**BỘ CÔNG THƯƠNG**

**TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THỰC PHẨM**

**THÀNH PHỐ HỒ CHÍ MINH**

**ĐỒ ÁN KẾT THÚC MÔN HỌC**

**Tìm hiểu phương pháp đồng bộ hóa tiến trình Semaphore - viết ứng dụng minh họa bài toán tiến trình sản xuất, tiêu thụ (Producer – Consumer).**

**GVBM: Phạm Tuấn Khiêm**

**Sinh viên thực hiện**:

* Từ Huệ Sơn 2001190791
* Lê Đức Tài 2001190794
* Trần Thành Tâm 2001190248
* Trương Văn Bình 2001210071
* Phạm Văn Long 2001210918

*TP. HỒ CHÍ MINH, tháng 06 năm 2022*

**Hệ điều hành**

**Đề tài: Tìm hiểu phương pháp đồng bộ hóa tiến trình Semaphore - viết ứng dụng minh họa bài toán tiến trình sản xuất, tiêu thụ (Producer – Consumer).**

|  |  |
| --- | --- |
| **Điểm** | **Lời phê của Giảng Viên** |

|  |  |  |
| --- | --- | --- |
| **MSSV** | **HỌ TÊN** | **CÔNG VIỆC** |
| 2001190791 | Từ Huệ Sơn | Demo |
| 2001190794 | Lê Đức Tài | Tìm hiểu về bài toán Producer & consumer |
| 2001190248 | Trần Thành Tâm | So sánh giữa semaphore & mutex |
| 2001210918 | Phạm Văn Long | Tổng quan về semaphore |
| 2001210071 | Trương Văn Bình | Tổng quan về semaphore |

**Sinh viên thực hiện**:

MỤC LỤC

[Chương 1: Tìm hiểu đồng bộ hóa 1](#_Toc104271768)

[1.1 Nhu cầu đồng bộ hóa (Synchronisation) 1](#_Toc104271769)

[1.2 Đồng bộ hóa là gì ? 1](#_Toc104271770)

[1.3 Vùng tranh chấp 1](#_Toc104271771)

[1.3.1 Khái niệm 1](#_Toc104271772)

[1.3.2 Mutual exclusion (mutex) 2](#_Toc104271773)

[1.3.3 Cấu trúc tổng quát của tiến trình khi vào vùng tranh chấp 2](#_Toc104271774)

[1.3.4 Nguyên tắc giải quyết tranh chấp 3](#_Toc104271775)

[1.3.5 Phương pháp giải quyết tranh chấp 3](#_Toc104271776)

[1.4 Vấn đề đồng bộ 3](#_Toc104271777)

[1.4.1 Giải pháp sử dụng biến cờ hiệu (Semaphore) 6](#_Toc104271778)

[Chương 2: Tìm hiểu về Semaphore 8](#_Toc104271779)

[2.1 Giới thiệu Edsger Wybe Dijkstra 8](#_Toc104271780)

[2.2 Kỹ thuật đèn báo (Semaphore) 8](#_Toc104271781)

[2.2.1 Khi tiến trình gọi P(S) và đèn báo S không dương 9](#_Toc104271782)

[2.2.2 Khi tiến trình đưa ra lời gọi V(S) 9](#_Toc104271783)

[2.2.3 Ưu điểm và nhược điểm kỹ thuật đèn báo (Semaphore) 9](#_Toc104271784)

[2.2.3.1 Ưu điểm của kỹ thuật đèn báo 9](#_Toc104271785)

[2.2.3.2 Nhược điểm của kĩ thuật đèn báo 10](#_Toc104271786)

[2.3 So sánh Semaphore với Mutex 10](#_Toc104271787)

[2.4 Những điểm khác nhau giữa Semaphore và Monitor 11](#_Toc104271788)

[2.4.1 Những điểm giống nhau giữa Semaphore và Monitor 13](#_Toc104271789)

