

Báo cáo thực hành môn Hệ điều hành - Giảng viên: Lê Hoài Nghĩa.

Họ và tên	MSSV	Lớp
Lại Quan Thiên	22521385	IT007.O211.1
Đặng Đức Tài	22521270	
Mai Nguyễn Nam Phương	22521164	
Phùng Trần Thế Nam	21522366	

HỆ ĐIỀU HÀNH BÁO CÁO LAB 5

CHECKLIST

5.5. BÀI TẬP THỰC HÀNH

Phân công:

	BT 1	BT 2	BT 3	BT 4	BTOT
Lại Quan Thiên					<input checked="" type="checkbox"/>
Đặng Đức Tài		<input checked="" type="checkbox"/>			
Mai Nguyễn Nam Phương			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Phùng Trần Thế Nam	<input checked="" type="checkbox"/>				

	BT 1	BT 2	BT 3	BT 4
Trình bày cách làm	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Chụp hình minh chứng	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Giải thích kết quả	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

5.6. BÀI TẬP ÔN TẬP

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

Tự chấm điểm: 10/10

**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:

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

(MSSV: 21522366)

* Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t sem;
int sells = 0, products = 0;
void* PROCESSA(void* arg) {
    while (1) {
        sem_wait(&sem);
        sells++;
        printf("sells = %d\n", sells);
        sleep(2);
    }
}
void* PROCESSB(void* arg) {
    while (1) {
        if (products <= sells + 2366) {
            products++;
            printf("products = %d\n", products);
            sem_post(&sem);
            sleep(1);
        }
    }
}
int main() {
    sem_init(&sem, 0, 0);
    pthread_t th1, th2;
    pthread_create(&th1, NULL, &PROCESSA, NULL);
    pthread_create(&th2, NULL, &PROCESSB, NULL);
    while (1)
        ;
    return 0;
}
```

* Giải thích:

- Khai báo thư viện và biến:

+ Chương trình bắt đầu bằng việc khai báo các thư viện cần thiết bao gồm stdio.h, stdlib.h, pthread.h, semaphore.h và unistd.h (để sử dụng hàm sleep).

Báo cáo thực hành môn Hệ điều hành - Giảng viên: Lê Hoài Nghĩa.

- + Biến toàn cục `sells` và `products` được khai báo để đếm số lượng hàng bán được và số lượng sản phẩm đã được tạo ra.

- Hàm `PROCESSA`:

- + Hàm này chạy trong một luồng riêng biệt.
- + Trong vòng lặp vô hạn nó sử dụng `sem_wait` để chờ đợi tín hiệu từ luồng `PROCESSB`.
- + Khi nhận được tín hiệu nó tăng biến `sells` lên 1 đơn vị và in ra giá trị của `sells`.
- + Sau đó chờ 2 giây trước khi lặp lại.

- Hàm `PROCESSB`:

- + Hàm này cũng chạy trong một luồng riêng biệt.
- + Trong vòng lặp vô hạn nó kiểm tra điều kiện `products <= sells + 2366`.
- + Nếu điều kiện này được thỏa mãn, nó tăng biến `products` lên 1 đơn vị in ra giá trị của `products` và gửi một tín hiệu tới semaphore `sem` bằng `sem_post`.
- + Sau đó chờ 1 giây trước khi lặp lại.

- Hàm `main`:

- + Trong hàm `main`, semaphore, và `sem` được khởi tạo với giá trị ban đầu là 0 bằng `sem_init`.
- + Hai luồng mới được tạo bằng `pthread_create`, một cho `PROCESSA` và một cho `PROCESSB`.

=> Điều này tạo ra một môi trường mô phỏng việc bán hàng và sản xuất hàng, trong đó hai luồng hoạt động song song và tương tác với nhau thông qua một semaphore.

*** Màn hình kết quả:**

```
• thenam@21522366:~/Documents$ nano B1.cpp
• thenam@21522366:~/Documents$ g++ -o B1 B1.cpp
○ thenam@21522366:~/Documents$ ./B1
products = 1
sells = 1
products = 2
sells = 2
products = 3
products = 4
sells = 3
products = 5
products = 6
sells = 4
products = 7
products = 8
sells = 5
products = 9
products = 10
sells = 6
products = 11
products = 12
sells = 7
products = 13
products = 14
sells = 8
products = 15
products = 16
sells = 9
products = 17
products = 18
sells = 10
products = 19
products = 20
sells = 11
products = 21
products = 22
sells = 12
products = 23
products = 24
sells = 13
products = 25
products = 26
sells = 14
products = 27
products = 28
sells = 15
products = 29
products = 30
sells = 16
products = 31
products = 32
sells = 17
products = 33
products = 34
sells = 18
products = 35
products = 36
sells = 19
products = 37
```

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.

Trả lời:

- **Chương trình khi chưa được đồng bộ:**

