CÂU HỎI VÀ BÀI TẬP CHƯƠNG 5

1. Khi nào thì xảy ra tranh chấp?

- Race condition là tình trạng nhiều process truy xuất và thao tác đồng thời lên dữ liệu chia sẻ. Kết quả cuối cùng của việc truy xuất đồng thời này phụ thuộc thứ tự thực thi của các lệnh thao tác dữ liệu.

- Ví dụ điển hình về bounded buffer:

Producer	**Consumer**

`While (true){`
`  counter++;`
`  //Tang bien counter len 1 don vi`
`}`
	`While (true){`
	`  Counter--;`
	`  //Giam bien counter di 1 don vi`
	`}`

- Cho rằng biến counter là biến được chia sẻ giữa Producer và Consumer (tức là, biến counter có thể được đọc và được ghi bởi cả Producer và Consumer). Và Producer và Consumer chạy đồng thời cùng lúc với nhau (Concurrent)

2. Vấn đề vùng tranh chấp (critical section) là gì?

- Đầu tiên, chúng ta hãy xét một hệ thống có n tiến trình (tạm đặt tên của n tiến trình này là {P0, P1, ..., Pn−1}). Từng tiến trình đều có một đoạn mã, gọi là critical section (CS), tên tiếng Việt là vùng tranh chấp. Trong CS, các đoạn mã thao tác lên dữ liệu chia sẻ giữa các tiến trình.

- Một đặc tính quan trọng mà chúng ta cần quan tâm, đó chính là khi process P0 đang chạy đoạn mã bên trong CS thì không một process nào khác được chạy đoạn mã bên trong CS (để đảm bảo cho dữ liệu được nhất quán). Hay nói cách khác là 1 CS trong một thời điểm nhất định, chỉ có 1 process được phép chạy.

- Và vấn đề vùng tranh chấp (Critical Section Problem) là vấn đề về việc tìm một cách thiết kế một giao thức (một cách thức) nào đó để các process có thể phối hợp với nhau hoàn thành nhiệm vụ của nó.

3. Có những yêu cầu nào dành cho lời giải của bài toán vùng tranh chấp?

Một lời giải cho vấn đề vùng tranh chấp (CS Problem) phải đảm bảo được 3 tính chất sau :

- Loại trừ tương hỗ (Mutual Exclusion): Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.

- Phát triển (Progress): Một tiến trình tạm dừng bên ngoài CS không được ngăn cản các tiến trình khác vào CS.

- Chờ đợi giới hạn (Bounded Waiting): Mỗi process chỉ phải chờ để được vào CS trong một

khoảng thời gian có hạn (finite wait time). Không được xảy ra tình trạng đói tài nguyên (starvation)

4. Có mấy loại giải pháp đồng bộ? Kể tên và trình bày đặc điểm của các loại giải pháp đó?

Có một số loại giải pháp đồng bộ trong lập trình đa luồng hoặc đa tiến trình để quản lý truy cập vào các tài nguyên chia sẻ. Dưới đây là các loại giải pháp phổ biến:

- Locks (Khóa): Locks là một cơ chế để đồng bộ hóa việc truy cập vào các tài nguyên chia sẻ bằng cách "khóa" tài nguyên khi một tiến trình hoặc luồng đang sử dụng nó. Có hai loại chính:

+ Mutex (Mutual Exclusion): Loại lock này chỉ cho phép một tiến trình hoặc luồng truy cập vào tài nguyên chia sẻ tại một thời điểm.

+ Semaphore: Semaphore có thể được sử dụng để đồng bộ hóa một số lượng cụ thể của các tiến trình hoặc luồng có thể truy cập tài nguyên.

- Monitor: Monitor là một cơ chế cung cấp môi trường để triển khai đồng bộ hóa và bảo vệ dữ liệu chia sẻ. Nó bao gồm một tập hợp các biến và thủ tục, và chỉ một tiến trình hoặc luồng có thể thực thi các thủ tục trong monitor tại một thời điểm. Monitor cung cấp một cấu trúc trừu tượng hóa cho việc đồng bộ hóa.

