66 28 57 98 6 77 63 39 16 52 57 45 61
55 20 77 24 17 90 82 14 55 87 38 17 92 92 19 25 36 21 28 15 61 12 44 18 10 45 46 33 96 64 90 1 14 33
45 79 86 3 1 8 54 94 79 16 14 27 26 28 71 0 40 70 20 47 37 84 6 29 3 91 58 62 15 66 25 29 4 26 46 59
32 3 60 4 8 22 9 21 32 10 74 42 70 63 73 77 94 43 96 73 52 92 93 3 74 42 87 97 44 97 37 88 80 71 14 8
8 77 94 28 23 66 6 12 80 89 95 75 97 41 0 35 78 76 9 88 84 92 30 23 86 69 63 81 27 92 29 26 19 22 0 7
5 10 7 8 86 1 89 66 86 28 63 57 87 98 82 6 77 13 63 39 16 0 52 34 57 45 61 74 55 20 77 55 24 17 17 58
90 82 14 49 55 87 38 38 17 27 92 92 19 20
```

Hình 2: Kết quả bài 2 khi chưa đồng bộ

## Giải thích

Khi chương trình chưa sử dụng cơ chế đồng bộ hóa, các lỗi sau sẽ xảy ra:

- Race condition xảy ra khi hai luồng cùng thao tác trên biến toàn cục `i` và mảng `a` mà không có cơ chế bảo vệ, dẫn đến dữ liệu không nhất quán. Một luồng có thể đang thêm phần tử, trong khi luồng kia lại xóa phần tử hoặc thay đổi giá trị `i`, gây ra kết quả sai hoặc lỗi runtime.
- Biến `i` không được kiểm soát đúng cách có thể dẫn đến lỗi truy cập ngoài phạm vi mảng. Nếu `i` vượt quá kích thước tối đa (100) hoặc âm, chương trình có thể ghi đè hoặc truy cập vào vùng nhớ không hợp lệ, dẫn đến lỗi như "segmentation fault".
- Hai luồng chạy liên tục trong các vòng lặp vô hạn mà không có thời gian nghỉ (`sleep`) hoặc cơ chế chờ, khiến CPU bị sử dụng 100%. Điều này làm giảm hiệu năng hệ thống và gây ra tình trạng gián đoạn cho các ứng dụng khác.
- Trong luồng tiêu thụ, nếu mảng rỗng, chương trình liên tục in thông báo "Nothing in array a" mà không chờ mảng được thêm phần tử, gây ra spam thông báo vô nghĩa.

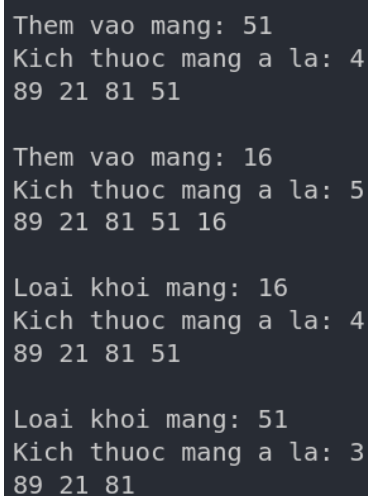
- Những lỗi này dẫn đến chương trình hoạt động không ổn định, dữ liệu sai lệch và hiệu năng kém, cần được khắc phục bằng cơ chế đồng bộ hóa như mutex hoặc semaphore.

### 3.2 Chương trình đã được đồng bộ

