LAB 2.2: QUẢN LÝ TIẾN TRÌNH (PHẦN 2)

(2312678 – Nguyễn Lê Bảo Long – CTK47A)

# Mục tiêu

Hiểu về các công cụ đồng bộ hóa

# Nội dung

## Bài tập cơ bản

1. Trình bày các điều kiện của một giải pháp đồng bộ?

Các điều kiện cơ bản của một giải pháp đồng bộ là:

* **Mutual Exclusion (Độc quyền):** Chỉ một tiến trình có thể vào critical section tại một thời điểm.
* **Progress (Tiến trình):** Một tiến trình phải được phép vào critical section nếu không có tiến trình nào đang ở trong đó.
* **Bounded Waiting (Chờ có giới hạn):** Mỗi tiến trình có một thời gian chờ đợi hợp lý trước khi có thể vào critical section.

1. Cho giải pháp đồng bộ:

## Biến chia sẻ

* + int turn; /\* khởi đầu turn = 0 \*/
  + nếu turn = i thì Pi được phép vào critical section, với i = 0 hay 1

## Process Pi

do {

while (turn != i);

## /\* critical section \*/

turn = j;

## /\* remainder section \*/

} while (1);

Giải pháp này có thỏa mãn các yêu cầu của một giải pháp đồng bộ? Giải thích?

* ***Mutual Exclusion (Tính độc quyền):*** Giải pháp này đảm bảo tính loại trừ lẫn nhau. Chỉ có một tiến trình (Pi) được phép vào critical section mỗi lần, vì nếu một tiến trình đã vào critical section, tiến trình còn lại phải đợi đến khi turn thay đổi và tiến trình đó rời khỏi critical section.
* ***Progress (Tiến triển):*** Tính tiến triển có thể bị vi phạm trong một số trường hợp. Nếu cả hai tiến trình đều liên tục kiểm tra turn != i mà không có tiến trình nào thay đổi turn, thì không có tiến trình nào có thể tiến vào critical section, dẫn đến tình trạng tắc nghẽn (deadlock). Tuy nhiên, nếu một tiến trình thực hiện đúng theo logic của giải pháp (thay đổi giá trị của turn sau khi ra khỏi critical section), tiến trình còn lại sẽ vào được critical section.

#### **Bounded Waiting (Chờ có giới hạn):**Giải pháp này không hoàn toàn đảm bảo tính chất chờ có giới hạn. Nếu một tiến trình liên tục vào critical section, tiến trình còn lại có thể bị đợi lâu. Điều này có thể dẫn đến một tiến trình bị loại trừ vĩnh viễn nếu không có cơ chế đảm bảo việc thay đổi lượt (turn) theo cách công bằng.

### **Kết luận:** Giải pháp này không thỏa mãn hoàn toàn các yêu cầu của một giải pháp đồng bộ. Nó đảm bảo tính loại trừ lẫn nhau nhưng có thể vi phạm tính chất tiến triển, chờ có giới hạn và công bằng, đặc biệt khi có tiến trình liên tục chiếm critical section mà không cho tiến trình còn lại có cơ hội vào.

1. Cho giải pháp đồng bộ:

## Biến chia sẻ

* + boolean flag[2]; /\* khởi đầu flag[0] = flag[1] = false \*/
  + Nếu flag[i] = true thì Pi “sẵn sàng” vào critical section.

## Process Pi

do {

flag[ i ] = true; /\* Pi “sẵn sàng” vào CS \*/ while ( flag[ j ] ); /\* Pi “nhường” Pj \*/

## /\* critical section \*/

flag[ i ] = false;

## /\* remainder section \*/

} while(1);

Giải pháp này có thỏa mãn các yêu cầu của một giải pháp đồng bộ? Giải thích?