- Điều kiện (Condition Variables): Điều kiện là một cơ chế để đợi và thông báo về các sự kiện xảy ra trong quá trình thực thi đa luồng hoặc đa tiến trình. Các tiến trình hoặc luồng có thể chờ cho một điều kiện xác định được đáp ứng trước khi tiếp tục thực thi.

- Transaction: Transaction là một khái niệm được sử dụng trong cơ sở dữ liệu, mô tả một chuỗi các hoạt động đọc và ghi dữ liệu mà phải được thực hiện nguyên vẹn và atomic (tức là hoàn toàn hoặc không hoàn toàn) để đảm bảo tính nhất quán của dữ liệu.

5. Phân tích và đánh giá ưu, nhược điểm của các giải pháp đồng bộ busy waiting (cả phần cứng và phần mềm)?

Nhóm giải pháp Busy Waiting:

- Tính chất:

+ Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng (thông qua việc kiểm tra điều kiện vào CS liên tục).

+ Không đòi hỏi sự trợ giúp của hệ điều hành.

- Cơ chế chung:

```
While (chưa có quyền) do nothing();
CS;
Từ bỏ quyền sử dụng CS;
```

- Bao gồm một vài loại:

+ Sử dụng các biến cờ hiệu.

+ Sử dụng việc kiểm tra luân phiên.

+ Giải pháp của Peterson.

+ Cấm ngắt (giải pháp phần cứng – hardware).

+ Chỉ thị TSL (giải pháp phần cứng – hardware)

6. Semaphore là gì? Đặc điểm của semaphore? Cách thức hiện thực semaphore? Có mấy loại semaphore? Khi sử dụng semaphore cần lưu ý những vấn đề gì?

- Semaphore là một công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi Busy Waiting. Semaphore là một số nguyên.

- Giả sử ta đang có một Semaphore S. Có 3 thao tác có thể thực thi trên S:

+ Khởi tạo semaphore. Giá trị khởi tạo ban đầu của Semaphore chính là số lượng process được thực hiện CS trong cùng 1 thời điểm.

+ Wait(S) hay còn gọi là P(S): Giảm giá trị semaphore đi một đơn vị (S = S – 1). Nếu giá trị S âm,

process thực hiện lệnh wait() này sẽ bị blocked cho đến khi được đánh thức.

+ Signal(S) hay còn gọi là V(S) : Tăng giá trị semaphore (S = S + 1). Kế đó nếu giá trị S <= 0 (S<=0 tức là vẫn còn process đang bị blocked), lấy một process Q nào đó đang bị blocked rồi gọi wakeup(Q) để đánh thức process Q đó.

- Semaphore trong lập trình có thể được chia thành hai loại chính:

+ Binary Semaphore (Semaphore nhị phân): Loại semaphore này chỉ có hai giá trị có thể nhận được, thường là 0 hoặc 1. Nó được sử dụng để đồng bộ hóa giữa các tiến trình hoặc luồng trong trường hợp chỉ có một tài nguyên duy nhất có thể truy cập được vào một thời điểm nhất định. Binary Semaphore thường được sử dụng để giải quyết vấn đề critical section trong đồng bộ hóa tiến trình.

+ Counting Semaphore (Semaphore đếm): Loại semaphore này có thể nhận các giá trị nguyên dương lớn hơn không. Nó được sử dụng để quản lý một tài nguyên mà có thể có nhiều hơn một phiên bản được truy cập cùng một lúc. Counting Semaphore cho phép quy định số lượng tiến trình hoặc luồng có thể truy cập vào một tài nguyên cụ thể cùng một lúc

- Khi sử dụng semaphore trong lập trình đồng thời, cần lưu ý những vấn đề sau:

+ Deadlock: Semaphore có thể gây ra deadlock nếu không được sử dụng đúng cách. Điều này xảy ra khi các tiến trình hoặc luồng bị kẹt do chúng cố gắng có được tài nguyên mà đã bị khóa bởi tiến trình hoặc luồng khác mà không thể hoàn thành để giải phóng tài nguyên của mình.

