**­­­­TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**VIỆN CÔNG NGHỆ THÔNG TIN & TRUYỀN THÔNG**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***



**HỆ ĐIỀU HÀNH**

**Đề tài số 10**

**Xây dựng chương trình minh họa giải pháp cho bài toán Bữa tối của triết gia**

**Giảng viên hướng dẫn:** TS. Đỗ Quốc Huy

**Sinh viên thực hiện:**

Đỗ Gia Ân - 20207652

Hoàng Hải My - 20176102

Nguyễn Việt Anh - 20187292

***Hà Nội, 1 – 2022***

# MỤC LỤC

[MỤC LỤC 2](#_Toc93336013)

[LỜI NÓI ĐẦU 3](#_Toc93336014)

[CHƯƠNG 1: GIỚI THIỆU VỀ ĐỀ TÀI 4](#_Toc93336015)

[1. Yêu cầu đề tài 4](#_Toc93336016)

[2. Mô tả đề tài 4](#_Toc93336017)

[3. Yêu cầu bài toán 5](#_Toc93336018)

[CHƯƠNG 2: CƠ SỞ LÝ THUYẾT 6](#_Toc93336019)

[1. Tiến trình (proccess) 6](#_Toc93336020)

[a. Khái niệm 6](#_Toc93336021)

[b. Định nghĩa tiến trình 6](#_Toc93336022)

[c. Các loại tiến trình 6](#_Toc93336023)

[d. Luồng (Thread) 7](#_Toc93336024)

[2. Tài nguyên găng và đoạn găng 7](#_Toc93336025)

[a. Tài nguyên găng (Critical Resource) 7](#_Toc93336026)

[b. Đoạn găng (Critical Section) 8](#_Toc93336027)

[c. Yêu cầu đối với đoạn găng 9](#_Toc93336028)

[3. Giải pháp Semaphore 9](#_Toc93336029)

[4. Deadlock 10](#_Toc93336030)

[a. Giới thiệu vấn đề: 10](#_Toc93336031)

[b. Điều kiện hình thành tắc nghẽn 10](#_Toc93336032)

[c. Ngăn chặn tắc nghẽn (Deadlock Prevention) 11](#_Toc93336033)

[CHƯƠNG 2: XÂY DỰNG BÀI TOÁN 13](#_Toc93336034)

[KẾT LUẬN 20](#_Toc93336035)

[TÀI LIỆU THAM KHẢO 21](#_Toc93336036)

LỜI NÓI ĐẦU

Nếu không có phần mềm, máy tính chỉ là một thiết bị điện tử thông thường. Với sự hỗ trợ của phần mềm, máy tính có thể lưu trữ, xử lý thông tin và người sử dụng có thể gọi lại được thông tin này. Phần mềm máy tính có thể chia thành nhiều loại: chương trình hệ thống, quản lý sự hoạt động của chính máy tính. Chương trình ứng dụng, giải quyết các vấn đề liên quan đến việc sử dụng và khai thác máy tính của người sử dụng. Hệ điều hành thuộc nhóm các chương trình hệ thống và nó là một chương trình hệ thống quan trọng nhất đối với máy tính và cả người sử dụng. Hệ điều hành điều khiển tất cả các tài nguyên của máy tính và cung cấp một môi trường thuận lợi để các chương trình ứng dụng do người sử dụng viết ra có thể chạy được trên máy tính. Một máy tính hiện đại có thể bao gồm: một hoặc nhiều processor, bộ nhớ chính, clocks, đĩa, giao diện mạng, và các thiết bị vào/ra khác. Tất cả nó tạo thành một hệ thống phức tạp. Để viết các chương trình để theo dõi tất cả các thành phần của máy tính và sử dụng chúng một cách hiệu quả, người lập trình phải biết processor thực hiện chương trình như thế nào, bộ nhớ lưu trữ thông tin như thế nào, các thiết bị đĩa làm việc (ghi/đọc) như thế nào, lỗi nào có thể xảy ra khi đọc một block đĩa, … đây là những công việc rất khó khăn và quá khó đối với người lập trình. Nhưng rất may cho cả người lập trình ứng dụng và người sử dụng là những công việc trên đã được hệ điều hành hỗ trợ nên họ không cần quan tâm đến nữa. chúng ta cần tiềm hiểu về hệ điều hành để có một cái nhìn tổng quan về những gì liên quan đến việc thiết kế cài đặt cũng như chức năng của hệ điều hành để hệ điều hành đạt được mục tiêu: Giúp người sử dụng khai thác máy tính dễ dàng và chương trình của người sử dụng có thể chạy được trên máy tính. "Bài toán bữa tối của các triết gia" (Dining Philosophers), một bài toán kinh điển về tương tranh và chia sẻ tài nguyên. Việc nghiên cứu bài toán sẽ cho chúng ta hiểu rõ hơn về khía cạnh này của hệ điều hành.