* ***Mutual Exclusion (Tính độc quyền):*** Giải pháp này đảm bảo tính loại trừ lẫn nhau. Điều này xảy ra vì sau khi một tiến trình Pi đặt flag[i] = true, tiến trình kia (Pj) phải kiểm tra xem flag[j] có phải là false không trước khi vào critical section. Nếu flag[j] vẫn là true, tiến trình Pi sẽ chờ. Vì vậy, chỉ có một tiến trình có thể vào critical section tại một thời điểm.
* ***Progress (Tiến triển):*** Tiến triển được đảm bảo nếu không có tiến trình nào ở trong critical section. Sau khi một tiến trình đã ra khỏi critical section, tiến trình khác có thể vào ngay lập tức nếu nó muốn, vì nó không bị chặn bởi điều kiện nào khác ngoài việc tiến trình còn lại đang ở trong critical section.
* ***Bounded Waiting (Chờ có giới hạn):*** Tính chất này được thỏa mãn trong giải pháp này. Sau khi một tiến trình ra khỏi critical section và thay đổi flag[i] thành false, tiến trình còn lại có thể vào critical section sau một số lần chờ nhất định. Không có tiến trình nào bị chờ vô hạn, vì tiến trình Pi luôn có cơ hội vào critical section sau một số vòng lặp giới hạn.

Kết luận:

Giải pháp đồng bộ này thỏa mãn tất cả các yêu cầu của một giải pháp đồng bộ:

* Mutual Exclusion: Đảm bảo chỉ một tiến trình vào critical section tại một thời điểm.
* Progress: Nếu không có tiến trình nào trong critical section, tiến trình tiếp theo có thể vào.
* Bounded Waiting: Không có tiến trình nào bị đợi mãi mãi.

Vì vậy, giải pháp này thỏa mãn tất cả các yêu cầu của một giải pháp đồng bộ.

1. Xét giải pháp phần mềm do Dekker đề nghị để tổ chức truy xuất độc quyền cho n tiến trình . Các biến chia sẻ :

enum pstate {idle, want in, in cs}; pstate flag[n];

int turn; /\* turn=[0,n-1] \*/

## Process Pi

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 mãn các yêu cầu của một giải pháp đồng bộ? Giải thích?

Giải pháp phần mềm do Dekker đề nghị được thiết kế để tổ chức truy xuất độc quyền (mutual exclusion) cho n tiến trình, với các biến chia sẻ như sau:

* flag[i]: Biến chỉ trạng thái của tiến trình i, có thể có ba giá trị: idle, want in, và in cs (trong phần vùng quan trọng).
* turn: Biến chỉ "lượt" tiến trình được phép vào phần vùng quan trọng. Biến này có giá trị từ 0 đến n-1, đại diện cho từng tiến trình.

Trong khi thực hiện thuật toán này, mỗi tiến trình thực hiện các bước sau:

1. Đặt flag[i] thành true, báo hiệu rằng tiến trình i muốn vào phần vùng quan trọng.
2. Sau đó, tiến trình kiểm tra nếu flag[j] của tiến trình khác (j) đã được đặt là true, tức là tiến trình j cũng muốn vào phần vùng quan trọng. Tiến trình i sẽ kiểm tra biến turn:
   * Nếu turn == j, tiến trình i tạm thời không vào phần vùng quan trọng và đặt lại flag[i] thành false, sau đó chờ đợi cho đến khi turn không còn bằng j.
   * Nếu không phải vậy, tiến trình i tiếp tục kiểm tra tiến trình j cho đến khi điều kiện cho phép tiến trình i vào phần vùng quan trọng.
3. Tiến trình i sẽ vào phần vùng quan trọng (critical section) khi không có tiến trình nào khác đang chiếm dụng.
4. Khi ra khỏi phần vùng quan trọng, tiến trình sẽ thay đổi biến turn để cho phép tiến trình khác có thể vào, sau đó đặt lại flag[i] thành false.

### Các yêu cầu của một giải pháp đồng bộ:

* **Mutual Exclusion (Độc quyền)**: Chỉ có một tiến trình duy nhất được phép vào phần vùng quan trọng tại một thời điểm. Giải pháp này thỏa mãn yêu cầu này vì mỗi tiến trình chỉ vào phần vùng quan trọng khi không có tiến trình nào khác muốn vào, và việc thay đổi giá trị turn đảm bảo rằng chỉ một tiến trình được phép vào.
* **Progress (Tiến trình)**: Giải pháp phải đảm bảo rằng nếu không có tiến trình nào đang ở trong phần vùng quan trọng, thì sẽ có tiến trình tiếp theo vào phần vùng quan trọng mà không bị tắc nghẽn. Giải pháp này thỏa mãn yêu cầu này bởi vì tiến trình sẽ không bị chặn vô hạn và sẽ tiếp tục vào phần vùng quan trọng sau khi một tiến trình khác kết thúc và thay đổi giá trị turn.
* **Bounded Waiting (Giới hạn chờ đợi)**: Giải pháp phải đảm bảo rằng mỗi tiến trình sẽ có cơ hội vào phần vùng quan trọng sau một số lần không quá giới hạn. Giải pháp của Dekker có thể không hoàn toàn thỏa mãn yêu cầu này đối với số lượng tiến trình n lớn, vì việc kiểm tra biến turn và flag có thể khiến một tiến trình phải chờ đợi quá lâu nếu các tiến trình khác liên tục thay đổi giá trị turn. Điều này có thể dẫn đến tình trạng tiến trình phải chờ đợi vô hạn trong trường hợp không có tiến trình nào được ưu tiên.