```
C Baiz.c > @main()
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<time.h>
4 #include<pthread.h>
5
6 int* a;
7 int n;
8 int iNum = 0;
9
10 void Arrange(int* a, int x) {
11     if (x == iNum) {
12         iNum--;
13     } else {
14         for (int i = x; i < iNum - 1; i++) {
15             a[i] = a[i+1];
16         }
17         iNum--;
18     }
19 }
20
21 void* ProcessA(void* mess) {
22     while(1) {
23         srand((int)time(0));
24         a[iNum] = rand();
25         iNum++;
26         printf("So phan tu trong a la: %d\n", iNum);
27     }
28 }
29
30 void* ProcessB(void* mess) {
31     while(1) {
32         srand((int)time(0));
33         if (iNum == 0) {
34             printf("Nothing in array a\n");
35         } else {
36             int r = rand() % iNum;
37             Arrange(a, r);
38             printf("So phan tu con lai trong a: %d\n", iNum);
39         }
40     }
41 }
```

Output of ProcessA:

```
So phan tu trong a la: 329
So phan tu trong a la: 330
So phan tu trong a la: 331
So phan tu trong a la: 332
So phan tu trong a la: 333
So phan tu trong a la: 334
So phan tu trong a la: 335
So phan tu trong a la: 336
So phan tu trong a la: 337
So phan tu trong a la: 338
So phan tu trong a la: 339
So phan tu trong a la: 340
So phan tu trong a la: 341
```

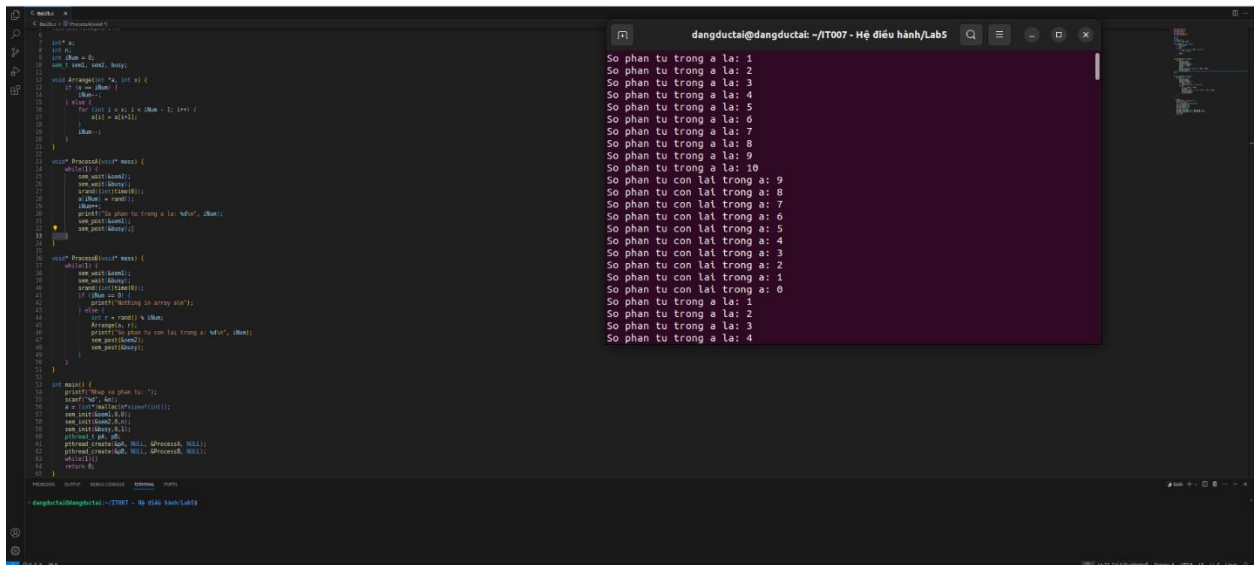
Output of ProcessB:

```
So phan tu con lai trong a: 340
So phan tu con lai trong a: 339
So phan tu con lai trong a: 338
So phan tu con lai trong a: 337
So phan tu con lai trong a: 336
So phan tu con lai trong a: 335
So phan tu con lai trong a: 334
So phan tu con lai trong a: 333
So phan tu con lai trong a: 332
So phan tu con lai trong a: 331
So phan tu con lai trong a: 330
```

*** Giải thích:** Hàm Arrange có chức năng sắp xếp lại hàm khi lấy ngẫu nhiên một phần tử trong mảng a ra. Vì chưa có semaphore nên chương trình chưa được đồng bộ. Xảy ra lỗi do B lấy ra kích thước của mảng A khi A chưa được vào chương trình.

=> Cần dùng semaphore để xử lí.

- **Chương trình sau khi được thêm semaphore:**



```
int a;
int sem1 = 0;
int sem2 = 0;
int N = 10;

void processA(int *a, int N) {
    for (int i = 0; i < N; i++) {
        a[i] = rand();
        printf("Process A: %d\n", i);
    }
}

void processB(int *a, int N) {
    for (int i = 0; i < N; i++) {
        printf("Process B: %d\n", i);
        a[i] = rand();
    }
}

int main() {
    sem_t sem1, sem2;
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 1);
    processA(a, N);
    processB(a, N);
    return 0;
}
```

*** Giải thích:**

- Biến sem1 đóng vai trò điều kiện đảm bảo rằng sẽ không thể lấy phần tử trong A khi không có phần tử nào.
- Biến sem2 đóng vai trò điều kiện đảm bảo rằng sẽ không thể thêm phần tử quá số lượng phần tử được cho phép (mỗi lần chỉ lấy 1 phần tử).