[Chương 3: ÁP DỤNG VÀ DEMO 14](#_Toc104271790)

# Tìm hiểu đồng bộ hóa

## Nhu cầu đồng bộ hóa (Synchronisation)

Hệ thống máy tính hiện nay của chúng ta là hệ thống đa nhiệm do đó thì có nhiều tiến trình ở trong máy tính. Và tiến trình ở trong máy tính có 2 loại:

* Các tiến trình có nhu cầu liên lạc với nhau
* Các tiến trình không có nhu cầu liên lạc với nhau (các tiến trình làm việc theo lô)

Hệ điều hành luôn cần cung cấp những cơ chế đồng bộ hóa để đảm bảo hoạt động đồng thời giữa các tiến trình mà các các tiến trình này hoạt động sẽ không làm sai lệch kết quả của tiến trình khác.

Việc dẫn tới sự sai lệch các kết quả chính là sai lệch các giá trị trong vùng nhớ dùng chung

* Yêu cầu về độc quyền truy xuất (mutual exclusion) : tại một thời điểm, chỉ có một tiến trình được quyền truy xuất một tài nguyên không thể chia sẻ.
* Yêu cầu phối hợp (synchronization) : các tiến trình cần hợp tác với nhau để hoàn thành công việc. Ví dụ chương trình in sẽ xuất kí tự vào buffer, chương trình điều khiển máy in (printer driver) sẽ lấy kí tự trong buffer ra để in. Hai tiến trình này phải phối hợp với nhau để làm sao chương trình in không được xuất kí tự vào buffer khi buffer đầy mà phải chờ printer driver lấy bớt dữ liệu trong buffer ra.

## Đồng bộ hóa là gì ?

Đồng bộ các tiến trình là bảo đảm các tiến trình xử lý song song không tác động sai lệch đến nhau.

## Vùng tranh chấp

### Khái niệm

Giả sử rằng hệ thống có n tiến trình P0, P1, ..., Pn-1 cùng truy xuất đồng thời một vùng dữ liệu chia sẻ chẳng hạn như một đoạn code nào đó. Đoạn code được chia sẻ đó được gọi là vùng tranh chấp, trong vùng tranh chấp này các tiến trình có thể thay đổi biến dùng chung, cập nhật giá trị cho các dữ liệu.

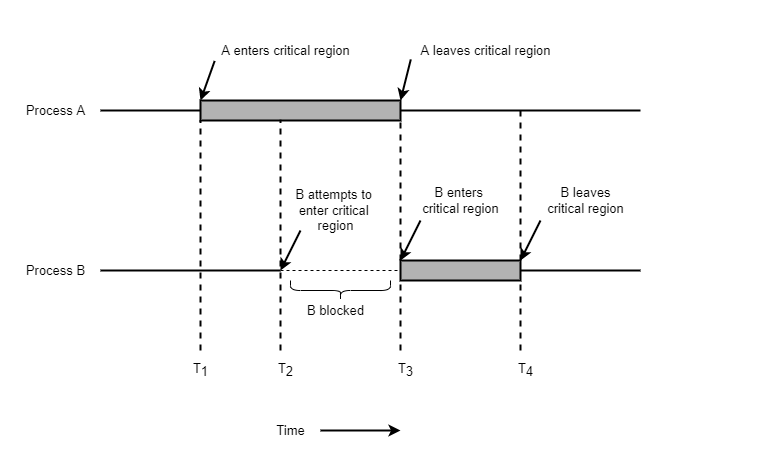
Đối với vùng tranh chấp miền găng là hệ thống phải đảm bảo sao cho khi một tiến trình đang thực thi trên vùng tranh chấp thì không có tiến trình nào khác được thực hiện tại vùng này. Sự thực thi của các tiến trình trên vùng tranh chấp được gọi là sự loại trừ tương hỗ(mutual exclusion)

### Mutual exclusion (mutex)

Mutex là một thuộc tính của đồng bộ hóa tiến trình nói rằng “không có hai tiến trình nào có thể tồn tại trong vùng tranh chấp bất kỳ thời điểm nhất định nào”. Thuật ngữ này lần đầu tiên được định nghĩa bởi Dijkstra. Bất kỳ kỹ thuật đồng bộ hóa tiến trình nào đang được sử dụng đều phải thỏa mãn thuộc tính mutex, nếu không có đặc tính này sẽ không thể loại bỏ điều kiện chủng tộc.

