

Họ và tên: Bùi Lê Nhật Tri

MSSV: 23521634

Lớp: IT00007.P11.1

BÁO CÁO LAB 5

Câu 1: Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: $\text{sells} \leq \text{products} \leq \text{sells} + [4 \text{ số cuối của MSSV}]$

Code:

```

1  #include <pthread.h>
5  #include <semaphore.h>
6
7  using namespace std;
8
9  #define MSSV 1634 // mssv la 23521634
10
11 int products = 0;
12 int sells = 0;
13
14 sem_t sem;
15
16 void* producer(void* arg) {
17     while (true) {
18         sem_wait(&sem);
19         if (products <= sells + MSSV) {
20             products++;
21             printf("Đã sản xuất: %d\n", products);
22         }
23         sem_post(&sem);
24     }
25     return NULL;
26 }
27
28 void* consumer(void* arg) {
29     while (true) {
30         sem_wait(&sem);
31         if (products >= sells) {
32             sells++;
33             printf("Đã tiêu thụ: %d\n", sells);
34         }
35         sem_post(&sem);
36     }
37 }
38
39 int main() {
40     pthread_t prod_thread, con_thread;
41
42     sem_init(&sem, 0, 1);
43
44     pthread_create(&prod_thread, NULL, producer, NULL);
45     pthread_create(&con_thread, NULL, consumer, NULL);
46
47     pthread_join(prod_thread, NULL);
48     pthread_join(con_thread, NULL);
49
50     sem_destroy(&sem);
51
52     return 0;
53 }

```

Hình 1: Source code câu 1

Kết quả:

```
PROBLEMS  DEBUG CONSOLE  TERMINAL  PORTS

Da san xuat: 3913
Da san xuat: 3914
Da san xuat: 3915
Da san xuat: 3916
Da san xuat: 3917
Da san xuat: 3918
Da san xuat: 3919
Da san xuat: 3920
Da san xuat: 3921
Da san xuat: 3922
Da san xuat: 3923
Da san xuat: 3924
Da san xuat: 3925
Da san xuat: 3926
Da san xuat: 3927
Da san xuat: 3928
Da san xuat: 3929
Da san x^C
nhattri@nhattri-VirtualBox:~$
```

Hình 2: Kết quả câu 1

Giải thích:

- **Producer** chỉ sản xuất khi chưa đạt giới hạn (không vượt quá `sells + 1634`).
- **Consumer** chỉ tiêu thụ khi có ít nhất một sản phẩm (`products > sells`).
- **Semaphore** đảm bảo rằng không có luồng nào thay đổi giá trị của `sells` và `products` cùng lúc, tránh race condition và đảm bảo các điều kiện đồng bộ được thỏa mãn.

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.

Code không có semaphore:

Code:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <pthread.h>
5  #include <semaphore.h>
6
7  int arr[10];
8  int counter = 0, limit;
9
10 void *ProcessA(void *arg) {
11     while (1) {
12         if (counter < limit) {
13             arr[counter++] = rand() % 100 + 1;
14             printf("\nSố phần tử trong mảng arr: %d (Thêm)", counter);
15         }
16     }
17     return NULL;
18 }
19
20 void *ProcessB(void *arg) {
21     while (1) {
22         for (int i = 0; i < counter; i++) {
23             arr[i] = arr[i + 1];
24         }
25         counter--;
26
27         if (counter == 0) {
28             printf("\nNothing in array");
29         } else if (counter < limit) {
30             printf("\nSố phần tử trong mảng arr: %d (Xóa)", counter);
31         }
32     }
33     return NULL;
34 }
35
36 int main() {
37     printf("Nhập limit: ");
38     scanf("%d", &limit);
39
40     pthread_t pA, pB;
41     pthread_create(&pA, NULL, ProcessA, NULL);
42     pthread_create(&pB, NULL, ProcessB, NULL);
43
44     pthread_join(pA, NULL);
45     pthread_join(pB, NULL);
46
47     return 0;
48 }

```

Hình 3:Source code câu 2a

Kết quả:

```

nhattri@nhattri-VirtualBox:~$ ./test
Nhập limit: 5

Số phần tử trong mảng arr: 1 (Thêm)
Số phần tử trong mảng arr: 2 (Thêm)
Số phần tử trong mảng arr: 3 (Thêm)
Số phần tử trong mảng arr: 4 (Thêm)
Số phần tử trong mảng arr: 5 (Thêm)
Số phần tử trong mảng arr: 4 (Xóa)
Số phần tử trong mảng arr: 4 (Xóa)
Số phần tử trong mảng arr: 3 (Xóa)
Số phần tử trong mảng arr: 2 (Xóa)
Số phần tử trong mảng arr: 1 (Xóa)
Nothing in array
Số phần tử trong mảng arr: -1 (Xóa)
Số phần tử trong mảng arr: -2 (Xóa)
Số phần tử trong mảng arr: -3 (Xóa)
Số phần tử trong mảng arr: -4 (Xóa)
Số phần tử trong mảng arr: 5 (Thêm)
Số phần tử trong mảng arr: -4 (Thêm)
Số phần tử trong mảng arr: -3 (Thêm)
Số phần tử trong mảng arr: -2 (Thêm)
Số phần tử trong mảng arr: -1 (Thêm)
Số phần tử trong mảng arr: 0 (Thêm)

```

Hình 4: Kết quả câu 2a

Giải thích:

- Tiểu Trình A không có phần tử nào để xử lý nhưng Tiểu Trình B vẫn tiếp tục thay đổi giá trị của counter, điều này có thể do thiếu đồng bộ hóa, khiến luồng B có thể thay đổi giá trị mà không có sự kiểm soát từ luồng A.

Code có semaphore:

Code:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <time.h>
5  #include <pthread.h>
6  #include <semaphore.h>
7  int arr[10];
8  int counter = 0, limit;
9  sem_t sem_add, sem_remove;
10
11 void *ProcessA(void *arg) {
12     while (1) {
13         sem_wait(&sem_add);
14         if (counter < limit) {
15             arr[counter++] = rand() % 100 + 1;
16             printf("\nSố phần tử trong mảng arr sau khi thêm: %d", counter);
17         }
18         sem_post(&sem_remove);
19     }
20     return NULL;
21 }
22
23 void *ProcessB(void *arg) {
24     while (1) {
25         sem_wait(&sem_remove);
26         if (counter > 0) {
27             for (int i = 0; i < counter - 1; i++) {
28                 arr[i] = arr[i + 1];
29             }
30             counter--;
31
32             if (counter == 0) {
33                 printf("\nNothing in array");
34             } else if (counter < limit) {
35                 printf("\nSố phần tử trong mảng arr sau khi xóa: %d", counter);
36             }
37         }
38         sem_post(&sem_add);
39     }
40     return NULL;
41 }

```

Hình 5: Source code câu 2b.1

```

43 int main() {
44     printf("Nhập limit: ");
45     scanf("%d", &limit);
46
47     sem_init(&sem_add, 0, 1);
48     sem_init(&sem_remove, 0, 0);
49
50     pthread_t pA, pB;
51     pthread_create(&pA, NULL, ProcessA, NULL);
52     pthread_create(&pB, NULL, ProcessB, NULL);
53
54     pthread_join(pA, NULL);
55     pthread_join(pB, NULL);
56
57     sem_destroy(&sem_add);
58     sem_destroy(&sem_remove);
59
60     return 0;
61 }

```

Hình 6: Source code câu 2b.2

Kết quả:

```
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử ^C
nhattri@nhattri-VirtualBox:~$
```

Hình 7: Kết quả câu 2b

Giải thích:

- Khi sử dụng 2 semaphore là `sem_add` và `sem_remove` để kiểm soát 2 tiến trình thêm và xóa. Chúng ta có thể thấy số phần tử của mảng a luôn dương.