```
1  /*#####
2  /*#####
3  # University of Information Technology
4  # IT007 Operating System
5  #
6  # <Dang Quoc Cuong>, <23520192>
7  # File: BAI2.cpp synchronized
8  #
9  /*#####*/
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <pthread.h>
13 #include <semaphore.h>
14
15 sem_t sem1, sem2;
16 pthread_mutex_t pt1;
17 int i = 0, a[100], n;
18
19 void *processA(void *arr) {
20     while (1) {
21         sem_wait(&sem2); // Doi semaphore sem2
22         pthread_mutex_lock(&pt1); // Khoa mutex de tranh xung dot
23
24         int temp = rand() % 100; // Sinh so ngau nhien
25         a[i++] = temp; // Them vao mang
26
27         printf("\nThem vao mang: %d", temp);
28         printf("\nKich thuoc mang a la: %d\n", i);
29
30         for (int k = 0; k < i; k++) {
31             printf("%d ", a[k]); // In cac phan tu trong mang
32         }
33         printf("\n");
34
35         pthread_mutex_unlock(&pt1); // Mo khoa mutex
36         sem_post(&sem1); // Mo semaphore sem1
37     }
38 }
39
40 void *processB(void *arr) {
41     while (1) {
42         if (i <= 0) {
43             printf("\nNothing in array a\n");
44             continue;
45         }
46
47         sem_wait(&sem1); // Doi semaphore sem1
48         pthread_mutex_lock(&pt1); // Khoa mutex de tranh xung dot
49     }
```



```
50     printf("\nLoai khoi mang: %d", a[i - 1]); // Lay phan tu cuoi
51     i--; // Giam chi so i
52     printf("\nKich thuoc mang a la: %d\n", i);
53
54     for (int k = 0; k < i; k++) {
55         printf("%d ", a[k]); // In cac phan tu trong mang
56     }
57     printf("\n");
58
59     pthread_mutex_unlock(&pt1); // Mo khoa mutex
60     sem_post(&sem2); // Mo semaphore sem2
61 }
62 }
63
64 int main() {
65     printf("Nhap so phan tu: ");
66     scanf("%d", &n);
67
68     pthread_mutex_init(&pt1, NULL); // Khoi tao mutex
69     sem_init(&sem1, 0, 0); // Khoi tao semaphore sem1
70     sem_init(&sem2, 0, n); // Khoi tao semaphore sem2
71
72     pthread_t pA, pB;
73     pthread_create(&pA, NULL, &processA, NULL); // Tao luong A
74     pthread_create(&pB, NULL, &processB, NULL); // Tao luong B
75
76     while (1) {
77         // Chuong trinh chay lien tuc
78     }
79
80     return 0;
81 }
```



```
Them vao mang: 51
Kich thuoc mang a la: 4
89 21 81 51

Them vao mang: 16
Kich thuoc mang a la: 5
89 21 81 51 16

Loai khoi mang: 16
Kich thuoc mang a la: 4
89 21 81 51

Loai khoi mang: 51
Kich thuoc mang a la: 3
89 21 81
```

Hình 3: Kết quả bài 2 đã đồng bộ

Chương trình sử dụng hai luồng song song:

- **Luồng processA (thêm phần tử):** Sinh số ngẫu nhiên, thêm vào mảng, in kết quả, và cho phép luồng processB chạy thông qua semaphore.
- **Luồng processB (xóa phần tử):** Lấy phần tử cuối mảng, giảm kích thước mảng, in kết quả, và cho phép luồng processA tiếp tục.

Sử dụng các cơ chế đồng bộ hóa:

- **Semaphore:** Đồng bộ thứ tự giữa hai luồng (chỉ cho phép chạy khi điều kiện phù hợp, như mảng không rỗng hoặc chưa đầy).
- **Mutex:** Bảo vệ dữ liệu dùng chung (mảng và biến i) để tránh xung đột khi hai luồng thao tác đồng thời.

Kết quả: Hai luồng hoạt động luân phiên, đảm bảo tính chính xác và đồng bộ.

## 4 BÀI 3

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

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

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

```
1 /*#####
2 # University of Information Technology
3 # IT007 Operating System
4 #
5 # <Dang Quoc Cuong>, <23520192>
6 # File: BAI3.cpp
7 #
8 #####*/
9 #include <semaphore.h>
10 #include <stdio.h>
11 #include <pthread.h>
12
```