### Cấu trúc tổng quát của tiến trình khi vào vùng tranh chấp

Để được vào vùng tranh chấp, mỗi tiến trình phải gửi yêu cầu cho hệ điều hành và phải được chấp thuận khi đó tiến trình mới được vào vùng tranh chấp. Phần đoạn code thực thi yêu cầu này được gọi là Entry\_section(). Sau khi vào được vùng tranh chấp, tiến trình thực thi trên vùng tranh chấp được gọi là  Critial\_section(). Sau khi thực hiện xong trên vùng tranh chấp, tiến trình thoát ra khỏi  vùng tranh chấp bằng đoạn code được gọi là Exit\_section().



### Nguyên tắc giải quyết tranh chấp

* Chia nhỏ các lệnh thì chia nhỏ cho đến khi không thể chia nhỏ được nữa.
* Có thể chuyển các tiến trình động thời đồng bộ về tiến trình đồng thời bất đồng bộ.
* Tiến trình ngoài vùng tranh chấp không có quyền cấm các tiến trình khác vào vùng tranh chấp.
* Tiến trình không được trì hoãn vô hạn định khi vào vùng tranh chấp.

### Phương pháp giải quyết tranh chấp

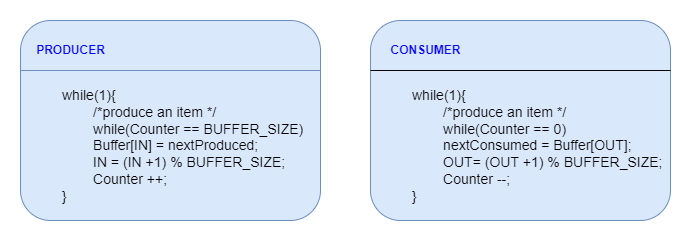
Có 2 phương pháp cơ bản:

* Phương pháp phần mềm (software solution):
  + Người sử dụng tự thực hiện thông thường có sự hỗ trợ của lập trình viên
  + Hệ điều hành cung cấp công cụ hỗ trợ lập trình viên qua lời gọi hệ thống
* Giải pháp phần cứng: Dựa trên một số lệnh đặc biệt interrupt disable, test and set.

## Vấn đề đồng bộ

Bài toán liên quan

* **Bài toán : Producer – Consumer**

****

Producer (nhà sản xuất)

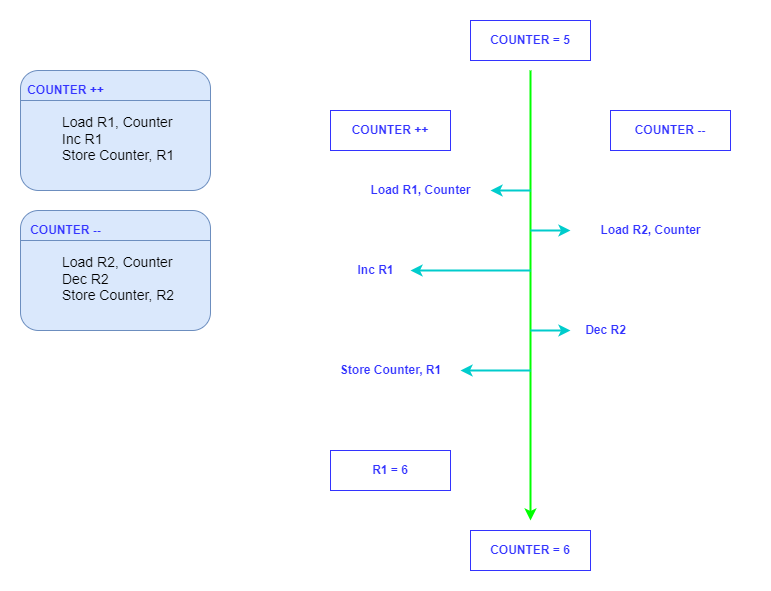
Nếu như mà biến đếm (Counter) bằng **BUFFER\_SIZE** (là vùng nhớ mà ta khai báo) mà Counter đếm đã đầy không thể sản xuất được nữa thì nó sẽ dừng lại. Còn không bằng thì vòng lặp while sai nó sẽ thoát ra và sẽ tạo **Buffer[IN] = nextProduced** (Buffer của con trỏ IN) để lưu trữ thêm sản phẩm được tạo ra (bởi vì con trỏ trỏ đến vùng nhớ trống trong bộ nhớ đệm) và con trỏ **IN = (IN+1) % BUFFER\_SIZE** sẽ được tăng lên vùng nhớ tiếp theo và Counter sẽ tăng lên 1(**Counter ++**)