# CHƯƠNG 1: GIỚI THIỆU VỀ ĐỀ TÀI

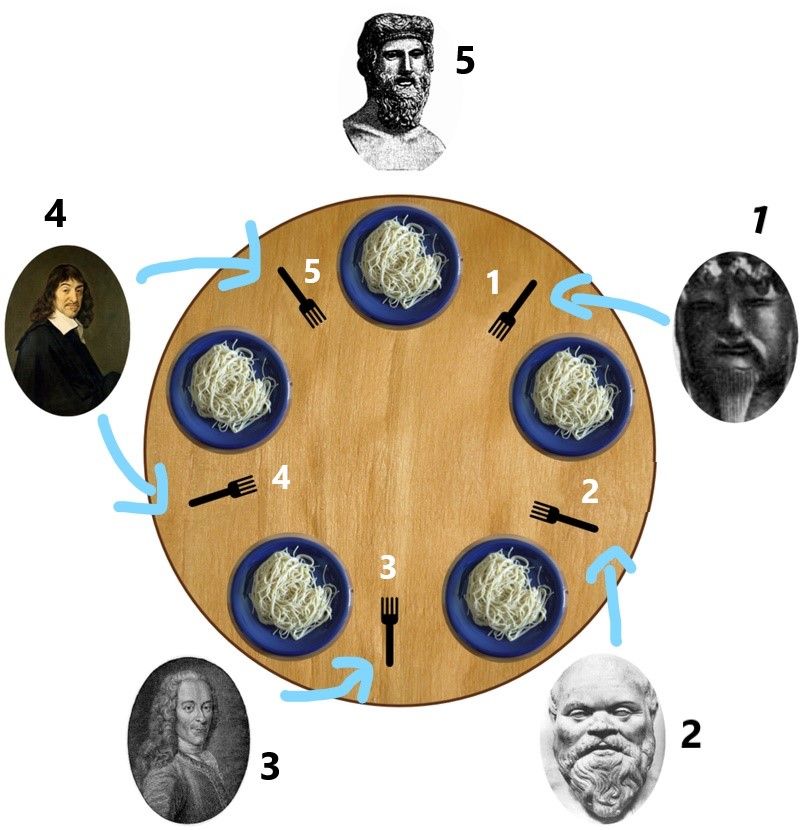
## Yêu cầu đề tài

Chương trình minh họa cần giải quyết bài toán **“Bữa ăn tối của triết gia”**. Chương trình tạo ra năm quá trình con mô phỏng hoạt động của năm triết gia. Sử dụng Semaphore để đồng bộ hoạt động của năm triết gia này.

