

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

**TRƯỜNG ĐIỆN – ĐIỆN TỬ**

**HỆ ĐIỀU HÀNH**

**Đề tài: Deadlock – CPU Affinity**

**GVHD:** TS Hàn Huy Dũng

**Sinh viên thực hiện:**

|  |  |  |
| --- | --- | --- |
| Phạm Trọng Khang | 20192929 |  |
| Hoàng Văn Huy | 20192909 |  |
| Nguyễn Văn Huân | 20192879 |  |
| Nguyễn Văn Dương | 20192796 |  |
|  |  |  |

Hà Nội, 3-2023

# **DANH MỤC BẢNG BIỂU**

Bảng 1.1: Bảng kết quả đo thời gian chạy của deadlock và số thread………………17

Bảng 1.2: Bảng kết quả đo thời gian chạy của deadlock và số tài nguyên………….19

Bảng 2.1: Kết quả đo thời gian ngủ……………………………………………………21

Bảng 2.3: Kết quả đo với sleep\_time khác nhau………………………………………..32

# **Danh mục hình ảnh**

Hình 1.1: Ví dụ mình họa về Deadlock………………………………………………………….…6

Hình 2.1: Minh họa bài toán của các triết gia…………………………………………………....14

Hình 2.2: Biểu đồ thể hiện kết quả sự phụ thuộc của deadlock và số thread……………………18

Hình 2.3: Biểu đồ thể hiện kết quả sự phụ thuộc của deadlock và số tài nguyên……………….19

Hình 2.4: Sơ đồ khối giải pháp deadlock………………………………………………………..21

Hình 2.5: Kết quả chạy khi sleep\_seconds = 0 (s)………………………………………………22

Hình 2.6: Kết quả chạy khi sleep\_seconds = 6 (s)………………………………………………23

Hình 2.7: Kết quả chạy khi sleep\_seconds = 10 (s)……………………………………………..24

Hình 2.8: Kết quả chạy khi sleep\_seconds = 15 (s)……………………………………………..25

Hình 2.9: Sơ đồ thuật toán phân cấp tài nguyên…………………………………………………27

Hình 2.10: Kết quả chạy lệnh taskset -c trên 4 CPU…………………………………………….29

Hình 2.11: Kết quả chạy lệnh taskset -c trên 3 CPU…………………………………………….30

Hình 2.12: Kết quả chạy lệnh taskset -c trên 2 CPU…………………………………………….31

Hình 2.13: Kết quả chạy lệnh taskset -c trên 1 CPU…………………………………………….32

Mục lục

[**DANH MỤC BẢNG BIỂU** 2](#_Toc130225762)

[**Danh mục hình ảnh** 3](#_Toc130225763)

[**1.** **Những thứ mà nhóm em sẽ làm.** 5](#_Toc130225764)

[**2.** **Tìm hiểu Deadlock** 5](#_Toc130225765)

[**a.** **Deadlock là gì?** 5](#_Toc130225766)

[**b.** **Mô hình hệ thống.** 6](#_Toc130225767)

[**c.** **Điều kiện tồn tại Deadlock.** 6](#_Toc130225768)

[**d.** **Các phương án xử lý vấn đề deadlock.** 7](#_Toc130225769)

[**3.** **Một số trường hợp cụ thể của deadlock.** 8](#_Toc130225770)

[**4.** **Giải thuật phát hiện và phòng tránh deadlock.** 14](#_Toc130225771)

[**4.1** **Bài toán Dining Philosopher.** 14](#_Toc130225772)

[***a.*** ***Thay đổi số thread.*** 17](#_Toc130225773)

[***b.*** ***Thay đổi số tài nguyên.*** 18](#_Toc130225774)

[**4.2** **Phương pháp giải quyết bài toán.** 20](#_Toc130225775)

[**a.** ***Cho một tiến trình ngủ***. 20](#_Toc130225776)

[***b.*** ***Phương pháp phân cấp tài nguyên.*** 25](#_Toc130225777)

[**4.3** ***Áp dụng kỹ thuật Processor Affinity vào bài toán Dining Philosopher*.** 28](#_Toc130225778)

[**5.** **Kết luận** 33](#_Toc130225779)

[**6.** **Tài liệu tham khảo** 33](#_Toc130225780)