+ Starvation: Các tiến trình có thể bị starvation (đói) nếu một hoặc một số trong số chúng không bao giờ được phép truy cập vào tài nguyên được bảo vệ bởi semaphore vì ưu tiên cao hơn của các tiến trình khác.

7. Monitor và Critical Region là gì?

- Monitor là một công cụ đồng bộ, có chức năng tương tự như semaphore nhưng dễ điều khiển hơn. Monitor, là một kiểu dữ liệu trừu tượng, lưu lại các biến chia sẻ và các phương thức dùng để thao tác lên các biến chia sẻ đó. Các biến chia sẻ trong monitor chỉ có thể được truy cập bởi các phương thức được định nghĩa trong monitor. Một monitor chỉ có 1 process hoạt động, tại một thời điểm bất kỳ đang xét. Bằng cách này, monitor sẽ đảm bảo mutual exclusion và dữ liệu chia sẻ của bạn sẽ được an toàn.

- Critical Region (Vùng Criticial): Đây là một phần của chương trình mà khi một tiến trình hoặc luồng đang thực thi trong vùng này, không ai khác được phép thực hiện các hoạt động đồng thời truy cập hoặc sửa đổi các tài nguyên chia sẻ. Việc sử dụng vùng critical là để đảm bảo rằng chỉ có một tiến trình hoặc luồng có thể thực thi trong vùng này tại một thời điểm, ngăn chặn xung đột dữ liệu và bảo vệ tính nhất quán của dữ liệu chia sẻ.

8. Đặc điểm và yêu cầu đồng bộ của các bài toán đồng bộ kinh điển?

Các bài toán đồng bộ kinh điển thường có những đặc điểm và yêu cầu đồng bộ sau:

1. Bài toán Producer - Consumer: Đặc điểm của bài toán này là có hai tiến trình, một tiến trình sản xuất (Producer) và một tiến trình tiêu thụ (Consumer), cùng thao tác trên một bộ đệm (buffer). Yêu cầu đồng bộ của bài toán này bao gồm:

- Producer không được ghi dữ liệu vào buffer đã đầy.

- Consumer không được đọc dữ liệu từ buffer đang trống.

- Buffer không được có kích thước không giới hạn.

2. Bài toán Readers - Writers: Trong bài toán này, có hai loại tiến trình là Readers (người đọc) và Writers (người viết) cùng thao tác trên một tài nguyên chung. Yêu cầu đồng bộ thường là đảm bảokhông có hiện tượng "đói" (starvation) và "đua" (race condition).

3. Bài toán các triết gia ăn tối (Dining Philosophers): Đây là một bài toán đồng bộ hoá kinh điển, trình bày yêu cầu cấp phát nhiều tài nguyên giữa các quá trình trong cách tránh việc khoá chết và đói tài nguyên

Mỗi bài toán đều có những yêu cầu đồng bộ riêng và cần được giải quyết bằng các phương pháp đồng bộ phù hợp.

9. (Bài tập mẫu) Xét giải pháp phần mềm do Dekker đề nghi để tổ chức truy xuất độc quyền cho 2 tiến trình. Hai tiến trình P0 và P1 chia sẻ các biến sau:

```
boolean flag[2]; /* initially false */
int turn;
```

Cấu trúc một tiến trình Pi (với i = 0 hay 1 và j là tiến trình còn lại) như sau:

```
while (true) {
      flag[i] = true;

      while (flag[j]) {
            if (turn == j) {
                  flag[i] = false;
                  while (turn == j)
                        ; /* do nothing */
                  flag[i] = true;
            }
      }
            /* critical section */
      turn = j;
      flag[i] = false;
            /* remainder section */
}
```

Giải pháp này có thỏa 3 yêu cầu trong viêc giải quyết tranh chấp không?