## Mô tả đề tài

Đây là bài toán cổ điển về hệ điều hành, được đưa ra bởi nhà toán học E. W. Dijkstra. Bài toán được mô tả như sau:

* Có năm triết gia cùng ngồi ăn tối quanh một chiếc bàn tròn, trước mặt mỗi người có một đĩa mì Ý, giữa 2 triết gia thì có một chiếc nĩa. Mỗi triết gia dành toàn bộ thời gian để suy nghĩ hoặc ăn khi đói. Mỗi triết gia chỉ có thể ăn khi có được 2 chiếc nĩa bên cạnh mình.
* **Đói**: Một triết gia có thể chết đói nếu ông ta không có cách nào để ăn được.
* **Tắc nghẽn**: Các triết gia phải đợi lẫn nhau nên không có ai ăn được.



Hình 1: Minh họa bài toán

## Yêu cầu bài toán

Phải đặt ra thuật toán sao cho khi một triết gia bị đói thì ông ta sẽ được ăn và đảm bảo không có triết gia nào bị chết đói. Bài toán đặt ra vấn đề “đồng bộ giữa các tiến trình”, giải quyết vấn đề tắc nghẽn có thể xảy ra.

Thuật toán được đưa ra là thuật toán Semaphore.

# CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

## Tiến trình (proccess)

### Khái niệm

Tiến trình là một bộ phận của một chương trình đang thực hiện, đơn vị thực hiện tiến trình là processer.

Vì tiến trình là một bộ phận của chương trình nên tương tự như chương trình tiến trình cũng sở hữu một con trỏ lệnh, một con trỏ stack, một tập các thanh ghi, một không gian địa chỉ trong bộ nhớ chính và tất cả các thông tin cần thiết khác để tiến trình có thể hoạt động được.

### Định nghĩa tiến trình

Định nghĩa của Saltzer: Tiến trình là một chương trình do một processor logic thực hiện.

Định nghĩa của Horning & Rendell: Tiến trình là một quá trình chuyển từ trạng thái này sang trạng thái khác dưới tác động của hàm hành động, xuất phát từ một trạng thái ban đầu nào đó.

### Các loại tiến trình

Các tiến trình trong hệ thống có thể chia thành hai loại: tiến trình tuần tự và tiến trình song song.

Tiến trình tuần tự là các tiến trình mà điểm khởi tạo của nó là điểm kết thúc của tiến trình trước đó. Tiến trình tuần tự xuất hiện trong các hệ điều hành đơn nhiệm như hệ điều hành MSDOS

Tiến trình song song là các tiến trình mà điểm khởi tạo của tiến trình này mằn ở thân của các tiến trình khác, tức là có thể khởi tạo một tiến trình mới khi các tiến trình trước đó chưa kết thúc. Các tiến trình song song xuất hiện trong hệ điều hành đa nhiệm.

### Luồng (Thread)

Luồng là một thành phần của tiến trình sở hữu ngăn xếp và thực thi độc lập ngay trong mã lệnh của tiến trình. Nếu như hệ điều hành có nhiều tiến trình thì trong mỗi tiến trình bạn có thể tạo ra nhiều luồng hoạt dộng song song với nhau tương tự như các tiến trình hoạt động song song trong hệ điều hành. Ưu điểm của luồng là chúng hoạt động trong cùng một không gian địa chỉ của tiến trình. Tập hợp một nhóm các tuyến có thể sử dụng chung biến toàn cục, vùng nhớ heap, bảng mô tả file… của tiến trình, cơ chế liên lạc giữa các luồng đơn giản và hiệu quả hơn cơ chế liên lạc giữa các tiến trình với nhau (nếu hệ điều hành của bạn chạy trên phần cứng nhiều bộ xử lí thì tuyến thực sự chạy song song chứ không phải giả lập kiểu xoay vòng).

Ưu điểm của sử dụng luồng trong tiến trình đơn giản hơn lập trình tuần tự. Nhiều thao tác xuất nhập hoặc hiển thị dữ liệu có thể tách rời và phân cho các tuyến chạy độc lập thực thi. Ví dụ trong môi trường đồ họa, khi bạn copy một file có kích thước lớn, chương trình sẽ được thiết kế sao cho một luồng thực hiện đọc ghi dữ liệu từ đĩa, luồng khác sẽ đảm trách việc hiển thị phần trăm hoàn thành công việc cho người dùng theo dõi tiến độ.