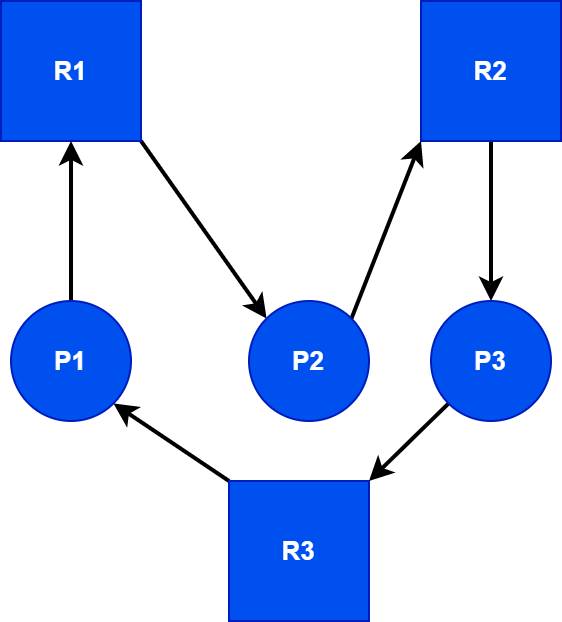
# **Những thứ mà nhóm em sẽ làm.**

* Giới thiệu khái quát về deadlock và lấy mốt số ví dụ dễ để minh họa cho deadlock.
* Bài toán Dining Philosopher là một ví dụ điển hình của một bài toán tổng quát về deadlock cho một số tiến trình cần sử dụng tài nguyên và có thể tranh chấp dẫn đến gây ra deadlock. Khảo sát sự phụ thuộc của deadlock và phương pháp giải quyết để tránh gây ra deadlock.
* Bên cạnh đó chúng em sẽ dùng 2 giải pháp để giải quyết bài toán trên, đó là: đưa 1 tiến trình vào trạng thái ngủ và giải pháp phân cấp tài nguyên.
* Qua sự góp ý của thầy nhóm em sử dụng thêm kỹ thuật ***Processor Affinity*** vào để đo thời gian thực thi của bài toán.

# **Tìm hiểu Deadlock**

## **Deadlock là gì?**

* **Deadlock là tình trạng thường xuyên xảy ra trên mọi thiết bị máy tính hay hệ thống phân tán có các quy trình hoạt động tài nguyên**, cụ thể đó là hai quá trình đang ở trong trạng thái chờ tài nguyên đã được tổ chức sẵn. Ta có thể hiểu là trong môi trường multiprogramming 1 số process có thể tranh nhau 1 số tài nguyên hạn chế. Một process yêu cầu các tài nguyên, tại thời điểm đó tài nguyên không thể đáp ứng thì process sẽ chuyển sang trạng thái chờ. Các process chờ có thể sẽ ko bao giờ thay đổi lại trạng thái được vì các tài nguyên mà nó yêu cầu bị giữ bởi các process khác.
* Ta lấy một ví dụ minh họa dễ hiểu như sau:



**Hình 1.1: Ví dụ mình họa về Deadlock**

Với ba quá trình P1, P2, P3 và ba tài nguyên R1, R2, R3:

Nếu P1 yêu cầu R2 của P2 thì cần phải chờ cho P2 giải phóng tài nguyên và tạo ra R3. Tương tự như vậy đối với P1 và P3. Vì thế mà nó thường xuyên xảy ra các tình trạng Deadlock trong hệ điều hành.

## **Mô hình hệ thống.**

* Trong một hệ thống , các tiến trình từ khi được gọi đến khi kết thúc sẽ qua các giai đoạn sau :
* Yêu cầu tài nguyên (request): nếu yêu cầu không được giải quyết ngay (ví dụ khi tài nguyên đang được tiến trình khác sử dụng) thì tiến trình yêu cầu phải đợi cho đến khi nhận được tài nguyên.
* Sử dụng tài nguyên (use).
* Giải phóng tài nguyên (release).

## **Điều kiện tồn tại Deadlock.**

* Bốn điều kiện cơ bản nhất của Deadlock là:
* Điều kiện loại trừ lẫn nhau (*Mutual Exclusion*): Các Tài nguyên loại trừ lẫn nhau được chia sẻ chẳng hạn như tệp chỉ đọc không dẫn đến bế tắc nhưng các tài nguyên, chẳng hạn như máy in và ổ đĩa băng, yêu cầu quyền truy cập độc quyền bởi một quy trình duy nhất.
* Điều kiện giữ và chờ tài nguyên mới (*Hold and Wait*): Giữ và chờ Trong điều kiện này, các quy trình phải được ngăn chặn giữ một hoặc nhiều tài nguyên trong khi đồng thời chờ một hoặc nhiều tài nguyên khác.
* Điều kiện không có trưng dụng tài nguyên (*No Preemption*): Không có sự ưu tiên Việc lựa chọn phân bổ tài nguyên quy trình có thể tránh được tình trạng bế tắc, nếu có thể
* Điều kiện chờ đợi vòng tròn (*Circular Wait*): Có thể tránh được thời gian chờ theo vòng tròn nếu chúng ta đánh số tất cả các tài nguyên và yêu cầu các quy trình chỉ yêu cầu tài nguyên theo thứ tự tăng (hoặc giảm) nghiêm ngặt.