Consumer (Người tiêu thụ)

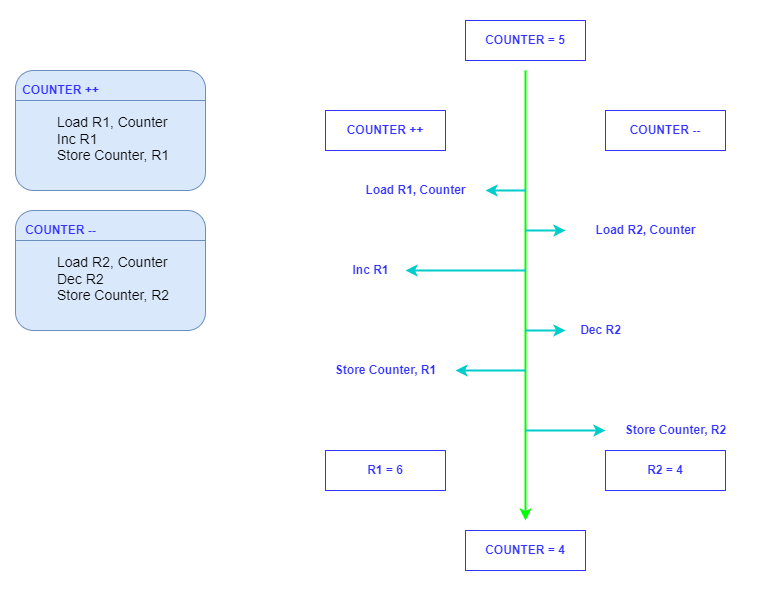
Nếu Counter = 0 (có nghĩa là trong hàng đợi của nó không có sản phẩm nào thì làm sao người tiêu thụ có thể tiêu thụ được) thì kết thúc chờ while .Ngược lại, nếu Counter khác 0 (có nghĩa là có sản phẩm) thì sẽ sử dụng sản phẩm được lưu trữ trong Buffer mà con trỏ OUT trỏ vào **nextConsumed = Buffer[OUT]**. Sau khi trỏ vào Consumer lấy thì sẽ trỏ đến ô tiếp theo **OUT = (OUT+1) % BUFFER\_SIZE** để lần sau Consumer lấy ô đó, mỗi lần Consumer lấy thì Counter sẽ giảm đi 1(**Counter --**)

Giả sử Producer chạy để lưu sản phẩm thứ 7 thì đến lượt Consumer thí nó sẽ tiêu thụ 1 sản phẩm thì Counter sẽ giảm từ 7 sang 6. Như vậy, Producer cứ sản xuất 1 mà Consumer lại tiêu thụ 1 thì nó sẽ gọi đến Counter ++ và Counter --. Counter chính là biến đếm để Producer biết được nó đã sản xuất đầy BUFFER\_SIZE hay chưa nếu đầy thì dừng lại không sản xuất được nữa. Và Consumer cũng vậy sẽ kiểm tra BUFFER\_SIZE đó hết hay chưa nếu hết thì nó không tiêu thụ nữa => Buffer không đổi