Trả lời:

Giải pháp này thỏa 3 yêu cầu trong giải quyết tranh chấp vì:

- Loại trừ tương hỗ: Tiến trình Pi chỉ có thể vào vùng tranh chấp khi flag[j] = false. Giả sử P0 đang ở trong vùng tranh chấp, tức là flag[0] = true và flag[1] = false. Khi đó P1 không thể vào vùng tranh chấp (do bị chặn bởi lệnh while (flag[j])). Tương tự cho tình huống P1 vào vùng tranh chấp trước.

- Progress: Giá trị của biến turn chỉ có thể thay đổi ở cuối vùng tranh chấp. Giả sử chỉ có 1 tiến trình Pi muốn vào vùng tranh chấp. Lúc này, flag[j] = false và tiến trình Pi sẽ được vào vùng tranh chấp ngay lập tức. Xét trường hợp cả 2 tiến trình đều muốn vào vùng tranh chấp và giá trị của turn đang là 0. Cả flag[0] và flag[1] đều bằng true. Khi đó, P0 sẽ được vào vùng tranh chấp, bởi tiến trình P1 sẽ thay đổi flag[1] = false (lệnh kiểm tra điều kiện if (turn == j) chỉ đúng với P1). Tương tự cho trường hợp turn = 1.

- Chờ đợi giới hạn: Pi chờ đợi lâu nhất là sau 1 lần Pj vào vùng tranh chấp (flag[j] = false sau khi Pj ra khỏi vùng tranh chấp). Tương tự cho trường hợp Pj chờ Pi.

10. Xét giải pháp đồng bộ hóa sau:

```
while (TRUE) {
      int j = 1-i;
      flag[i]= TRUE;
      turn = i;
      while (turn == j && flag[j]==TRUE);
      critical-section ();
      flag[i] = FALSE;
      Noncritical-section ();
}
```

Giải pháp này có thỏa yêu cầu độc quyền truy xuất không?

- Ta thấy code này có đoạn j = 1 – i => đề chỉ đang nói tới xét 2 tiê trình P1 và P0 vì khi i = 0 thì j = 1 và ngược lại.

- Giải thuật này không thoả mãn: Xét tình huống khi flag[0] =1; turn =0; lúc này P0 vào CS, Nếu lúc đó flag[1] = 1, P1 có thể gán turn = 1 và vào luôn CS (2 tiến trình cùng vào CS một lúc).

11. Giả sử một máy tính không có chỉ thị TSL, nhưng có chỉ thị Swap có khả năng hoán đổi nội dung của hai từ nhớ chỉ bằng một thao tác không thể phân chia:

```
procedure Swap(var a,b: boolean){
      var temp: boolean;
      begin
            temp:= a;
            a:= b;
            b:= temp;
      end;
}
```

Sử dụng chỉ thị này có thể tổ chức truy xuất độc quyền không? Nếu có, xây dựng cấu trúc chương trình tương ứng.

- Chỉ thị Swap như đã được mô tả cho phép hoán đổi nội dung của hai biến nhớ một cách không thể phân chia. Dựa trên đây, chúng ta có thể sử dụng chỉ thị Swap này để xây dựng một giải thuật đồng bộ hóa đơn giản nhưng hiệu quả như Peterson's Algorithm để đạt được truy cập độc quyền vào một vùng critical section giữa hai tiến trình.

- Dưới đây là triển khai của Peterson's Algorithm sử dụng chỉ thị Swap trong ngôn ngữ lập trình C:

```
#include <stdio.h>
#include <stdbool.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

bool interest[2]; // Mảng đánh dấu sự quan tâm của mỗi tiến trình
int next_turn;    // Biến theo dõi lượt truy cập

void swap_bool(bool *a, bool *b)
{
    bool temp = *a;
    *a = *b;
    *b = temp;
}

void critical_section(int id)
{
    interest[id] = true; // Đánh dấu sự quan tâm của tiến trình id
    next_turn = 1 - id;  // Xác định lượt truy cập của tiến trình id

    while (interest[1 - id] && next_turn == 1 - id)
    {
        // Chờ đợi
        swap_bool(&interest[id], &next_turn); // Swap interest[id] và next_turn nếu cần
    }

    // Critical Section
    printf("Tiến trình %d đang ở vùng độc quyền (Critical Section)\n", id);

    // Kết thúc vùng độc quyền
    interest[id] = false;
}

int main()
{
    interest[0] = false;
    interest[1] = false;
```

```
    // Tiến trình 0 và tiến trình 1 cùng chạy song song
    // Để minh họa, ta gọi hai tiến trình bằng fork()
    if (fork() == 0)
    {
        critical_section(0); // Tiến trình con 1
    }
    else
    {
        critical_section(1); // Tiến trình cha (tiến trình con 2)
    }

    return 0;
}
```