Đối với hệ điều hành chi phí để chuyển đổi giữa ngữ cảnh của tiến trình cao và chậm hơn chi phí chuyển đổi ngữ cảnh dành cho luồng ( với tiến trình hệ điều hành phải cất thông số môi trường, thanh ghi trạng thái, hoán đổi vùng nhớ…)

Tuy nhiên, điểm yếu của việc dùng luồng đó là khả năng đổ vở của một luồng sẽ ảnh hưởng đến tất cả các luồng khác và toàn bộ tiến trình đang hoạt động. Lí do là các luồng dùng chung vùng nhớ và không gian địa chỉ của tiến trình. Ngược lại, một tiến trình bị đổ vỡ luôn được hệ điều hành cô lập hoàn toàn không gây ảnh hưởng đến các tiến trình khác. Tiến trình có thể chạy trên nhiều máy khác nhau trong khi luồng chỉ được thực thi trên một máy và trong một tiến trình.

## Tài nguyên găng và đoạn găng

### Tài nguyên găng (Critical Resource)

Trong môi trường hệ điều hành đa nhiệm - đa chương – đa người sử dụng, việc chia sẻ tài nguyên cho các tiến trình của người sử dụng dùng chung là cần thiết, nhưng nếu hệ điều hành không tổ chức tốt việc sử dụng tài nguyên dung chung của các tiến trình hoạt động đồng thời, thì không những không mang lại hiệu quả khai thác tài nguyên của hệ thống mà còn làm hỏng dữ liệu của các ứng dụng. Và nguy hiểm hơn là việc hỏng dữ liệu này có thể hệ điều hành và ứng dụng không thể phát hiện được. Việc hỏng dữ liệu của ứng dụng có thể làm sai lệch ý nghĩa thiết kế của nó. Đây là điều mà cả hệ điều hành và người lập trình đều không mong muốn.

Các tiến trình hoạt động đồng thời thường cạnh tranh với nhau trong việc sử dụng tài nguyên dùng chung. Hai tiến trình hoạt động đồng thời cùng ghi vào một không gian nhớ chung (một biến chung) trên bộ nhớ hay hai tiến trình đồng thời cùng ghi dữ liệu vào một file chia sẻ, đó là những biểu hiện của sự cạnh tranh về việc sử dụng tìa nguyên dùng chung của các tiến trình. Để các tiến trình hoạt động đồng thời không cạnh tranh hay xung đột với nhau khi sử dụng tài nguyên dùng chung hệ điều hành phải tổ chức cho các tiến trình này được độc quyền truy xuất/sử dụng trên các tài nguyên dùng chung này.

Những tài nguyên được hệ điều hành chia sẻ cho nhiều tiến trình hoạt động đồng thời dùng chung, mà có nguy cơ dẫn đến sự tranh chấp giữa các tiến trình này khi sử dụng chúng, được gọi là tài nguyên găng. Tài nguyên găng có thể là tài nguyên phần cứng hoặc tài nguyên phần mền, có thể là tài nguyên phân chia được hoặc không phân chia được, nhưng đa số thường là tài nguyên phân chia được như là: các biến chung, các file chia sẻ

### Đoạn găng (Critical Section)

Đoạn code trong các tiến trình đồng thời, có tác động đến các tài nguyên có thể trở thành tài nguyên găng được gọi là đoạn găng hay miền găng. Tức là, các đoạn code trong các chương trình dùng để truy cập đến các vùng nhớ chia sẻ, các tập tin chia sẻ được gọi là các đoạn găng.