### Kết luận: Giải pháp của Dekker thỏa mãn yêu cầu **mutual exclusion** và **progress**, nhưng không hoàn toàn thỏa mãn yêu cầu **bounded waiting** đối với số lượng tiến trình lớn. Trong trường hợp có nhiều tiến trình, sự thay đổi giá trị turn có thể tạo ra tình trạng chờ đợi quá lâu cho một số tiến trình.

1. Cho giải pháp đồng bộ:

while (TRUE) {

int j = 1-i;

flag[i]= TRUE; turn = i;

while (turn == j && flag[j]==TRUE);

## /\* critical section \*/

flag[i] = FALSE;

## /\* remaindersection \*/

}

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

Giải pháp đồng bộ này là một biến thể của thuật toán Peterson, nhằm tổ chức truy xuất độc quyền (mutual exclusion) cho hai tiến trình P0 và P1, với các biến chia sẻ như sau:

* flag[i]: Biến trạng thái của tiến trình i, có giá trị TRUE khi tiến trình i muốn vào phần vùng quan trọng, và FALSE khi không.
* turn: Biến cho biết tiến trình nào có lượt vào phần vùng quan trọng. Biến này có giá trị là i hoặc j (với i và j là hai tiến trình).

### Giải thích chi tiết thuật toán:

* Mỗi tiến trình i sẽ làm như sau:
  1. Đặt flag[i] = TRUE, báo hiệu rằng tiến trình i muốn vào phần vùng quan trọng.
  2. Đặt turn = i, cho thấy tiến trình i đang có lượt vào phần vùng quan trọng.
  3. Tiến trình i kiểm tra điều kiện while (turn == j && flag[j] == TRUE). Nếu tiến trình j muốn vào và đang chờ lượt, tiến trình i sẽ đợi (chờ cho đến khi turn không còn bằng j hoặc flag[j] == FALSE).
  4. Khi các điều kiện cho phép, tiến trình i vào phần vùng quan trọng.
  5. Sau khi ra khỏi phần vùng quan trọng, tiến trình i đặt flag[i] = FALSE để thông báo rằng mình không còn muốn vào phần vùng quan trọng.

### Các yêu cầu của một giải pháp đồng bộ:

* **Mutual Exclusion (Độc quyền)**:
  + Đảm bảo rằng chỉ có một tiến trình vào phần vùng quan trọng tại một thời điểm.
  + Trong giải pháp này, nếu cả hai tiến trình đều muốn vào phần vùng quan trọng, chỉ một tiến trình sẽ được cho phép vào, dựa vào biến turn. Nếu một tiến trình i có lượt, và tiến trình j muốn vào, nó sẽ phải đợi cho đến khi turn không còn bằng i. Điều này đảm bảo rằng chỉ một tiến trình sẽ vào phần vùng quan trọng, thỏa mãn yêu cầu **mutual exclusion**.
* **Progress (Tiến trình)**:
  + Đảm bảo rằng nếu không có tiến trình nào đang ở trong phần vùng quan trọng, thì sẽ có tiến trình tiếp theo vào mà không bị tắc nghẽn.
  + Giải pháp này thỏa mãn yêu cầu **progress**. Khi một tiến trình ra khỏi phần vùng quan trọng, nó sẽ thay đổi giá trị turn và cho phép tiến trình khác vào mà không bị tắc nghẽn vô hạn.
* **Bounded Waiting (Giới hạn chờ đợi)**:
  + Đảm bảo rằng không có tiến trình nào phải chờ đợi quá lâu trước khi có cơ hội vào phần vùng quan trọng.
  + Giải pháp này cũng thỏa mãn yêu cầu **bounded waiting**. Mỗi tiến trình sẽ có lượt vào phần vùng quan trọng sau một số lần không quá giới hạn, vì biến turn và flag sẽ thay đổi theo cách hợp lý. Mặc dù có thể có một tiến trình phải đợi trong một số tình huống, nhưng không có tiến trình nào sẽ phải đợi vô hạn vì mỗi tiến trình sẽ có lượt vào trong một vòng lặp.