=> đây là một triển khai của Peterson's Algorithm trong ngôn ngữ lập trình C, sử dụng chỉ thị Swap. Các biến flag được sử dụng để đánh dấu sự quan tâm của mỗi tiến trình và biến turn để theo dõi lượt truy cập. Mỗi tiến trình cố gắng vào vùng độc quyền (Critical Section) sẽ kiểm tra điều kiện để xem liệu tiến trình kia có đang quan tâm và có đến lượt truy cập không. Nếu cả hai điều kiện đều đúng, tiến trình sẽ chờđợi bằng cách swap flag[i] và turn nếu cần, để đảm bảo tính đồng bộ.

12. Xét hai tiến trình sau:

process A {while (TRUE) na = na +1;}

process B {while (TRUE) nb = nb +1;}

a. Đồng bộ hóa xử lý của 2 tiến trình trên, sử dụng 2 semaphore tổng quát, sao cho tại bất kỳ thời điểm nào cũng có nb <= na <= nb +10.

Chúng ta phải làm 2 việc với 2 semaphore đã cho: đảm bảo na >= nb và na <= nb + 10, như vậy nghĩa là khi na = nb thì tiến trình B bị block cho tới khi A tiến hành được ít nhất một lần, cũng như khi na = nb + 10 thì A bị block cho tới khi B tiến hành được ít nhất một lần.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t semaphoreA, semaphoreB;
int na = 0, nb = 0;
void *processA(void *arg)
{
```

```
    while (1)
    {
        sem_wait(&semaphoreA); // Đợi semaphore A
        na = na + 1;
        if (na <= nb || na > nb + 10)
        {
            na = nb + 1; // Đồng bộ hóa giá trị của na
        }
        sem_post(&semaphoreB); // Báo hiệu semaphore B
    }
    return NULL;
}
void *processB(void *arg)
{
    while (1)
    {
        sem_wait(&semaphoreB); // Đợi semaphore B
        nb = nb + 1;
        sem_post(&semaphoreA); // Báo hiệu semaphore A
    }
    return NULL;
}
int main()
{
    sem_init(&semaphoreA, 0, 1);      // Khởi tạo semaphore A với giá trị ban đầu
    là 1 sem_init(&semaphoreB, 0, 0);
    pthread_create(&threadA, NULL, processA, NULL);
    pthread_create(&threadB, NULL, processB, NULL);
    pthread_join(threadA, NULL);
    pthread_join(threadB, NULL);
    sem_destroy(&semaphoreA);
    sem_destroy(&semaphoreB);
    return 0;
}
```

b. Nếu giảm điều kiện chỉ còn là na <= nb +10, cần sửa chữa giải pháp trên như thế nào?

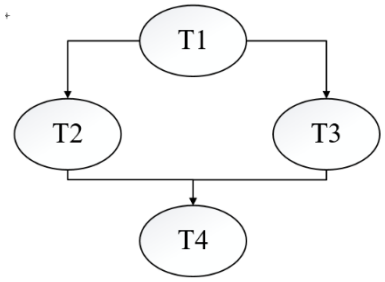
```
Void processA ()
{
```