Trong ví dụ 1 ở trên có hai đoạn găng là:

{ L1 := Count và Count := L1 }

Để hạn chế các lỗi có thể xảy ra do sử dụng tài nguyên găng, hệ điều hành phải điều khiển các tiến trình sao cho, tại một thời điểm chỉ có một tiến trình nằm trong đoạn găng, nếu có nhiều tiến trình cùng muốn vào (thực hiện) đoạn găng thì chỉ có một tiến trình được vào, các tiến trình khác phải chờ, một tiến trình khi ra khỏi (kết thúc) đoạn găng phải báo cho hệ điều hành và/hoặc các tiến trình khác biết để các tiến trình này vào đoạn găng, vv. Các công tác điều khiển tiến trình thực hiện đoạn găng của hệ điều hành được gọi là điều độ tiến trình qua đoạn găng. Để công tác điều độ tiến trình qua đoạn găng được thành công, thì cần phải có sự phối hợp giữa vi xử lý, hệ điều hành và người lập trình. Vi xử lý đưa ra các chỉ thị, hệ điều hành cung cấp các công cụ để người lập trình xây dựng các sơ đồ điều độ hợp lý, để đảm bảo sự độc quyền trong việc sử dụng tài nguyên găng của các tiến trình.

### Yêu cầu đối với đoạn găng

Đoạn găng phải thỏa các yêu cầu sau :

* Tại một thời điểm không thể có hai tiến trình nằm trong đoạn găng.
* Nếu có nhiều tiến trình đồng thời cùng xin được vào đoạn găng thì chỉ có một tiến trình được phép vào đoạn găng, các tiến trình khác phải xếp hàng chờ trong hàng đợi.
* Tiến trình chờ ngoài đoạn găng không được ngăn cản các tiến trình khác vào đoạn găng.
* Không có tiến trình nào được phép ở lâu vô hạn trong đoạn găng và không có tiến trình phải chờ lâu mới được vào đoạn găng (chờ trong hàng đợi).
* Nếu tài nguyên găng được giải phóng thì hệ điều hành có nhiệm vụ đánh thức các tiến trình trong hàng đợi ra để tạo điều kiện cho nó vào đoạn găng.
* Các vấn đề có thể gặp phải đối với đoạn găng
* Có thể dẫn đến tắc nghẽn (Deadlock) trong hệ thống.
* Các tiến trình có thể bị đói (Stravation) tài nguyên.

## Giải pháp Semaphore

Semaphore là một đóng góp quan trọng khác của nhà toán học E. W. Dijkstra.

Có thể xem Semaphore như là một mở rộng của Mutex locks.

Semaphore (sự đánh tín hiệu bằng cờ) S là một biến nguyên, khởi gán bằng một giá trị không âm, đó là khả năng phục vụ của tài nguyên găng tương ứng với nó.

Ứng với S có một hàng đợi F(s) để lưu các tiến trình đang bị blocked trên S.

Chỉ có hai thao tác Down và Up được tác động đến semaphore (sự đánh tín hiệu bằng cờ) S. Down giảm S xuống một đơn vị, Up tăng S lên một đơn vị.

Mỗi tiến trình trước khi vào đoạn găng thì phải gọi Down để kiểm tra và xác lập quyền vào đoạn găng Mỗi tiến trình ngay sau khi ra khỏi đoạn găng phải gọi Up để kiểm tra xem có tiến trình nào đang đợi trong hàng đợi hay không, nếu có thì đưa tiến trình trong hàng đợi vào đoạn găng. Khi tiến trình gọi Up thì hệ thống sẽ thực hiện.

Ở đây chúng ta cần lưu ý rằng: Down và Up là các thủ tục của hệ điều hành, nên hệ điều hành đã cài đặt cơ chế độc quyền cho nó, tức là các lệnh bên trong nó không thể tách rời nhau.