- Khi chương trình được nạp, giả sử ProcessB được nạp trước sẽ bị khóa bởi biến sem1 (khởi tạo = 0). Khi đó ProcessA được chạy, tiếp tục thêm vào phần tử có giá trị ngẫu nhiên vào a và tăng sem1 lên 1 (bởi hàm sem_post) cho đến khi sem2=0 (được giảm bởi hàm sem_wait). Khi đó ProcessB được chạy, lấy một phần tử ngẫu nhiên trong a ra và tăng sem2 và sem1 (bởi hàm sem_post) cho đến khi sem1 = 0 (được giảm bởi hàm sem_wait). Hai tiến trình này sẽ được lặp đi lặp lại trong 1 vòng lặp vô hạn.

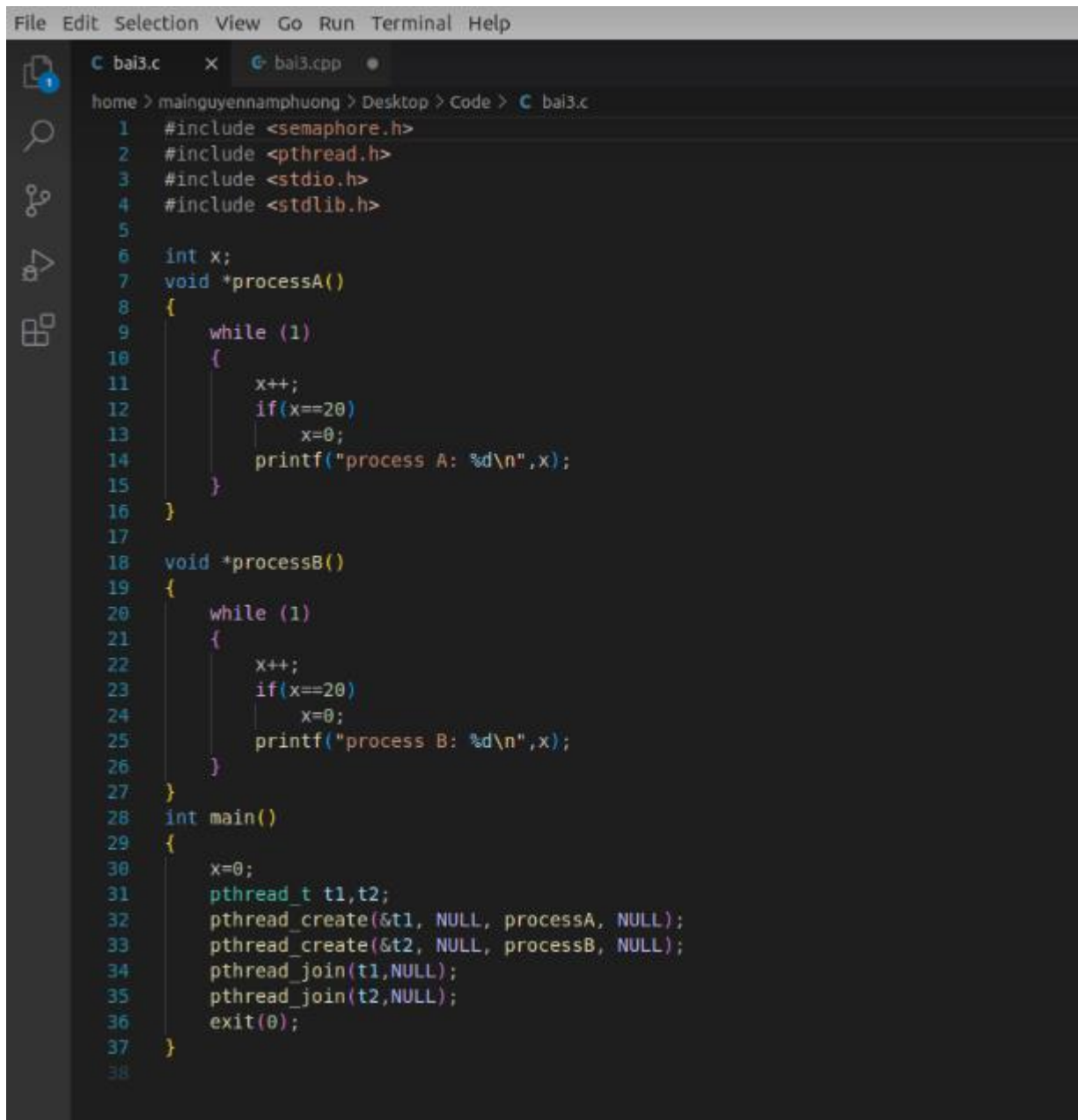
3. Cho 2 process A và B chạy song song như sau:

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

<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.

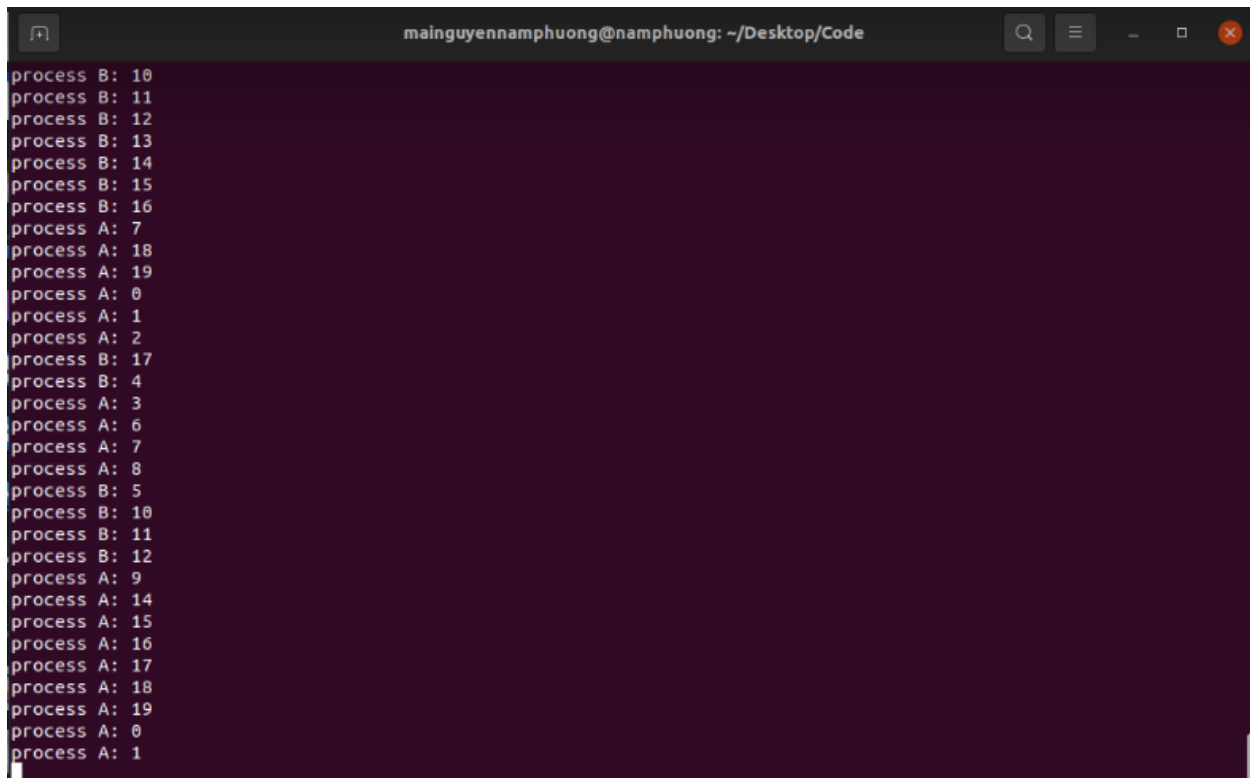
*** Source code:**



The image shows a screenshot of a code editor with a dark theme. The editor has a menu bar at the top with 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', and 'Help'. Below the menu bar, there are two tabs: 'C bai3.c' and 'C bai3.cpp'. The active tab is 'C bai3.c'. The file path is shown as 'home > mainguyennamphuong > Desktop > Code > C bai3.c'. The code is written in C and uses pthreads for process synchronization. It includes headers for semaphore, pthread, stdio, and stdlib. It defines a global variable 'x' and two functions, 'processA()' and 'processB()', both of which increment 'x' until it reaches 20, at which point they reset 'x' to 0 and print the current value. The 'main()' function creates two threads, 't1' and 't2', each running one of the process functions, and then joins them before exiting.