### Kết luận:

Giải pháp này thỏa mãn các yêu cầu của một giải pháp đồng bộ:

* **Mutual exclusion**: Đảm bảo chỉ có một tiến trình vào phần vùng quan trọng tại một thời điểm.
* **Progress**: Tiến trình có cơ hội vào phần vùng quan trọng khi không có tiến trình nào khác.
* **Bounded waiting**: Không có tiến trình nào phải chờ đợi quá lâu.

Vì vậy, giải pháp này thỏa mãn yêu cầu của một giải pháp đồng bộ.

1. Xét hai tiến trình sau:

process P1 { A1 ; A2 } process P2 { B1 ; B2 }

Đồng bộ hóa hoạt động của hai tiến trình này sao cho cả A1 và B1 đều hoàn thành

trước khi B2 và B2 bắt đầu?

Để đồng bộ hóa hoạt động của hai tiến trình sao cho cả A1 và B1 đều hoàn thành trước khi B2 và B2 bắt đầu, bạn có thể sử dụng cơ chế đồng bộ hóa như **khóa (locks)**, **semaphores**, hoặc **barriers**. Dưới đây là một cách tiếp cận sử dụng **Semaphores**:

Giả sử chúng ta có một semaphore S1 để đồng bộ hóa giữa tiến trình P1 và P2.

1. Tiến trình P1 thực thi A1, sau đó thông báo rằng A1 đã hoàn thành bằng cách giải phóng semaphore S1.
2. Tiến trình P2 sẽ chờ semaphore S1 (tức là phải đợi đến khi A1 hoàn thành), và sau đó thực thi B1.
3. Sau khi B1 hoàn thành, P2 tiếp tục thực thi B2.
4. Tiến trình P1 có thể tiếp tục thực thi A2 sau khi P2 hoàn thành B1.
5. Một biến X được chia sẻ bởi hai tiến trình cùng thực hiện đoạn code sau:

do

X = X +1;

if ( X == 20) X = 0;

while ( TRUE)

Khởi đầu với giá trị X=0, chứng minh rằng giá trị của X có thể vượt quá 20. Chỉnh sửa lại chương trình trên để X không vượt quá 20?

**Chứng minh rằng giá trị của XXX có thể vượt quá 20:**

Trong chương trình ban đầu, hai tiến trình thực hiện đoạn mã sau:

do

X = X + 1;

if (X == 20) X = 0;

while (TRUE);

* Ban đầu, giá trị của X là 0.
* Trong mỗi vòng lặp:
* X được tăng lên 1.
* Nếu X=20, thì giá trị của X được đặt lại thành 0.

Vấn đề xảy ra khi hai tiến trình thực thi đoạn mã này đồng thời, gây ra tình trạng **đua tranh dữ liệu (race condition)**.

Ví dụ:

1. **Tiến trình 1** tăng X lên 19.
2. **Tiến trình 2** vào và tăng X lên 20. Sau đó, **Tiến trình 2** kiểm tra và đặt X=0.
3. **Tiến trình 1** tiếp tục và tăng X lên 21 mà không bị kiểm tra lại, vì lúc này **Tiến trình 2** đã thay đổi X về 0.

Do đó, giá trị của X có thể vượt quá 20 vì không có sự đồng bộ giữa các tiến trình.

**Chỉnh sửa chương trình để X không vượt quá 20:**

Để ngăn giá trị của X vượt quá 20, chúng ta có thể dùng một cơ chế đồng bộ để đảm bảo chỉ một tiến trình thay đổi giá trị của X tại một thời điểm. Một cách đơn giản là sử dụng một khóa (lock) để đồng bộ hóa các tiến trình. Ví dụ:

lock(mutex);

do

X = X + 1;

if (X == 20) X = 0;

while (TRUE);

unlock(mutex);

Bằng cách sử dụng khóa, chỉ một tiến trình có thể thay đổi giá trị của X tại một thời điểm, giúp tránh tình trạng đua tranh và đảm bảo rằng X không bao giờ vượt quá 20.