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

PROCESS A	PROCESS B
-----------	-----------

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

```

G+ test.cpp > ProcessB(void *)
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int x = 0;
6
7  void *ProcessA(void *arg)
8  {
9      while (1)
10     {
11         x = x + 1;
12         if (x == 20)
13             x = 0;
14         printf("A: %d\n", x);
15     }
16     return NULL;
17 }
18
19 void *ProcessB(void *arg)
20 {
21     while (1)
22     {
23         x = x + 1;
24         if (x == 20)
25             x = 0;
26         printf("B: %d\n", x);
27     }
28     return NULL;
29 }
30
31 int main()
32 {
33     pthread_t pA, pB;
34
35     pthread_create(&pA, NULL, ProcessA, NULL);
36     pthread_create(&pB, NULL, ProcessB, NULL);
37
38     pthread_join(pA, NULL);
39     pthread_join(pB, NULL);
40
41     return 0;
42 }

```

Hình 8:Source code câu 3

Kết quả:

```
A: 19
A: 0
A: 1
A: 2
A: 3
A: 4
A: 5
A: 6
A: 7
A: 8
A: 9
A: 10
A: 11
A: 12
A: 13
A: 14
A: 15
A: 16
A: 17
A: 18
^C
○ nhattri@nhattri-VirtualBox:~$
```

Hình 9: Kết quả câu 3

Giải thích:

- **Không chính xác:** Giá trị của x không thể chắc chắn theo đúng thứ tự, vì không có bảo vệ đồng bộ. Chẳng hạn, cả hai luồng có thể cùng đọc giá trị x và tăng nó, dẫn đến việc mất dữ liệu. Điều này tạo ra một hành vi không xác định và khó đoán.
- **In kết quả sai:** Bạn có thể thấy kết quả không đồng nhất, với các giá trị x không liên tục và không như mong đợi. Điều này xảy ra vì các luồng thay đổi x đồng thời mà không có cơ chế đồng bộ.

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

Code:

```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int x = 0;
6  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
7
8  void *ProcessA(void *arg)
9  {
10     while (1)
11     {
12         pthread_mutex_lock(&mutex);
13         x = x + 1;
14         if (x == 20)
15             x = 0;
16         printf("A: %d\n", x);
17         pthread_mutex_unlock(&mutex);
18     }
19     return NULL;
20 }
21
22 void *ProcessB(void *arg)
23 {
24     while (1)
25     {
26         pthread_mutex_lock(&mutex);
27         x = x + 1;
28         if (x == 20)
29             x = 0;
30         printf("B: %d\n", x);
31         pthread_mutex_unlock(&mutex);
32     }
33     return NULL;
34 }
35
36 int main()
37 {
38     pthread_t pA, pB;
39
40     pthread_create(&pA, NULL, ProcessA, NULL);
41     pthread_create(&pB, NULL, ProcessB, NULL);
42
43     pthread_join(pA, NULL);
44     pthread_join(pB, NULL);
45
46     pthread_mutex_destroy(&mutex);
47     return 0;
48 }

```

Hình 10:Source code câu 4

Kết quả:

```
A: 0
B: 1
A: 2
A: 3
A: 4
A: 5
A: 6
A: 7
B: 8
A: 9
B: 10
A: 11
B: 12
A: 13
B: 14
A: 15
B: 16
A: 17
B: 18
A: 19
B: 0
A: 1
B: 2
A: 3
B: 4
A: 5
B: 6
B: 7
A: 8
B: 9
A: 10
B: 11
A: 12
B: 13
^C
○ nhattri@nhattri-VirtualBox:~$
```

Hình 11: Kết quả câu 4

Giải thích:

- **pthread_mutex_lock(&mutex):** Khóa mutex trước khi thay đổi biến x. Điều này đảm bảo rằng chỉ có một luồng có thể thay đổi giá trị của x tại một thời điểm.
- **pthread_mutex_unlock(&mutex):** Giải phóng mutex sau khi thay đổi xong. Sau khi mutex được giải phóng, luồng khác có thể tiếp tục thay đổi giá trị của x.
- **pthread_mutex_destroy(&mutex):** Hủy mutex sau khi sử dụng xong, để giải phóng tài nguyên.