**Bài 1.1**

**Khái niệm DeadLock:** là 1 khái niệm chỉ trạng thái mà mỗi nhân tố đợi 1 nhân tố khác trong nhóm bao gồm chính bản thân nó tiến hành thực thi hoặc ra hiệu thường là để giải phóng 1 “khóa”. Đây là một vấn đề phổ biến trong thiết kế xây dựng hệ thông đa luồng, tính toán lập trình song song, hệ phân tán… những lĩnh vực mà khi ấy tài nguyên của phần cứng hay phần mềm sử dụng khái niệm “khóa” để từ đó cấp phát, quản lí tài nguyên chung của hệ thống và triển khai quá trình đồng bộ hóa.

**Có 4 điều kiện cần thiết để sinh ra DeadLock:**

1. **Điều kiện loại trừ lẫn nhau: 1 tài nguyên không thể được sử dụng bởi nhiều hơn 1 tiến trình tại 1 thời điểm.**
2. **Điều kiện giữ và chờ (hay chiếm hữu tài nguyên): các tiến trình giữ tài nguyên và chờ tài nguyên mới (cần thiết để tiến hành xử lí) được giữ bởi một tiến trình khác.**
3. **Điều kiện không thể chiếm: Một khi tiến trình đã chiếm tài nguyên thì không thể bị đòi lại trừ phi tiến trình kết thúc và tài nguyên đó được giải phóng.**
4. **Điều kiện chu trình chờ: các tiến trình giữ tài nguyên và chờ tài nguyên bị chiếm giữ bởi tiến trình khác tạo thành 1 chu trình.**

**Ví dụ về trường hợp sinh DeadLock:**

A screenshot of a cell phone

Description automatically generated

**Kết quả chạy code :**

A close up of a logo

Description automatically generated

Ta tạo 2 lock và tương ứng là 2 thread để xử lí chúng. Với thread 1 nó dùng synchronized để chiếm lấy lock2. Sau đó tạm thời sleep thread này để thread 2 tương ứng chiếm lock1 và sau đó sleep 10ms để chuyển sang thực hiện thread 1. Khi này thread 1 yêu cầu sử dụng lock 1 nhưng rõ ràng lúc này lock1 đang bị thread 2 chiếm giữ. Do đó nó đợi thread 2 nhả lock1 ra nhưng cùng lúc đó thread 2 cũng đang yêu cầu lock2 – đang bị thread 1 chiếm. => Xảy ra tình trạng 2 thread đợi nhau nhả lock để thực hiện tiếp (DeadLock)

**Để sửa tránh hiện tượng DeadLock trên: Có 2 cách**

**C1:** Như hình dưới, ta tạm thời để thread 2 sleep trong 1s (trong lúc đó thread 1 sẽ hoàn thành hết nhiệm vụ của nó ). Sau khi thread 1 hoàn thành nhiệm vụ thì thread 2 tỉnh dậy (sau 1s) để thực hiện tác vụ => Không có DeadLock

A screenshot of a cell phone screen with text

Description automatically generated

**C2:** Để thứ tự truy cập của thread 2 là lock2 và lock1 (giống với thứ tự của thread 1) . Do đó hiện tượng cũng tương tự như C1. Khi thread 2 phải đợi thread 1 xong hết nhiệm vụ với từng lock 1 thì mới được truy cập vào lock đó tương ứng.

**Bài 1.2**

**Bài toán Dinning Philosophers (Triết gia ăn tối):** Một bàn ăn cho 5 người được bố trí 5 cái dĩa, người ăn chỉ có thể ăn khi có đủ trong tay 2 chiếc dĩa. (2 chiếc dĩa này chỉ có thể lấy ở vị trị kế bên phải và trái của người ăn.) Mỗi người ăn có 2 hành động là ĂN và NGHĨ (trong lúc nghĩ thì phải bỏ hết dĩa xuống). Bài toán đặt ra là phải xây dựng được 1 thuật toán mà đảm bảo KHÔNG AI TRONG SỐ 5 NGƯỜI KHÔNG ĐƯỢC ĂN. Giả sử như không ai trong số họ có khả năng biết trước được những người còn lại sẽ ăn hay nghĩ tiếp theo.