```
13 int x = 0;
14
15 void *processA(void *mess) {
16     while (1) {
17         x = x + 1; // Tang x
18         if (x == 20) {
19             x = 0; // Reset x ve 0 neu x bang 20
20         }
21         printf("A = %d\n", x); // In gia tri x
22     }
23 }
24
25 void *processB(void *mess) {
26     while (1) {
27         x = x + 1; // Tang x
28         if (x == 20) {
29             x = 0; // Reset x ve 0 neu x bang 20
30         }
31         printf("B = %d\n", x); // In gia tri x
32     }
33 }
34
35 int main() {
36     pthread_t pA, pB; // Khai bao cac luong
37
38     pthread_create(&pA, NULL, &processA, NULL); // Tao luong A
39     pthread_create(&pB, NULL, &processB, NULL); // Tao luong B
40
41     while (1) {
42         // Chuong trinh chay vo han
43     }
44
45     return 0;
46 }
```

```
A = 19
A = 0
A = 1
A = 2
A = 3
A = 4
B = 14
B = 6
B = 7
B = 8
B = 9
B = 10
B = 11
B = 12
B = 13
B = 14
B = 15
B = 16
B = 17
B = 18
B = 19
B = 0
B = 1
```

Hình 4: Kết quả bài 3

Lỗi xảy ra khi giá trị của  $x$  nhảy từ  $A = 4$  sang  $B = 14$  là do **race condition**, tình trạng hai luồng `processA` và `processB` cùng truy cập và thay đổi biến toàn cục  $x$  mà không có cơ chế đồng bộ hóa. Cụ thể, khi luồng `processA` tăng giá trị của  $x$  lên 4 và chuẩn bị in giá trị tiếp theo, luồng `processB` được CPU ưu tiên chạy và thực hiện tăng giá trị  $x$  liên tục, bỏ qua các bước từ 5 đến 13. Điều này dẫn đến sự không đồng nhất trong việc thao tác trên  $x$ , làm giá trị nhảy cóc và không đúng thứ tự mong muốn. Lỗi này phát sinh do không có cơ chế bảo vệ tài nguyên dùng chung giữa hai luồng, như mutex hoặc semaphore, để đảm bảo chỉ một luồng truy cập và thay đổi  $x$  tại một thời điểm. Kết quả là chương trình in ra giá trị không liên tục, gây ra sai lệch dữ liệu.

## 5 BÀI 4

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

```
1  /*#####
2  # University of Information Technology
3  # IT007 Operating System
4  #
5  # <Dang Quoc Cuong>, <23520192>
6  # File: BAI4.cpp (synchronized)
7  #
8  #####*/
9  #include <semaphore.h>
10 #include <stdio.h>
11 #include <pthread.h>
12
13 int x = 0; // Bien toan cuc
14 pthread_mutex_t lock; // Mutex de dong bo
15
16 void *processA(void *mess) {
17     while (1) {
18         pthread_mutex_lock(&lock); // Khoa mutex de tranh xung dot
19
20         x = x + 1; // Tang x
21         if (x == 20) {
22             x = 0; // Reset x ve 0 neu x bang 20
23         }
24         printf("A = %d\n", x); // In gia tri x
25
26         pthread_mutex_unlock(&lock); // Mo khoa mutex
27     }
28 }
29
30 void *processB(void *mess) {
31     while (1) {
32         pthread_mutex_lock(&lock); // Khoa mutex de tranh xung dot
33
34         x = x + 1; // Tang x
35         if (x == 20) {
36             x = 0; // Reset x ve 0 neu x bang 20
37         }
38         printf("B = %d\n", x); // In gia tri x
39
40         pthread_mutex_unlock(&lock); // Mo khoa mutex
```

```
41     }  
42 }  
43  
44 int main() {  
45     pthread_t pA, pB; // Khai bao cac luong  
46  
47     pthread_mutex_init(&lock, NULL); // Khoi tao mutex  
48  
49     pthread_create(&pA, NULL, &processA, NULL); // Tao luong A  
50     pthread_create(&pB, NULL, &processB, NULL); // Tao luong B  
51  
52     while (1) {  
53         // Chuong trinh chay vo han  
54     }  
55  
56     pthread_mutex_destroy(&lock); // Huy mutex khi ket thuc  
57     return 0;  
58 }
```