```
1  #include <semaphore.h>
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int x;
7  void *processA()
8  {
9      while (1)
10     {
11         x++;
12         if(x==20)
13             x=0;
14         printf("process A: %d\n",x);
15     }
16 }
17
18 void *processB()
19 {
20     while (1)
21     {
22         x++;
23         if(x==20)
24             x=0;
25         printf("process B: %d\n",x);
26     }
27 }
28 int main()
29 {
30     x=0;
31     pthread_t t1,t2;
32     pthread_create(&t1, NULL, processA, NULL);
33     pthread_create(&t2, NULL, processB, NULL);
34     pthread_join(t1,NULL);
35     pthread_join(t2,NULL);
36     exit(0);
37 }
38
```

*** Kết quả:**



```
mainguyennamphuong@namphuong: ~/Desktop/Code
process B: 10
process B: 11
process B: 12
process B: 13
process B: 14
process B: 15
process B: 16
process A: 7
process A: 18
process A: 19
process A: 0
process A: 1
process A: 2
process B: 17
process B: 4
process A: 3
process A: 6
process A: 7
process A: 8
process B: 5
process B: 10
process B: 11
process B: 12
process A: 9
process A: 14
process A: 15
process A: 16
process A: 17
process A: 18
process A: 19
process A: 0
process A: 1
```

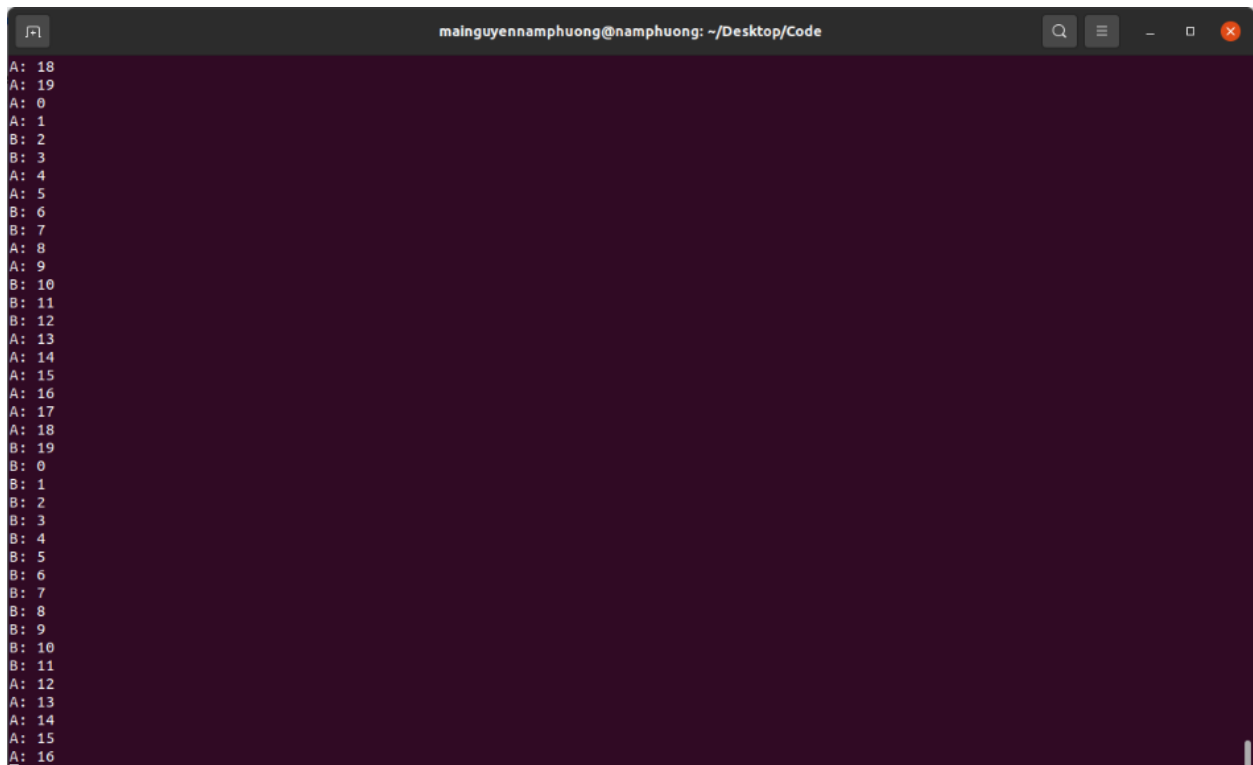
*** Nhân xét:** Process thứ nhất chạy từ 1 đến 19 và khi đến `if(x==20)` thì sẽ quay lại giá trị 0. Và trong khi process 1 đang chuẩn bị đọc dữ liệu từ đĩa, thao tác đọc chưa hoàn tất thì bị process 2 ghi đè dữ liệu mới lên dữ liệu cũ. Nguyên nhân là do khi chạy song song 2 process thứ nhất và thứ hai, biến toàn cục x đều được cho cả hai process biến x gọi ra nhưng không có semaphore để báo rằng đây là hai tiến trình dùng chung biến x dẫn đến việc xảy ra đụng độ khi truy cập và xử lý biến chung như dòng trên. Từ đó dẫn đến kết quả sai với yêu cầu và chương trình chạy bất hợp lý.

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

*** Source code:**