## Deadlock

### Giới thiệu vấn đề:

Trong môi truờng đa chương, nhiều quá trình có thể cạnh tranh một số giới hạn tài nguyên. Một quá trình yêu cầu tài nguyên, nếu tài nguyên không sẳn dùng tại thời điểm đó, quá trình đi vào trạng thái chờ. Quá trình chờ có thể không bao giờ chuyển trạng thái trở lại vì tài nguyên chúng yêu cầu bị giữ bởi những quá trình đang chờ khác. Trường hợp này được gọi là deadlock (khoá chết).

Trong trường hợp bài toán là khi tất cả các triết gia đều đói cùng một lúc, họ ngồi vào bàn và tất cả cùng nhấc chiếc nĩa bên tay phải của mình, và cùng chờ đợi chiếc nĩa từ hàng xóm bên tay trái dẫn đến các tiến trình bị khóa chết.

### Điều kiện hình thành tắc nghẽn

Năm 1971, Coffman đã đưa ra và chứng tỏ được rằng, nếu hệ thống tồn tại đồng thời bốn điều kiện sau đây thì hệ thống sẽ xảy ra tắt nghẽn:

* + Loại trừ lẫn nhau (mutual excution) hay độc quyền sử dụng: Đối với các tài nguyên không phân chia được thì tại mỗi thời điểm chỉ có một tiến trình sử dụng được tài nguyên.
  + Giữ và đợi (hold and wait): Một tiến trình hiện tại đang chiếm giữ tài nguyên, lại xin cấp phát thêm tài nguyên mới.
  + Không ưu tiên (No preemption): Không có tài nguyên nào có thể được
  + giải phóng từ một tiến trình đang chiếm giữ nó. Sự tắc nghẽn có thể tồn tại với ba điều kiện trên, nhưng cũng có thể không xảy ra chỉ với 3 điều kiện đó. Để chắc chắn tắc nghẽn xảy ra cần phải có điều kiện thư tư.
  + Đợi vòng tròn (Circular wait): Đây là trường hợp của ví dụ 1 mà chúng ta đã nêu ở trên. Tức là, mỗi tiến trình đang chiếm giữ tài nguyên mà tiến trình khác đang cần.

### Ngăn chặn tắc nghẽn (Deadlock Prevention)

Ngăn chặn tắc nghẽn là thiết kế một hệ thống sao cho hiện tượng tắc nghẽn bị loại trừ. Các phương thức ngăn chặn tắc nghẽn đều tập trung giải quyết bốn điều kiện gây ra tắc nghẽn, sao cho hệ thống không thể xảy ra đồng thời bốn điều kiện tắc nghẽn:

* + Đối với điều kiện độc quyền: Điều kiện này gần như không tránh khỏi, vì sự độc quyền là cần thiết đối với tài nguyên thuộc loại phân chia được như các biến chung, các tập tin chia sẻ, hệ điều hành cần phải hỗ trợ sự độc quyền trên các tài nguyên này.
  + Đối với điều kiện giữ và đợi: Điều kiện này có thể ngăn chặn bằng cách yêu cầu tiến trình yêu cầu tất cả tài nguyên mà nó cần tại một thời điểm và tiến trình sẽ bị khoá (blocked) cho đến khi yêu cầu tài nguyên của nó được hệ điều hành đáp ứng. Phương pháp này không hiệu quả. Thứ nhất, tiến trình phải đợi trong một khoảng thời gian dài để có đủ tài nguyên mới có thẻ chuyển sang hoạt động được, trong khi tiến trình chỉ cần một số ít tài nguyên trong số đó là có thể hoạt động được, sau đó yêu cầu tiếp. Thứ hai, lãng phí tài nguyên, vì có thể tiến trình giữa nhiều tài nguyên mà chỉ đến khi sắp kết thúc tiến trình mới sử dụng, và có thể đây là những tài nguyên mà các tiến trình khác đang rất cần. Ở đây hệ điều hành có thể tổ chức phân lớp tài nguyên hệ thống. Theo đó tiến trình phải trả tài nguyên ở mức thấp mới được cấp phát tài nguyên ở cấp cao hơn.
  + Đối với điều kiện No preemption: Điều kiện này có thể ngăn chặn bằng cách, khi tiến trình bị rơi vào trạng thái khoá, hệ điều hành có thể thu hồi tài nguyên của tiến trình bị khoá để cấp phát cho tiến trình khác và cấp lại đầy đủ tài nguyên cho tiến trình khi tiến trình được đưa ra khỏi trạng thái khoá.
  + Đối với điều kiện chờ đợi vòng tròn: Điều kiện này có thể ngăn chặn bằng cách phân lớp tài nguyên của hệ thống. Theo đó, nếu một tiến trình được cấp phát tài nguyên ở lớp L, thì sau đó nó chỉ có thể yêu cầu các tài nguyên ở lớp thấp hơn lớp L.