```
            While (1)
            {
                  wait(Semaphore_2);
                  na = na + 1;
            }
      }
      Void processB ()
      {
            While (1)
            {
                  nb = nb + 1;
                  signal(semaphore_2);
            }
```

c. Giải pháp trên còn đúng nếu có nhiều tiến trình loại A và B cùng thực hiện?

Đúng, vì có thể có nhiều tiến trình loại A hoặc loại B cùng thực hiện nhưng chỉ có 2 biến Semaphore toàn cục mà chúng sẽ thao tác (đối với câu b là 1 Semaphore).

13. (Bài tập mẫu) Xét một hệ thống có 4 tiểu trình T1, T2, T3, T4. Quan hệ giữa các tiểu trình này được biểu diễn như sơ đồ bên dưới, với mũi tên từ tiểu trình (Tx) sang tiểu trình (Ty) có nghĩa là tiểu trình Tx phải kết thúc quá trình hoạt động của nó trước khi tiểu trình Ty bắt đầu thực thi. Giả sử tất cả các tiểu trình đã được khởi tạo và sẵn sàng để thực thi. Hãy sử dụng semaphore để đồng bộ hoạt động của các tiểu trình sao cho đúng với sơ đồ đã cho.



Trả lời:

Khai báo và khởi tạo các semaphore:

init(sem1,0); //khởi tạo semaphore sem1 có giá trị bằng 0

init(sem2,0); //khởi tạo semaphore sem2 có giá trị bằng 0

void T1(void)
{
 	
//T1 thực thi
 	
signal(sem1)
signal(sem1)
}
	void T2(void)
	{
	
	wait(sem1)
	
	//T2 thực thi
	
	signal(sem2)
	
	}
		void T3(void)
		{
		
		wait(sem1)
		
		//T3 thực thi
		
		signal(sem2)
		
		}
			void T4(void)
			{
			
			wait(sem2)
			wait(sem2)
			//T4 thực thi
			
			}

14. Một biến X được chia sẻ bởi 2 tiến trình cùng thực hiện đoạn code sau:

```
do
      X = X + 1;
      if (X == 20) X = 0;
while (TRUE);
```

Bắt đầu với giá trị X = 0, chứng tỏ rằng giá tri X có thể vượt quá 20. Cần sửa chữa đoạn chương trình trên như thế nào để đảm bảo X không vượt quá 20?

- Do X được chia sẻ chung ở cả hai tiến trình và chỉ bị reset khi X == 20 nên X có thể vượt quá 20 khi:

- Có một lý do trong máy đột ngột khiến tiến trình 2 dừng lại. Sau đó tại tiến trình 1, khi X = 19 thì tiến trình 2 được release đúng ngay đoạn X = X + 1, và cộng dồn với X = 20 sau lệnh X = X + 1 của tiến trình 1 dẫn tới X vượt quá 20 và còn bị cộng tới vô cùng.

- Dưới đây là code demo với cách giả định P2 bị crash và release bằng một lệnh sleep nhỏ, kết quả của X sẽ vượt quá 20 trong tích tắc

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX_VALUE 20
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int X = 0;
void *process(void *arg)
```

```
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        if (X < MAX_VALUE)
        {
            X = X + 1;
            if (X == MAX_VALUE)
            {
                X = 0;
            }
        }
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
int main()
{
    pthread_t thread;
    pthread_create(&thread, NULL, process, NULL);
    // Chờ một khoảng thời gian để kiểm tra giá trị của X
    sleep(5);
    pthread_cancel(thread); // Kết thúc tiến trình sau một khoảng thời gian
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

15. Xét 2 tiến trình xử lý đoạn chương trình sau:

process P1 {A1 ; A2 }

process P2 {B1 ; B2 }

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
pthread_barrier_t barrier;
```

```
void *processP1(void *arg)
{
    // Hoạt động A1
    printf("A1 hoàn tất\n");
    pthread_barrier_wait(&barrier); // Đợi tất cả các tiến trình đạt barrier
    // Hoạt động A2
    printf("A2 bắt đầu\n");

    return NULL;
}
void *processP2(void *arg)
{
    // Hoạt động B1
    printf("B1 hoàn tất\n");
    pthread_barrier_wait(&barrier); // Đợi tất cả các tiến trình đạt barrier
    // Hoạt động B2
    printf("B2 bắt đầu\n");

    return NULL;
}
int main()
{
    pthread_t threadP1, threadP2;
    pthread_barrier_init(&barrier, NULL, 2); // Khởi tạo Barrier với số lượng
    tiến trình là 2 pthread_create(&threadP1, NULL, processP1, NULL);
    pthread_create(&threadP2, NULL, processP2, NULL);
    pthread_join(threadP1, NULL);
    pthread_join(threadP2, NULL);
    pthread_barrier_destroy(&barrier);
    return 0;
}
```