```
home > mainguyennamphuong > Desktop > Code > C bai3.c
1  #include <semaphore.h>
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8  int x;
9  void *processA()
10 {
11     while (1)
12     {
13         pthread_mutex_lock(&mutex);
14         x++;
15         if(x==20)
16             x=0;
17         printf("A: %d\n",x);
18         pthread_mutex_unlock(&mutex);
19     }
20 }
21
22 void *processB()
23 {
24     while (1)
25     {
26         pthread_mutex_lock(&mutex);
27         x++;
28         if(x==20)
29             x=0;
30         printf("B: %d\n",x);
31         pthread_mutex_unlock(&mutex);
32     }
33 }
34 int main()
35 {
36     pthread_mutex_init(&mutex,NULL);
37     x=0;
38     pthread_t t1,t2;
39     pthread_create(&t1, NULL, processA, NULL);
40     pthread_create(&t2, NULL, processB, NULL);
41     pthread_join(t1,NULL);
42     pthread_join(t2,NULL);
43     exit(0);
44 }
45
```

*** Kết quả:**



```
mainguyennamphuong@namphuong: ~/Desktop/Code
A: 18
A: 19
A: 0
A: 1
B: 2
B: 3
A: 4
A: 5
B: 6
B: 7
A: 8
A: 9
B: 10
B: 11
B: 12
A: 13
A: 14
A: 15
A: 16
A: 17
A: 18
B: 19
B: 0
B: 1
B: 2
B: 3
B: 4
B: 5
B: 6
B: 7
B: 8
B: 9
B: 10
B: 11
A: 12
A: 13
A: 14
A: 15
A: 16
```

*** Nhận xét:** Sau khi được đồng bộ với mutex thì ta đã sửa được lỗi trong bài 3, 2 tiến trình đã được báo rằng biến x cục bộ được sử dụng chung nên chương trình đã chạy đúng như yêu cầu

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

* **Code:**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <semaphore.h>
5  #include <unistd.h>
6
7  // Khai báo các semaphore và các biến global
8  sem_t sem_1, sem_2, sem_3, sem_4, sem_5, sem_6, sem_7, sem_8;
9  int x1 = 1;
10 int x2 = 2;
11 int x3 = 3;
12 int x4 = 4;
13 int x5 = 5;
14 int x6 = 6;
15 int w, v, z, y, x;
16 int ans = 0;
17
18 // Hàm thực thi tính toán cho process_1
19 void *process_1(void *arg)
20 {
21     w = x1 * x2;
22     printf("w = %d\n", w);
23     sem_post(&sem_1); // tính hiệu semaphore để cho biết w đã được tính báo cho process 5
24     sem_post(&sem_2); // tính hiệu semaphore để cho biết w đã được tính báo cho process 6
25     sleep(1);
26     return NULL;
27 }
28
29 // Tương tự như process_1
30 void *process_2(void *arg)
31 {
32     v = x3 * x4;
33     printf("v = %d\n", v);
34     sem_post(&sem_3); // tính hiệu semaphore để cho biết v đã được tính, báo cho process 3
35     sem_post(&sem_4); // tính hiệu semaphore để cho biết v đã được tính, báo cho process 4
36     sleep(1);
37     return NULL;
38 }
39
40 // Tương tự như process_1
41 void *process_3(void *arg)
42 {
43     sem_wait(&sem_3); // Chờ semaphore 3 được kích hoạt, tức là v đã được tính
44     y = v * x5;
45     printf("y = %d\n", y);
46     sem_post(&sem_5); // Kích hoạt semaphore để cho biết y đã được tính, báo cho process 5
47     sleep(1);
48     return NULL;