Vấn đề đồng bộ ở đây liên quan đến Counter ++ trong Producer và Counter -- Consumer. Ở đây Counter ++ sẽ chạy làn ba lệnh là: load Counter vào thanh ghi R1 (Load R1, Counter), tăng giá trị trong thanh ghi R1 (Inc R1), lưu trữ thanh ghi R1 vào Counter (**Store Counter, R1**). Counter -- sẽ tương tự cũng gồm ba lệnh là: load Counter vào thanh ghi R2 (**Load R2, Counter**), giảm giá trị trong thanh ghi R2 (**Dec R2**), lưu trữ thanh ghi R2 vào Counter (**Store Counter, R2**)



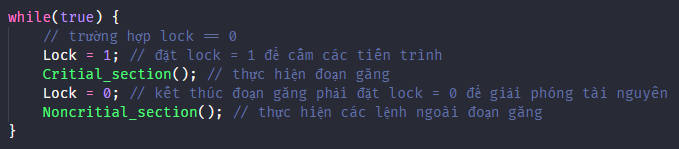
Nếu Counter ++ được chạy trước thì nó sẽ bắt đầu chạy dòng lệnh Load R1, Counter (Counter = 5) thì R1 = 5 và hết thời gian thì sẽ đến lượt tiến trình Counter – thực hiện, thì nó sẽ chạy dòng lệnh đầu tiên là load R2, Counter thì R2 = 5 (bởi vì Counter là biến dùng chung và Counter = 5). Giả sử đang sử dụng thuật toán Round Robin, mỗi tiến trình có lượng tử thời gian chiếm dụng q = 1. Counter -- hết thời gian thì Counter ++ thực hiện lệnh Inc R1 (tăng R1) lúc này R1 = 6. Tiếp tới thì đến Counter -- thực hiện thì chạy câu lệnh Dec R2 (giảm R2) khi đó R2 = 4. Sau đó, Counter ++ chạy cây lệnh cuối cùng là Store Counter, R1 (lưu trữ R1 vào Counter) và R1 lúc này đang bằng 6 do đó nó sẽ lưu trữ vào biến dùng chung là Counter = 6.



Tiến trình Counter – vẫn còn 1 dòng lệnh chưa chạy và tiếp tục chạy tiếp lệnh **Store Counter, R2** (lưu trữ R2 vào Counter) thì Counter = 4. Giá trị ban đầu của Counter = 5 như vậy sẽ gây ra sai lệch giữa Producer và Consumer thì lúc này lượng sản phẩm được sản xuất và lượng tiêu thụ nó sẽ bị sai.

### Giải pháp sử dụng biến cờ hiệu (Semaphore)

* **Ý tưởng**
* Các tiến trình sẽ chia sẻ một biến chung đóng vai trò là cờ hiệu hay là một cái khóa (lock). Ban đầu biến này được khởi động là 0.
* Một tiến trình muốn vào đoạn găng, trước tiên giá trị của biến lock. Nếu lock = 0, tiến trình đặt lại giá trị cho lock = 1 và đi vào đoạn găng. Nếu lock đang nhận giá trị 1, tiến trình phải chờ khi có lock có giá trị là 0
* **Triển khai**



# Tìm hiểu về Semaphore

## Giới thiệu Edsger Wybe Dijkstra

Edsger Wybe Dijkstra (11 tháng 5 năm 1930 tại Rotterdam – 6 tháng 8 năm 2002 tại Nuenen), là nhà khoa học máy tính người Hà Lan. Ông được nhận giải thưởng Turing năm 1972 cho các đóng góp có tính chất nền tảng trong lĩnh vực ngôn ngữ lập trình. Không lâu trước khi chết, ông đã được nhận giải Bài báo ảnh hưởng lớn trong lĩnh vực tính toán phân tán của ACM dành cho bài báo đã khởi đầu cho ngành con Tự ổn định. Sau khi ông qua đời, giải thưởng thường niên này đã được đổi tên thành giải thưởng ACM Edsger W. Dijkstra.