## **Các phương án xử lý vấn đề deadlock.**

Một tiến trình trong hệ điều hành sử dụng các tài nguyên khác nhau và sử dụng tài nguyên theo cách sau:

• Yêu cầu tài nguyên.

• Sử dụng tài nguyên.

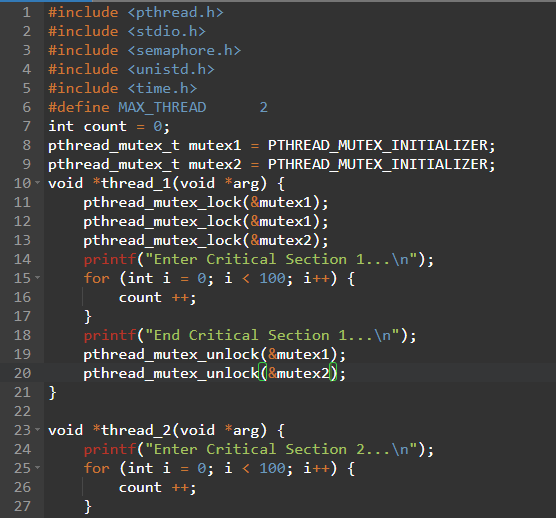
• Giải phóng tài nguyên.

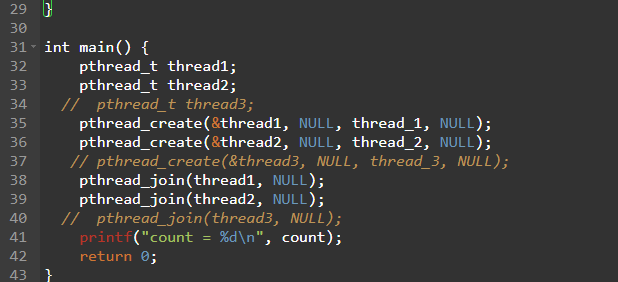
Nói chung, có ba cách để xử lý các bế tắc:

* **Ngăn chặn hoặc tránh deadlock** - Không để hệ thống rơi vào trạng thái deadlock, chúng ta có thể lấy một nguồn lực từ một quá trình và đưa nó cho quá trình khác. Điều này sẽ giải quyết tình trạng bế tắc, nhưng đôi khi nó gây ra vấn đề.
* **Phát hiện và khôi phục deadlock** - Hủy bỏ một quy trình hoặc xử lý trước một số tài nguyên khi phát hiện thấy các deadlock. Ta sẽ loại bỏ một hoặc nhiều quy trình, đây là cách đơn giản nhất nhưng nó rất dễ thực hiện tuy nhiên đây không phải phương pháp tối ưu cho việc chúng ta muốn khắc phục lỗi.
* **Bỏ qua tất cả vấn đề** - Nếu deadlock chỉ xảy ra mỗi năm một lần hoặc lâu hơn, có thể tốt hơn là chỉ để chúng xảy ra và khởi động lại nếu cần thiết hơn là phải chịu các hình phạt liên tục về hiệu suất hệ thống và chi phí liên quan đến việc phát hiện hoặc ngăn chặn deadlock.

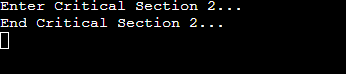
# **Một số trường hợp cụ thể của deadlock.**

* Deadlock có thể xảy ra khi việc bạn sử dụng tài nguyên không đúng thứ tự hoặc không hợp lý.
* TH1: tài nguyên 1 bị khóa 2 lần liên tiếp

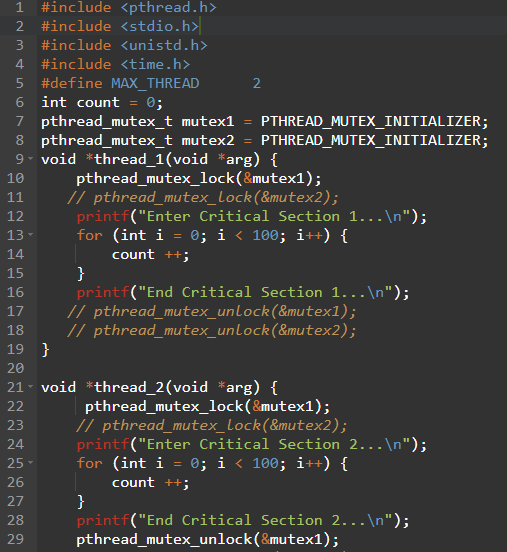
****

****

**Kết quả:** chỉ chạy được thread2 và không dừng được.



* TH2: thread1 sử dụng song tài nguyên không giải phóng dẫn đến thread2 vần phải đợi tài nguyên được giải phóng gây lên deadlock.