```
A = 1  
A = 2  
A = 3  
A = 4  
A = 5  
A = 6  
A = 7  
A = 8  
A = 9  
A = 10  
A = 11  
A = 12  
B = 13  
B = 14  
B = 15  
B = 16  
B = 17  
A = 18  
A = 19
```

Hình 5: Kết quả bài 4

Hai luồng `processA` và `processB` hoạt động đồng bộ, giá trị  $x$  sẽ được in ra liên tục và đúng thứ tự mà không có lỗi nhảy cóc hoặc xung đột.

## 6 BÀI 5 (ÔN TẬP)

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

```
1  /*#####
2  # University of Information Technology
3  # IT007 Operating System
4  #
5  # <Dang Quoc Cuong>, <23520192>
6  # File: BAI5.cpp
7  #
8  #####*/
9  #include <stdio.h>
10 #include <pthread.h>
11 #include <semaphore.h>
12
13 // Bien toan cuc
14 int x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5, x6 = 6; // Bien dau vao
15 int w, v, y, z, ans; // Bien trung gian va ket qua
16
17 // Semaphore dong bo hoa
18 sem_t sem_v, sem_y, sem_z, sem_e, sem_f, sem_g;
19
20 // Ham tinh w = x1 * x2
21 void *compute_w(void *arg) {
22     w = x1 * x2;
23     printf("w = %d (x1 * x2)\n", w);
24     sem_post(&sem_e); // Mo khoa de phep tinh (e) thuc hien
25     sem_post(&sem_f); // Mo khoa de phep tinh (f) thuc hien
26     return NULL;
27 }
28
29 // Ham tinh v = x3 * x4
30 void *compute_v(void *arg) {
31     v = x3 * x4;
32     printf("v = %d (x3 * x4)\n", v);
33     sem_post(&sem_y); // Mo khoa de phep tinh (c) thuc hien
34     sem_post(&sem_z); // Mo khoa de phep tinh (d) thuc hien
35     return NULL;
36 }
37
38 // Ham tinh y = v * x5
39 void *compute_y(void *arg) {
40     sem_wait(&sem_y); // Cho v duoc tinh
41     y = v * x5;
42     printf("y = (v * x5) = %d\n", y);
43     sem_post(&sem_e); // Mo khoa de phep tinh (e) thuc hien
44     return NULL;
45 }
46
47 // Ham tinh z = v * x6
48 void *compute_z(void *arg) {
49     sem_wait(&sem_z); // Cho v duoc tinh
50     z = v * x6;
51     printf("z = (v * x6) = %d\n", z);
```

```
52     sem_post(&sem_f); // Mo khoa de phep tinh (f) thuc hien
53     return NULL;
54 }
55
56 // Ham tinh y = w * y
57 void *compute_e(void *arg) {
58     sem_wait(&sem_e); // Cho w duoc tinh
59     sem_wait(&sem_e); // Cho y duoc tinh
60     y = w * y;
61     printf("y = (w * y) = %d\n", y);
62     sem_post(&sem_g); // Mo khoa de phep tinh (g) thuc hien
63     return NULL;
64 }
65
66 // Ham tinh z = w * z
67 void *compute_f(void *arg) {
68     sem_wait(&sem_f); // Cho w duoc tinh
69     sem_wait(&sem_f); // Cho z duoc tinh
70     z = w * z;
71     printf("z = (w * z) = %d\n", z);
72     sem_post(&sem_g); // Mo khoa de phep tinh (g) thuc hien
73     return NULL;
74 }
75
76 // Ham tinh ans = y + z
77 void *compute_g(void *arg) {
78     sem_wait(&sem_g); // Cho y duoc tinh
79     sem_wait(&sem_g); // Cho z duoc tinh
80     ans = y + z;
81     printf("ans = %d (y + z)\n", ans);
82     return NULL;
83 }
84
85 int main() {
86     pthread_t t_w, t_v, t_y, t_z, t_e, t_f, t_g;
87
88     // In ra cac bien dau vao
89     printf("x1 = %d, x2 = %d, x3 = %d, x4 = %d, x5 = %d, x6 = %d\n", x1, x2, x3,
90         x4, x5, x6);
91
92     // Khoi tao semaphore
93     sem_init(&sem_v, 0, 0);
94     sem_init(&sem_y, 0, 0);
95     sem_init(&sem_z, 0, 0);
96     sem_init(&sem_e, 0, 0);
97     sem_init(&sem_f, 0, 0);
98     sem_init(&sem_g, 0, 0);
99
100    // Tao cac thread
101    pthread_create(&t_w, NULL, compute_w, NULL);
102    pthread_create(&t_v, NULL, compute_v, NULL);
103    pthread_create(&t_y, NULL, compute_y, NULL);
104    pthread_create(&t_z, NULL, compute_z, NULL);
105    pthread_create(&t_e, NULL, compute_e, NULL);
106    pthread_create(&t_f, NULL, compute_f, NULL);
```