Edsger Wybe Dijkstra (người Hà Lan) phát minh ra khái niệm semaphore trong khoa học máy tính vào năm 1965. Semaphore được sử dụng lần đầu tiên trong cuốn sách “The operating system” của ông

## Kỹ thuật đèn báo (Semaphore)

Là một biến nguyên S, khởi tạo bằng khả năng phục vụ của tài nguyên nó điều độ

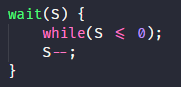
* Số tài nguyên có thể phục vụ tại một thời điểm (VD: 3 máy in)
* Số đơn vị tài nguyên có sẵn (VD: 10 chỗ trống trong buffer)

Semaphore gồm 2 loại:

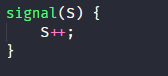
* **Binary Semaphore**
  + Đây còn được gọi là khóa mutex. Nó chỉ có thể có hai giá trị - 0 và 1. Giá trị của nó được khởi tạo bằng 1. Nó được sử dụng để thực hiện giải pháp của các vấn đề phần quan trọng với nhiều quy trình.
* **Counting Semaphore**
  + Giá trị của nó có thể nằm trong phạm vi miền không hạn chế. Nó được sử dụng để kiểm soát quyền truy cập vào một tài nguyên có nhiều phiên bản.

Chỉ có thể thay đổi giá trị bởi 2 toán tử cơ bản P (Proberen) và V (Verhogen)

* Toán tử P(S) (wait(S))



* Toán tử V(S) (signal(S))



* Các thao tác P(S) và V(S) xử lý không được tách rời

### Khi tiến trình gọi P(S) và đèn báo S không dương

* Tiến trình phải dừng bởi gọi tới câu lệnh block()
* Lệnh block() đặt tiến trình vào bảng hàng đợi gắn với đèn báo S
* Hệ thống lấy lại CPU giáo cho tiến trình khác (điều phối CPU)
* Tiến trình chuyển sang trạng thái chờ đợi (waiting)
* Tiến trình nằm trong hàng đợi đến khi tiến trình khác thực hiện thao tác V(S) trên cùng đèn báo

### Khi tiến trình đưa ra lời gọi V(S)

* Lấy một tiến trình trong hàng đợi ra (nếu có)
* Chuyển tiến trình lấy ra từ trạng thái chờ sang trạng thái sẵn sàng và đặt lên hàng đợi sẵn sàng gọi tới bởi wakeup(P)
* Tiến trình mới sẵn sàng có thể trưng dụng CPU từ tiến trình đang thực hiện nếu thuật toán điều phối CPU cho phép

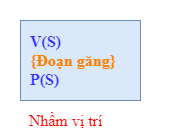
### Ưu điểm và nhược điểm kỹ thuật đèn báo (Semaphore)

#### Ưu điểm của kỹ thuật đèn báo

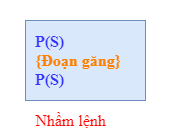
* Dễ dàng áp dụng cho các hệ thống phức tạp
* Vì có busy waiting trong semaphore, nên không bao giờ lãng phí thời gian xử lý và tài nguyên.
* Cho phép nhiều luồng truy cập vào miền găng
* Không cho phép nhiều tiến trình vào miền găng.
* Semaphores không phụ thuộc vào máy móc.