Trong bài này, chúng ta sử dụng pthread_barrier_t barrier để tạo ra một barrier với số lượng tiến trình cần chờ là 2 (trong trường hợp này là P1 và P2). Tiến trình P1 và P2 đều thực hiện các hoạt động của mình (A1 và B1), sau đó gọi pthread_barrier_wait(&barrier) để chờ tất cả các tiến trình khác đạt đến điểm barrier. Khi cả P1 và P2 đều đạt tới barrier, các hoạt động A2 và B2 sẽ bắt đầu. Barrier đảm bảo rằng cả A1 và B1 đều hoàn tất trước khi A2 và B2 bắt đầu, và nó đồng bộ hóa tiến trình để chờ đến khi tất cả các tiến trình đã đạt đến điểm barrier trước khi tiếp tục

16. Tổng quát hóa bài tập 14 cho các tiến trình có đoạn chương trình sau:

process P1 { for (i = 1; i <= 100; i ++) A_i }

process P2 { for (j = 1; j <= 100; j ++) B_j }

Đồng bộ hóa hoạt động của 2 tiến trình này sao cho với k bất kỳ (2<=k<=100), A_k chỉ có thể bắt đầu khi $B_{(k-1)}$ đã kết thúc và B_k chỉ có thể bắt đầu khi $A_{(k-1)}$ đã kết thúc.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX 100
sem_t semaphoreA, semaphoreB;
void *processP1(void *arg)
{
    int i;
    for (i = 1; i <= MAX; i++)
    {
        // Chờ B(k-1) đã kết thúc
        sem_wait(&semaphoreB);
        // Hoạt động Ai
        printf("A%d hoàn tất\n", i);
        // Báo hiệu Bk có thể bắt đầu
        sem_post(&semaphoreA);
    }
    return NULL;
}
void *processP2(void *arg)
{
    int j;
    for (j = 1; j <= MAX; j++)
    {
        // Chờ A(k-1) đã kết thúc
        sem_wait(&semaphoreA);
        // Hoạt động Bj
        printf("B%d hoàn tất\n", j);
        // Báo hiệu Ak có thể bắt đầu
        sem_post(&semaphoreB);
    }
    return NULL;
}
int main()
```