```
106 pthread_create(&t_g, NULL, compute_g, NULL);
107
108 // Cho cac thread ket thuc
109 pthread_join(t_w, NULL);
110 pthread_join(t_v, NULL);
111 pthread_join(t_y, NULL);
112 pthread_join(t_z, NULL);
113 pthread_join(t_e, NULL);
114 pthread_join(t_f, NULL);
115 pthread_join(t_g, NULL);
116
117 // Huy semaphore
118 sem_destroy(&sem_v);
119 sem_destroy(&sem_y);
120 sem_destroy(&sem_z);
121 sem_destroy(&sem_e);
122 sem_destroy(&sem_f);
123 sem_destroy(&sem_g);
124
125 // In ket qua cuoi cung
126 printf("Ket qua cuoi cung: y = %d, z = %d, ans = %d\n", y, z, ans);
127
128 return 0;
129 }
```

```
● dplayergod [ ] main [ ] cd "/home/dplayergod/UIT/OS/LAB/LAB05/" && g++ BAI5.cpp
B/LAB05/"BAI5
x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5, x6 = 6
w = 2 (x1 * x2)
v = 12 (x3 * x4)
y = (v * x5) = 60
z = (v * x6) = 72
y = (w * y) = 120
z = (w * z) = 144
ans = 264 (y + z)
Kết quả cuối cùng: y = 120, z = 144, ans = 264
```

Hình 6: Kết quả bài 5

## Phân tích bài toán

Các phép tính có thứ tự phụ thuộc như sau:

- (a)  $w = x1 * x2$  và (b)  $v = x3 * x4$  không phụ thuộc vào nhau, nên có thể chạy song song.
- (c)  $y = v * x5$  và (d)  $z = v * x6$  phụ thuộc vào (b) (khi v đã được tính), nên chỉ chạy sau khi (b) hoàn thành.
- (e)  $y = w * y$  chỉ thực hiện khi cả (a) (w) và (c) (y) đã hoàn thành.
- (f)  $z = w * z$  chỉ thực hiện khi cả (a) (w) và (d) (z) đã hoàn thành.
- (g)  $ans = y + z$  chỉ thực hiện khi (e) và (f) đã hoàn thành.



## Đồng bộ hóa bằng semaphore

Sử dụng semaphore để kiểm soát thứ tự thực thi giữa các luồng:

- `sem_y` đảm bảo (c) chỉ chạy sau (b).
- `sem_z` đảm bảo (d) chỉ chạy sau (b).
- `sem_e` đảm bảo (e) chỉ chạy sau khi (a) và (c) hoàn thành.
- `sem_f` đảm bảo (f) chỉ chạy sau khi (a) và (d) hoàn thành.
- `sem_g` đảm bảo (g) chỉ chạy sau khi (e) và (f) hoàn thành.

## Chia công việc thành các luồng

- Mỗi phép tính được thực hiện trong một luồng riêng biệt (`compute_w`, `compute_v`, `compute_y`, `compute_z`, `compute_e`, `compute_f`, `compute_g`).
- Semaphore đồng bộ hóa đảm bảo các luồng thực thi theo đúng thứ tự phụ thuộc.

## Thực thi song song

- Các phép tính không phụ thuộc, như (a) và (b), hoặc (c) và (d), có thể chạy song song để tận dụng tối đa tài nguyên CPU.
- Các phép tính phụ thuộc được thực thi khi semaphore cho phép, đảm bảo không xảy ra lỗi hoặc truy cập dữ liệu chưa tính toán xong.

## 7 Phụ lục

- Liên kết github : [link](#).