49 }
50
51 // Tương tự như process_1
52 void *process_4(void *arg)
53 {
54     sem_wait(&sem_4); // Chờ semaphore 4 được kích hoạt, tức là v đã được tính
55     z = v * x6;
56     printf("z = %d\n", z);
57     sem_post(&sem_6); // Kích hoạt semaphore để cho biết z đã được tính, báo cho process 6
58     sleep(1);
59     return NULL;
60 }
```

```
61
62 // Tương tự như process_1
63 void *process_5(void *arg)
64 {
65     sem_wait(&sem_1); // Chờ semaphore 1 được kích hoạt, tức là w đã được tính
66     sem_wait(&sem_5); // Chờ semaphore 5 được kích hoạt, tức là y đã được tính
67     y = w * y;
68     printf("y = %d\n", y);
69     sem_post(&sem_7); // Kích hoạt semaphore để cho biết y đã được tính, báo cho process 7
70     sleep(1);
71     return NULL;
72 }
73
74 // Tương tự như process_1
75 void *process_6(void *arg)
76 {
77     sem_wait(&sem_2); // Chờ semaphore 2 được kích hoạt, tức là w đã được tính
78     sem_wait(&sem_6); // Chờ semaphore 6 được kích hoạt, tức là z đã được tính
79     z = w * z;
80     printf("z = %d\n", z);
81     sem_post(&sem_8); // Kích hoạt semaphore để cho biết z đã được tính, báo cho process 7
82     sleep(1);
83     return NULL;
84 }
85
86 // Tương tự như process_1
87 void *process_7(void *arg)
88 {
89     sem_wait(&sem_7); // Chờ semaphore 7 được kích hoạt, tức là y đã được tính
90     sem_wait(&sem_8); // Chờ semaphore 8 được kích hoạt, tức là z đã được tính
91     ans = y + z;
92     printf("ans = %d\n", ans);
93     sleep(1);
94     return NULL;
95 }
96
```

```
96
97  int main()
98  {
99      // Khởi tạo semaphore
100     sem_init(&sem_1, 0, 0);
101     sem_init(&sem_2, 0, 0);
102     sem_init(&sem_3, 0, 0);
103     sem_init(&sem_4, 0, 0);
104     sem_init(&sem_5, 0, 0);
105     sem_init(&sem_6, 0, 0);
106     sem_init(&sem_7, 0, 0);
107     sem_init(&sem_8, 0, 0);
108
109     // Tạo các luồng
110     pthread_t th1, th2, th3, th4, th5, th6, th7;
111     pthread_create(&th1, NULL, process_1, NULL);
112     pthread_create(&th2, NULL, process_2, NULL);
113     pthread_create(&th3, NULL, process_3, NULL);
114     pthread_create(&th4, NULL, process_4, NULL);
115     pthread_create(&th5, NULL, process_5, NULL);
116     pthread_create(&th6, NULL, process_6, NULL);
117     pthread_create(&th7, NULL, process_7, NULL);
118
119     // Đợi các thread hoàn thành
120     pthread_join(th1, NULL);
121     pthread_join(th2, NULL);
122     pthread_join(th3, NULL);
123     pthread_join(th4, NULL);
124     pthread_join(th5, NULL);
125     pthread_join(th6, NULL);
126     pthread_join(th7, NULL);
127
128     return 0;
129 }
```

*** Giải thích:**

- **Khởi tạo Semaphore:** Trong hàm main(), các semaphore được khởi tạo với giá trị ban đầu là 0. Semaphore được sử dụng để đồng bộ hóa các phép tính và đảm bảo rằng chúng được thực hiện theo đúng thứ tự.