#### Nhược điểm của kĩ thuật đèn báo

* Hiệu quả sử dụng phụ thuộc vào người dùng
* Nhầm vị trí lệnh



* Nhầm lệnh



* Các phép xử lý P(S) và V(S) không phân chia được
* P(S) và V(S) cũng là tài nguyên găng
* Cần phải điều độ
* Hệ thống 1 vi xử lý: cấm ngắt khi thực hiện wait(), signal()
* Hệ thống nhiều vi xử lý:
* Không thể cấm ngắt trong vi xử lý ngắt
* Có thể dùng phương pháp khóa trong

## So sánh Semaphore với Mutex

|  |  |  |
| --- | --- | --- |
| **Các yếu tố so sánh** | **Semaphore** | **Mutex** |
| **Căn bản** | Semaphore là một cơ chế báo hiệu. | Mutex là một cơ chế khóa. |
| **Sự tồn tại** | Semaphore là một biến số nguyên. | Mutex là một đối tượng. |
| **Chức năng** | Semaphore cho phép nhiều luồng chương trình truy cập vào một thể hiện hữu hạn của tài nguyên. | Mutex cho phép nhiều luồng chương trình truy cập vào một tài nguyên nhưng không đồng thời. |
| **Quyền sở hữu** | Giá trị semaphore có thể được thay đổi bởi bất kỳ quá trình có được hoặc giải phóng tài nguyên. | Khóa đối tượng Mutex chỉ được phát hành bởi quá trình đã có được khóa trên nó. |
| **Phân loại** | Semaphore có thể được phân loại thành đếm semaphore và semaphore nhị phân. | Mutex không được phân loại thêm. |
| **Hoạt động** | Giá trị semaphore được sửa đổi bằng cách sử dụng thao tác Wait () và signal (). | Đối tượng Mutex bị khóa hoặc mở khóa bởi quá trình yêu cầu hoặc giải phóng tài nguyên. |
| **Tài nguyên chiếm dụng** | Nếu tất cả các tài nguyên đang được sử dụng, quá trình yêu cầu tài nguyên thực hiện thao tác Wait () và tự chặn cho đến khi số lượng semaphore trở nên lớn hơn một. | Nếu một đối tượng mutex đã bị khóa, quá trình yêu cầu tài nguyên chờ và được hệ thống xếp hàng cho đến khi khóa được giải phóng. |

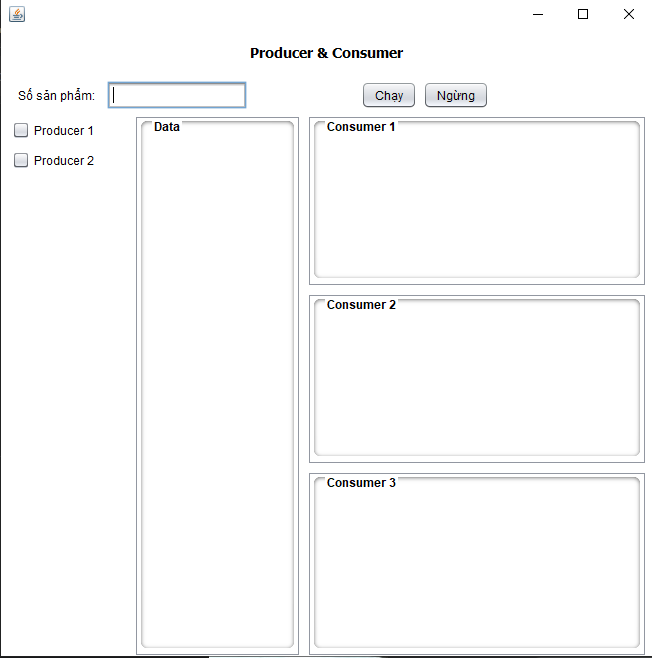
## Những điểm khác nhau giữa Semaphore và Monitor