1. Cho 2 tiến trình sau:

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

1. Sử dụng semaphore để đồng bộ hóa xử lý hai tiến trình trên sao cho tại bất kỳ thời điểm nào cũng có **nb < na <= nb + 10**

Để đảm bảo rằng nb<na≤nb+10 , chúng ta cần đồng bộ hóa tiến trình A và B để đảm bảo rằng sự thay đổi của nanana và nb diễn ra một cách có kiểm soát.

Giải pháp sử dụng **semaphore** có thể như sau:

* **Semaphore** mutex: Sử dụng để bảo vệ việc thay đổi các biến na và nb.
* **Semaphore** semA: Đảm bảo tiến trình A không vượt quá giới hạn nb+10.
* **Semaphore** semB: Đảm bảo tiến trình B không vượt qua giới hạn của tiến trình A.

Đoạn mã sử dụng semaphore sẽ trông như sau:

semaphore mutex = 1; // Khóa chung cho cả A và B

semaphore semA = 1; // Đảm bảo A không vượt quá nb + 10

semaphore semB = 1; // Đảm bảo B không vượt quá na

process A {

while (TRUE) {

wait(mutex);// Đảm bảo an toàn khi thay đổi na và nb

wait(semA); // Đảm bảo na không vượt quá nb + 10

na = na + 1;

signal(semA); // Thả semA sau khi thay đổi na

signal(mutex); // Thả mutex sau khi thay đổi xong

}

}

process B {

while (TRUE) {

wait(mutex);// Đảm bảo an toàn khi thay đổi na và nb

wait(semB); // Đảm bảo nb < na

nb = nb + 1;

signal(semB); // Thả semB sau khi thay đổi nb

signal(mutex); // Thả mutex sau khi thay đổi xong

}

}

**Giải thích:**

* **mutex**: Đảm bảo rằng chỉ một tiến trình có thể thay đổi giá trị của na và nb tại một thời điểm.
* **semA**: Đảm bảo rằng khi tiến trình A chạy, giá trị của na luôn nhỏ hơn hoặc bằng nb + 10. Nếu nanana sắp vượt quá nb+10, tiến trình A sẽ phải đợi.
* **semB**: Đảm bảo rằng nb<nanb < nanb<na, tức là tiến trình B sẽ phải đợi khi nbnbnb bằng hoặc lớn hơn na.

1. Cập nhật giải pháp với điều kiện **na <= nb +10**

Để đảm bảo na ≤ nb+10, điều kiện đồng bộ hóa sẽ thay đổi một chút. Chúng ta cần điều chỉnh lại **semA** và **semB** sao cho:

* Tiến trình A có thể tiếp tục tăng na miễn là na≤nb+10
* Tiến trình B có thể tiếp tục tăng nb miễn là nb≤na.

Cập nhật lại chương trình như sau:

semaphore mutex = 1; // Khóa chung cho cả A và B

semaphore semA = 1; // Đảm bảo na <= nb + 10

semaphore semB = 1; // Đảm bảo nb <= na

process A {

while (TRUE) {

wait(mutex); // Đảm bảo an toàn khi thay đổi na và nb

wait(semA); // Đảm bảo na <= nb + 10

na = na + 1;

signal(semA); // Thả semA sau khi thay đổi na

signal(mutex); // Thả mutex sau khi thay đổi xong

}

}

process B {

while (TRUE) {

wait(mutex); // Đảm bảo an toàn khi thay đổi na và nb

wait(semB); // Đảm bảo nb <= na

nb = nb + 1;

signal(semB); // Thả semB sau khi thay đổi nb

signal(mutex); // Thả mutex sau khi thay đổi xong

}

}

**Giải thích:**

* **semA**: Đảm bảo rằng na ≤ nb+10 khi tiến trình A thực hiện.
* **semB**: Đảm bảo rằng nb ≤ na khi tiến trình B thực hiện.

1. Giải phải còn đúng nếu có nhiều điều kiện tiến trình loại A và B cùng thực hiện

Giải pháp vẫn đúng nếu có nhiều tiến trình A và B cùng thực hiện, vì các semaphore sẽ đảm bảo rằng:

* Tiến trình A chỉ có thể thay đổi na khi không vượt quá nb+10 (giữa các tiến trình A).
* Tiến trình B chỉ có thể thay đổi nb khi nb ≤ na (giữa các tiến trình B).