```
{
    sem_init(&semaphoreA, 0, 0);      // Khởi tạo Semaphore A với giá trị ban đầu
    là 0 sem_init(&semaphoreB, 0, 0); // Khởi tạo Semaphore B với giá trị ban đầu
    là 0 pthread_t threadP1, threadP2;
    pthread_create(&threadP1, NULL, processP1, NULL);
    pthread_create(&threadP2, NULL, processP2, NULL);
    pthread_join(threadP1, NULL);
    pthread_join(threadP2, NULL);
    sem_destroy(&semaphoreA);
    sem_destroy(&semaphoreB);
    return 0;
}
```

17. Sử dụng semaphore để viết lại chương trình sau theo mô hình xử lý đồng hành:

w := x1 * x2

v := x3 * x4

y := v * x5

z := v * x6

x := w * y

z := w * z

ans := y + z

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

// Khai báo các semaphore và các biến global
sem_t sem_A, sem_B, sem_C, sem_D, sem_E, sem_F, sem_G, sem_H;
int a1 = 1;
int a2 = 2;
int a3 = 3;
int a4 = 4;
int a5 = 5;
int a6 = 6;
int u, v, t, s, r;
```

```
int result = 0;

// Hàm thực thi tính toán cho task_A
void *task_A(void *arg)
{
    u = a1 * a2;
    printf("u = %d\n", u);
    sem_post(&sem_A); // tính hiệu semaphore để cho biết u đã được tính báo cho task_E
    sem_post(&sem_B); // tính hiệu semaphore để cho biết u đã được tính báo cho task_F
    sleep(1);
    return NULL;
}

// Tương tự như task_A
void *task_B(void *arg)
{
    v = a3 * a4;
    printf("v = %d\n", v);
    sem_post(&sem_C); // tính hiệu semaphore để cho biết v đã được tính, báo cho task_C
    sem_post(&sem_D); // tính hiệu semaphore để cho biết v đã được tính, báo cho task_D
    sleep(1);
    return NULL;
}

// Tương tự như task_A
void *task_C(void *arg)
{
    sem_wait(&sem_C); // Chờ semaphore C được kích hoạt, tức là v đã được tính
    t = v * a5;
    printf("t = %d\n", t);
    sem_post(&sem_E); // Kích hoạt semaphore để cho biết t đã được tính, báo cho task_E
    sleep(1);
    return NULL;
}

// Tương tự như task_A
void *task_D(void *arg)
{
    sem_wait(&sem_D); // Chờ semaphore D được kích hoạt, tức là v đã được tính
```

```
    s = v * a6;
    printf("s = %d\n", s);
    sem_post(&sem_F); // Kích hoạt semaphore để cho biết s đã được tính, báo cho task_F
    sleep(1);
    return NULL;
}

// Tương tự như task_A
void *task_E(void *arg)
{
    sem_wait(&sem_A); // Chờ semaphore A được kích hoạt, tức là u đã được tính
    sem_wait(&sem_E); // Chờ semaphore E được kích hoạt, tức là t đã được tính
    r = u * t;
    printf("r = %d\n", r);
    sleep(1);
    return NULL;
}

// Tương tự như task_A
void *task_F(void *arg)
{
    sem_wait(&sem_B); // Chờ semaphore B được kích hoạt, tức là u đã được tính
    sem_wait(&sem_F); // Chờ semaphore F được kích hoạt, tức là s đã được tính
    s = u * s;
    printf("s = %d\n", s);
    sem_post(&sem_H); // Kích hoạt semaphore để cho biết s đã được tính, báo cho task_G
    sleep(1);
    return NULL;
}

// Tương tự như task_A
void *task_G(void *arg)
{
    sem_wait(&sem_E); // Chờ semaphore G được kích hoạt, tức là t đã được tính
    sem_wait(&sem_H); // Chờ semaphore H được kích hoạt, tức là s đã được tính
    result = t + s;
    printf("result = %d\n", result);
    sleep(1);
    return NULL;
```

```
}

int main()
{
    // Khởi tạo semaphore
    sem_init(&sem_A, 0, 0);
    sem_init(&sem_B, 0, 0);
    sem_init(&sem_C, 0, 0);
    sem_init(&sem_D, 0, 0);
    sem_init(&sem_E, 0, 0);
    sem_init(&sem_F, 0, 0);
    sem_init(&sem_G, 0, 0);
    sem_init(&sem_H, 0, 0);

    // Tạo các luồng
    pthread_t thA, thB, thC, thD, thE, thF, thG;
    pthread_create(&thA, NULL, task_A, NULL);
    pthread_create(&thB, NULL, task_B, NULL);
    pthread_create(&thC, NULL, task_C, NULL);
    pthread_create(&thD, NULL, task_D, NULL);
    pthread_create(&thE, NULL, task_E, NULL);
    pthread_create(&thF, NULL, task_F, NULL);
    pthread_create(&thG, NULL, task_G, NULL);

    // Đợi các thread hoàn thành
    pthread_join(thA, NULL);
    pthread_join(thB, NULL);
    pthread_join(thC, NULL);
    pthread_join(thD, NULL);
    pthread_join(thE, NULL);
    pthread_join(thF, NULL);
    pthread_join(thG, NULL);

    return 0;
}
```