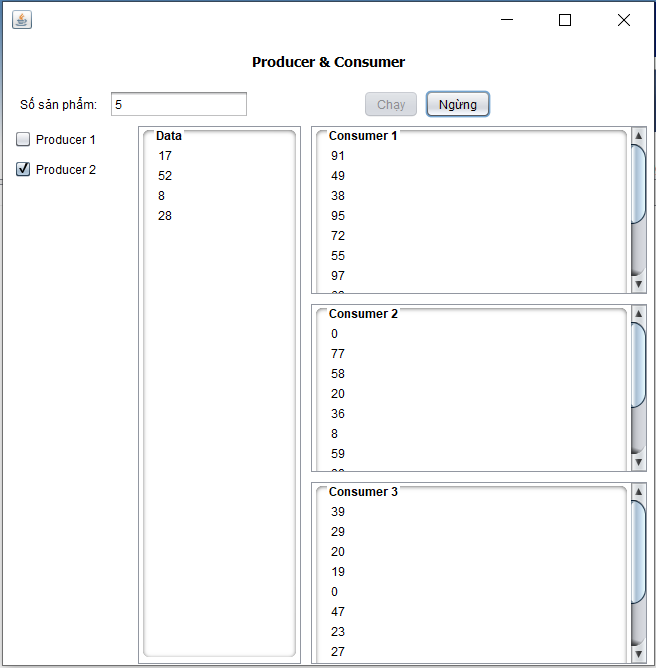
|  |  |  |
| --- | --- | --- |
| **Các yếu tố so sánh** | **Semaphore** | **Monitor** |
| **Cấu trúc cơ bản** | Semaphores là một số nguyên S. Hay nói cách semaphore chỉ là một biến đệm, nhưng được bảo đảm cho việc truy xuất đa luồng  (thread-safe). | Monitor là một kiểu dữ liệu trừu tượng (abstract data khác, semaphore chỉ là một type). |
| **Mức độ trừu tượng** | Cấp độ trừu tượng thấp hơn Monitor. | Cấp độ trừu tượng cao hơn Semaphore. Monitor có thể được cài đặt bằng Semaphore. |
| **Cách thức hoạt động** | Giá trị của Semaphore S biểu thị số lượng tài nguyên có sẵn trong hệ thống (available resources). | Monitor chứa các biến dùng chung và các phương thức thao tác lên các biến dùng chung đó. |
| **Truy cập** | Khi bất kỳ tiến trình nào cần truy cập vào tài nguyên dùng chung (shared resources), tiến trình đó sẽ phải gọi là wait() với semaphore S. Sau khi xong việc truy xuất tài nguyên dùng chung, nó sẽ trả lại tài nguyên rồi gọi là signal với semaphore S. | Khi bất kỳ tiến trình nào cần truy cập các biến dùng chung, tiến trình đó cần phải truy cập biến đó thông qua các phương thức trong Monitor. |
| **Các biến điều kiện**  **(Condition Variable)** | Semaphore không có biến điều kiện. | Monitor có các biến điều kiện. |
| **An toàn** | Semaphore cấp thấp hơn và yêu cầu người sử dụng cần thận trọng trong việc quản lý tài nguyên. | Monitor cấp cao hơn, tự động quản lý việc signal() và wait(), khiến cho việc sử dụng đơn giản và an toàn hơn. |

### Những điểm giống nhau giữa Semaphore và Monitor

* Cả semaphore và monitor đều được dùng như là một công cụ để đồng bộ hoá. Cả 2 đều có thể thực hiện công việc như nhau và thay thế được cho nhau dễ dàng.
* Điểm khác biệt có thể thấy rõ nhất, như đã nói ở bảng trên, đó là monitor dễ sử dụng hơn semaphore. Semaphore giống như con trỏ trong C++, mạnh mẽ và low-level nhưng lại yêu cầu sự cẩn thận, chu đáo từ người sử dụng nó.
* Nếu bạn wait() mà quên release() trên semaphore, bạn sẽ rất có thể gặp phải trường hợp deadlock, hoặc những bug khó phát hiện. Monitor, ngược lại, đã giúp bạn giải quyết vấn đề trên bằng cách gọi hàm thay vì phải tự kiểm soát signal() và wait().

# ÁP DỤNG VÀ DEMO

Giao diện ứng dụng



Kết quả chạy demo