# CHƯƠNG 3: XÂY DỰNG BÀI TOÁN

Semaphores là các biến số nguyên được sử dụng để giải quyết vấn đề phần quan trọng bằng cách sử dụng hai phép toán nguyên tử, chờ và tín hiệu được sử dụng để đồng bộ hóa quy trình. Các định nghĩa của chờ và tín hiệu như sau:

Wait- Thao tác chờ làm giảm giá trị của đối số S, nếu nó là số dương. Nếu S âm hoặc bằng 0, thì không có thao tác nào được thực hiện.

|  |
| --- |
| sem\_wait(Semaphore s){  while(s<=0);  s--;  } |

Signal - Phép toán tín hiệu tăng giá trị của đối số S.

|  |
| --- |
| sem\_signal(Semaphore s){  s++;  } |

Khi một tiến trình đang chạy phần quan trọng thì một quá trình khác không thể truy cập vào phần quan trọng cho đến khi sem\_post chạy và làm cho biến s bằng 1.

Bài toán về bữa tối nhà triết học

Bài toán được phát biểu như sau: "Khi một triết gia suy nghĩ, ông ta không giao tiếp với các triết gia khác. Thỉnh thoảng, một triết gia cảm thấy đói và cố gắng chọn hai chiếc dĩa gần nhất (hai chiếc dĩa nằm giữa ông ta với hai láng giềng trái và phải). Một triết gia có thể lấy chỉ một chiếc dĩa tại một thời điểm. Chú ý, ông ta không thể lấy chiếc dĩa mà nó đang được dùng bởi người láng giềng. Khi một triết gia đói và có hai chiếc dĩa cùng một lúc, ông ta ăn mà không đặt dĩa xuống. Khi triết gia ăn xong, ông ta đặt dĩa xuống và bắt đầu suy nghĩ tiếp."

Thuật toán:

|  |
| --- |
| repeat  wait(fork[i]);  wait(fork[(i+1) mod 5]);  . . .  eat  . . .  signal(fork[i]);  signal(fork[(i+1) mod 5]);  . . .  think  . . .  until false; |

Mã nguồn:

Tạo một lớp Thread có tên là Philosopher:

|  |
| --- |
| class Philosopher implements Runnable {  private int id;  private int eattime;  private int amount=0;  private Semaphore leftFork;  private Semaphore rightFork;  //Tạo constructor cho lớp Philosopher  public Philosopher(int id, Semaphore leftFork, Semaphore rightFork) {  this.id = id;  this.leftFork = leftFork;  this.rightFork = rightFork;  }  public void run() {  ...  }  //các code tiếp theo  } |

Các triết gia sẽ nghĩ với thời gian ngẫu nhiên thông qua method think:

|  |
| --- |
| private void think() throws InterruptedException {  System.out.println("Philosopher " + (id + 1) + " is thinking.");  System.out.flush();  Thread.sleep(new Random().nextInt(10));  } |

Sau đó tạo phương thức pickUpLeftFork và pickUpRightFork. Giá trị của các semaphore là 1. Khi semaphore = 1 thì triết gia có thể lấy đũa. Sau đó sử dụng phương thức aqquire trong lớp semaphore để khóa dĩa .