Tuy nhiên, cần chú ý rằng khi có nhiều tiến trình A và B, semaphore sẽ đảm bảo rằng mỗi tiến trình A hoặc B chỉ thực hiện thay đổi khi điều kiện đồng bộ hóa được thỏa mãn.

**Kết luận**: Giải pháp đồng bộ với semaphore vẫn đảm bảo điều kiện cho nhiều tiến trình A và B đồng thời thực hiện mà không vi phạm yêu cầu nb < na ≤ nb+10 hoặc na ≤ nb+10.

## II. 2. Bài tập nâng cao:

1. Tìm hiểu và sử dụng các lớp Mutex, Semaphore, Monitor của C#.

**Tổng quan:**

* **Mutex:** Được dùng để đảm bảo rằng chỉ có một luồng (thread) tại một thời điểm có thể truy cập tài nguyên được chia sẻ.
* **Semaphore:** Cho phép nhiều luồng truy cập tài nguyên cùng lúc nhưng có số lượng giới hạn.
* **Monitor:** Cung cấp một cách đơn giản để đồng bộ hóa truy cập tới tài nguyên, với cơ chế lock và khả năng chờ/signal (thường được dùng qua từ khóa lock trong C#).

**Ví dụ sử dụng trong C#:**

// Ví dụ sử dụng Mutex

Mutex mutex = new Mutex();

void CriticalSection() {

mutex.WaitOne();

try {

// Thực hiện các thao tác cần đồng bộ

} finally {

mutex.ReleaseMutex();

}

}

// Ví dụ sử dụng Semaphore

Semaphore sem = new Semaphore(initialCount: 3, maximumCount: 3);

void AccessResource() {

sem.WaitOne();

try {

// Thực hiện các thao tác

} finally {

sem.Release();

}

}

// Ví dụ sử dụng Monitor với từ khóa lock

object lockObj = new object();

void ProcessData() {

lock (lockObj) {

// Vùng mã được đồng bộ

}

}

1. Áp dụng viết chương trình minh họa giải bài toán Producer-Consumer.

**Giới thiệu bài toán:**  
Bài toán Producer-Consumer yêu cầu các tiến trình producer (sản xuất) đưa dữ liệu vào một buffer chia sẻ và các tiến trình consumer (tiêu thụ) lấy dữ liệu ra từ buffer đó. Cần đảm bảo rằng producer không ghi đè dữ liệu khi buffer đầy và consumer không lấy dữ liệu khi buffer rỗng.

**Ví dụ sử dụng Monitor trong C#:**

using System;

using System.Collections.Generic;

using System.Threading;

class ProducerConsumer {

private Queue<int> buffer = new Queue<int>();

private int capacity = 5; // Kích thước buffer

private object lockObj = new object();

public void Produce() {

int item = 0;

while (true) {

lock (lockObj) {

while (buffer.Count >= capacity) {

// Buffer đầy, producer chờ

Monitor.Wait(lockObj);

}

item++;

buffer.Enqueue(item);

Console.WriteLine("Produced: " + item);

// Thông báo cho consumer

Monitor.PulseAll(lockObj);

}

Thread.Sleep(500); // Giả lập thời gian sản xuất

}

}

public void Consume() {

while (true) {

int item;

lock (lockObj) {

while (buffer.Count == 0) {

// Buffer rỗng, consumer chờ

Monitor.Wait(lockObj);

}

item = buffer.Dequeue();

Console.WriteLine("Consumed: " + item);

// Thông báo cho producer

Monitor.PulseAll(lockObj);

}

Thread.Sleep(1000); // Giả lập thời gian tiêu thụ

}

}

}

class Program {

static void Main() {

ProducerConsumer pc = new ProducerConsumer();

Thread producerThread = new Thread(pc.Produce);

Thread consumerThread = new Thread(pc.Consume);

producerThread.Start();

consumerThread.Start();

producerThread.Join();

consumerThread.Join();

}

}

**Giải thích:**

* Producer sẽ sản xuất dữ liệu và thêm vào buffer, nếu buffer đầy sẽ chờ (Monitor.Wait).
* Consumer sẽ lấy dữ liệu từ buffer, nếu buffer rỗng sẽ chờ (Monitor.Wait).

Sau mỗi lần thay đổi buffer, dùng Monitor.PulseAll để đánh thức các luồng đang chờ

## Tài liệu tham khảo:

Trần Hạnh Nhi, Bài giảng Hệ điều hành, Trường Đại học KHTN TpHCM.