- **Tạo các luồng:** Trong hàm main(), bảy luồng (thread) được tạo bằng cách gọi hàm pthread_create. Mỗi luồng sẽ thực hiện một phép tính nhất định. Cụ thể là:

- + process_1 tính toán giá trị của biến w.
- + process_2 tính toán giá trị của biến v.

- + process_3 tính toán giá trị của biến y.
- + process_4 tính toán giá trị của biến z.
- + process_5 tính toán giá trị của biến y một lần nữa, sau khi đã có giá trị của w.
- + process_6 tính toán giá trị của biến z một lần nữa, sau khi đã có giá trị của w.
- + process_7 tính toán giá trị của biến ans, tức là tổng của y và z.

- **Đồng bộ hóa các phép tính:** Các semaphore được sử dụng để đảm bảo rằng các phép tính được thực hiện theo đúng thứ tự:

+ sem_1 và sem_2 cho biết rằng $x1 * x2$ đã được tính toán và ghi vào w, gửi tín hiệu cho process_5 và process_6.

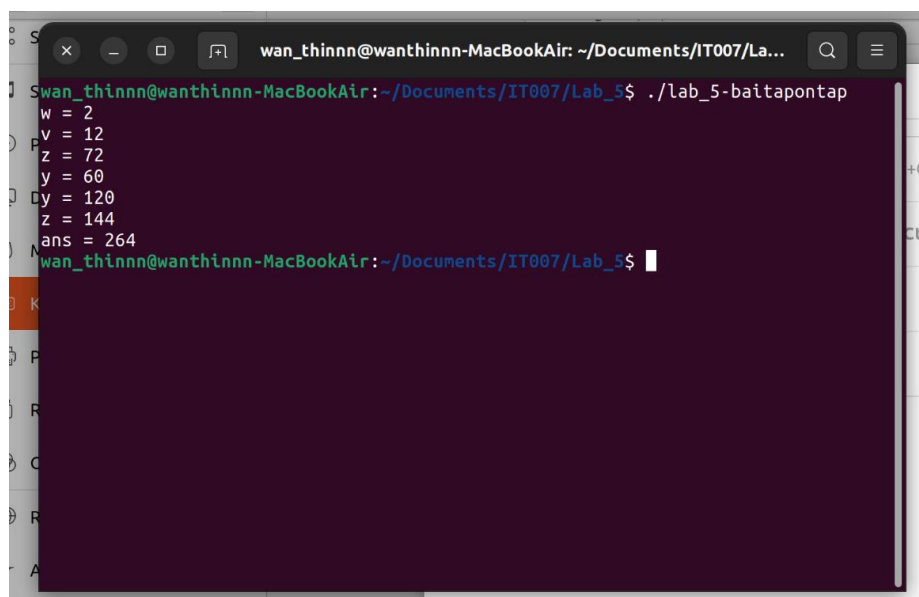
+ sem_3 và sem_4 cho biết rằng $x3 * x4$ đã được tính toán và ghi vào v, gửi tín hiệu cho process_3 và process_4

+ sem_5 cho biết rằng y đã được tính toán, gửi tín hiệu cho process_5.

+ sem_6 cho biết rằng z đã được tính toán, gửi tín hiệu cho process_6.

+ sem_7 và sem_8 cho biết rằng cả y và z đã được tính toán, gửi tín hiệu cho process_7.

- **In kết quả:** Khi tất cả các luồng đã hoàn thành, hàm main() in ra giá trị của biến ans, tức là kết quả của phép tính $y + z$.



```
wan_thinnn@wanthinnn-MacBookAir: ~/Documents/IT007/La...
wan_thinnn@wanthinnn-MacBookAir:~/Documents/IT007/Lab_5$ ./lab_5-baitapontap
w = 2
v = 12
z = 72
y = 60
y = 120
z = 144
ans = 264
wan_thinnn@wanthinnn-MacBookAir:~/Documents/IT007/Lab_5$
```