Bài toán được đưa ra như 1 ví dụ để xây dựng thuật toán tránh trạng thái DeadLock – trạng thái mà mọi tiến trình bị ngưng trệ vô thời hạn (tương ứng với 1 vd khi mỗi triết gia đều cầm trong tay 1 chiếc dĩa và đợi người kế cần buông dĩa để ăn nhưng tất nhiên là không ai buông cả) hoặc một tiến trình nào đó phải chờ để được xử lí vô thời hạn . Vấn đề tránh DeadLock rất phổ biến trong xây dựng hệ thống khi hệ thống phải đảm đương nhiều luồng thực hiện và mỗi luồng thực hiện sẽ chiếm giữ 1 phần tài nguyên của hệ thống (vd: không gian bộ nhớ, CPU thực hiện tính toán, xử lí…)

**Giải pháp: Quá trình chạy có thể miêu tả thông qua giả code sau**

While(true){

Think();

Pick\_up\_left\_fork();

Pick\_up\_right\_fork();

Eat();

Put\_down\_right\_fork();

Put\_down\_left\_fork();

}

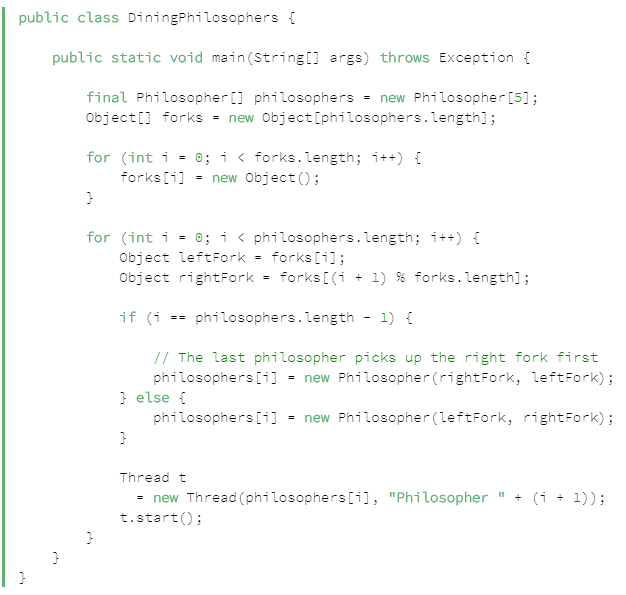
**Xây dựng trên JAVA:**

**B1: Xây dựng class Philosopher: là thread tương ứng**

A screenshot of a cell phone

Description automatically generated

**B2: Tiến hành chạy các thread đã tạo ở trên**



Để tránh trường hợp DeadLock như đã đề cập ở trên, (khi mỗi triết gia đều đồng thời cầm 1 dĩa ở bên tay trái, và đều đợi triết gia bên cạnh nhả dĩa ra). Do đó ở code trên ta đã quy định 1 luật nhằm yêu cầu triết gia cuối cùng thì phải nhặt dĩa bên phải trước ( trong khi bình thường họ sẽ ưu tiên nhặt dĩa trái trước ). Từ đó tránh được trạng thái DeadLock.

**Kết quả chạy:**

**A screenshot of text

Description automatically generated**

**Bài 1.3**

**Khái niệm về Thread Pool trong Java:** Thread Pool là nơi chưa các luồng thực thi của chương trình (Thread). Nó có khả năng tái sử dụng lại những luồng đã được tạo để thực hiện các tasks hiện tại và đưa ra giải pháp cho vấn đề vòng vận hành luồng cũng như phân bổ tài nguyên. Bởi vì khi thread đã và đang tồn tại khi yêu cầu (request) được đưa tới nên hạn chế được sự chậm trễ trong việc tạo luồng xử lí yêu cầu đó, khiến cho ứng dụng có khả năng đáp ứng cao. ( 1 vd có tính chất khá giống với **Thread Pool** đó chính là **String Pool** ).

**JAVA** cung cấp framework Executor tập trung xung quanh interface Executor, và sub-interface ExecutorService, trong đó ThreadPoolExecutor thì nó implement cả 2 interface ở trên. Bằng cách sử dụng executor ta có thể tiến hành xử lí các Runnable objects (các objects tương ứng với các luồng)

A screenshot of a cell phone

Description automatically generated

Sơ đồ bên trên miêu tả quá trình xử lí các tác vụ (Task) thông qua 3 luồng được tạo sẵn trong Thread Pool. Các task được đưa vào hang đợi (Task Queue). Trong đó Thread Pool có các thông số quan trọng , lấy vd như với class ThreadPoolExecutor bao gồm corePoolSize(số lượng thread tối đa được tạo ra khi queue chưa đầy) và maxPoolSize (số lượng thread tối đa của Thread Pool khi mà queue đã đầy)

**Quá trình hoạt động trải qua 4 giai đoạn:**

1. **Nếu số thread hiện tại nhỏ hơn so với corePoolSize thì sẽ tạo tiếp thread mới để chạy 1 task mới được đưa vào (hiện tại lúc này queue “chưa có gì cả”)**
2. **Khi mà số thread hiện tại bằng (hoặc lớn hơn) corePoolSize thì bắt đầu đẩy các task mới vào hang chờ queue.**
3. **Khi hang chờ đã đầy thì tiến hành tạo thêm các thread mới (miễn là khi số lượng thread hiện tại vẫn nhỏ hơn maxPoolSize) để xử lí các task mới được đưa vào**
4. **Khi mà queue bị đầy, và số lượng thread cũng bằng maxPoolSize thì sẽ tiến hành từ chối các task (reject task).**