|  |
| --- |
| private void pickUpLeftFork() throws InterruptedException{  if(leftFork.availablePermits() ==0){  System.out.println("Philosopher " + (id + 1) + " is waiting for left fork");  }  leftFork.acquire();  System.out.println("Philosopher " + (id + 1) + " is holding left fork.");  }  private void pickUpRightFork() throws InterruptedException{  if(rightFork.availablePermits() ==0){  System.out.println("Philosopher " + (id + 1) + " is waiting for right fork");  }  rightFork.acquire();  System.out.println("Philosopher " + (id + 1) + " is holding right fork.");  } |

Khóa chỉ có thể được gỡ khi sử dụng phương thức release trong lớp semaphore. Nếu biến semaphore = 0 thì các triết gia không được lấy dĩa mà phải đợi.

|  |
| --- |
| private void putDownFork() {  leftFork.release();  rightFork.release();  System.out.println("Philosopher " + (id + 1) + " ate "+amount+"% and"+" released left and right forks");  } |

Tạo một phương thức để ăn có tên eat

|  |
| --- |
| private void eat() throws InterruptedException {  System.out.println("Philosopher " + (id + 1) + " is eating.");  System.out.flush();  do{  eattime=new Random().nextInt(10);  //generate a random value for eat time which is greater than 0  }while (eattime<=0);  if (amount+eattime\*5>100){  eattime=(100-amount)/5;  amount=100;  }else {  amount=amount+eattime\*5;  }  Thread.sleep(eattime);  } |

Giả sử mỗi triết gia ăn 20% khẩu phần ăn của mình trong mỗi lần ăn.

|  |
| --- |
| public class DiningProblem {  private static final int n = 5;    public static void main(String[] args) {  Semaphore[] fork = new Semaphore[n];  for (int i = 0; i < n; i++) {  fork[i] = new Semaphore(1);  }  Philosopher[] philosophers = new Philosopher[n];  for (int i = 0; i < n; i++) {  philosophers[i] = new Philosopher(i, fork[i], fork[(i + 1) % n]);  new Thread(philosophers[i]).start();  }  }  } |

n là số triết học. Tạo một mảng biến semaphore cho mỗi dĩa, và cho chúng ở trạng thái đang chờ được sử dụng (biến semaphore = 1). Sau đó tạo một mảng các triết gia và khởi tạo với các tham số lần lượt là id, dĩa trái và dĩa phải.

## Kết quả thu được:

|  |
| --- |
| Philosopher 1 is thinking.  Philosopher 1 is holding left fork.  Philosopher 1 is holding right fork.  Philosopher 1 is eating.  Philosopher 1 ate 40% and released left and right forks  Philosopher 1 is thinking.  Philosopher 2 is thinking.  Philosopher 1 is holding left fork.  Philosopher 1 is holding right fork.  Philosopher 1 is eating.  Philosopher 1 ate 45% and released left and right forks  Philosopher 1 is thinking.  Philosopher 1 is holding left fork.  Philosopher 1 is holding right fork.  Philosopher 1 is eating.  Philosopher 3 is thinking.  Philosopher 2 is waiting for left fork  Philosopher 3 is holding left fork.  Philosopher 3 is holding right fork.  Philosopher 3 is eating.  Philosopher 2 is holding left fork.  Philosopher 1 ate 85% and released left and right forks  Philosopher 1 is thinking.  Philosopher 2 is waiting for right fork  Philosopher 4 is thinking.  Philosopher 1 is holding left fork.  Philosopher 2 is holding right fork.  Philosopher 3 ate 30% and released left and right forks  Philosopher 2 is eating.  Philosopher 1 is waiting for right fork... |

# KẾT LUẬN

# TÀI LIỆU THAM KHẢO

[1] <https://github.com/EANimesha/Dinning-Philosopher-Problem>

[2] https://www.baeldung.com/java-dining-philoshophers