Text

Description automatically generated

**Kết quả:** Sau khi chạy song thread1 thì xảy ra deadlock.

Text

Description automatically generated

* TH3: Trong đoạn mã này, cả hai luồng đều muốn giữ cả lockA và lockB, nhưng chỉ có một luồng có thể giữ cả hai lock đó tại một thời điểm. Vì vậy, khi một luồng giữ lockA và thứ hai giữ lockB, cả hai luồng sẽ chờ nhau để giải phóng lock, dẫn đến deadlock.

Text

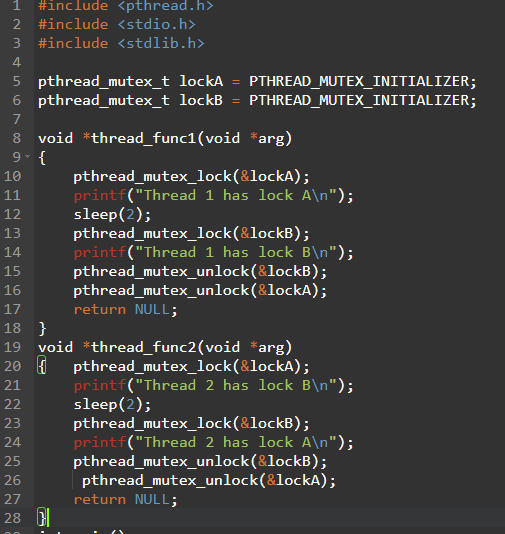
Description automatically generatedText

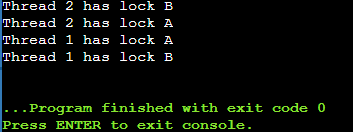
Description automatically generated

**Kết quả:**

**Text

Description automatically generated**

* Để không bị deadlock ta phải giữ các mutex theo một thứ tự nhất định.
* 
* Kết quả:



# **Giải thuật phát hiện và phòng tránh deadlock.**

## **Bài toán Dining Philosopher.**

Bài toán Dining Philosopher là một ví dụ điển hình của một bái toán tổng quát về deadlock cho một số tiến trình cần sử dụng tài nguyên và có thể tranh chấp dẫn đến gây ra deadlock.

*Giới thiệu bài toán*:

* Cho 5 triết gia ngồi chung một bàn tròn với 5 chiếc đũa xếp xem kẽ giữa 2 người ngồi cạnh nhau như trong hình:



**Hình 2.1: Minh họa bài toán của các triết gia**

* Mỗi triết gia tìm cách để ăn được thức ăn từ đĩa của mình với điều kiện: “chỉ ai có 2 chiếc đũa cạnh mình mới được phép ăn”, do đó họ lần lượt đổi trạng thái giữa ăn (eating) và đợi (thinking)) Mỗi người sau khi giữa đôi đũa để ăn sau 1 khoảng thời gian phải bỏ lại 2 chiếc đũa về vị trí cũ để tiếp tục quá trình này. Yêu cầu: tìm một phương pháp đảm bảo để các triết gia đều có thể được ăn / đợi đổi lượt để không ai bị chết đói (chỉ đợi chứ không được ăn).
* Với bài toán trên ta có các dữ liệu sau:
* Số triết học gia tương đương với số thread.
* Số thức ăn tương đương với số tài nguyên.
* Các phần sau sẽ có thêm thời gian ngủ (sleep).

*Các trường hợp xảy ra:*

* Trường hợp 1: lần lượt từng người lấy đũa, tiến hành ăn, ăn xong, đặt đũa xuống sau đó tới người tiếp theo. Như vậy sẽ không có bế tắc nào xảy ra ở đây.
* Trường hợp thứ 2: tất cả các triết gia đều cầm đũa của mình và chờ đũa từ người bên cạnh để có thể ăn:

Triết gia 0 đang cầm chiếc đũa thứ 0 và đợi chiếc đũa thứ 1.

Triết gia 1 đang cầm chiếc đũa thứ 1 và đợi chiếc đũa thứ 2.

Triết gia 2 đang cầm chiếc đũa thứ 2 và đợi chiếc đũa thứ 3.

Triết gia 3 đang cầm chiếc đũa thứ 3 và đợi chiếc đũa thứ 4.

Triết gia 4 đang cầm chiếc đũa thứ 4 và đợi chiếc đũa thứ 0.

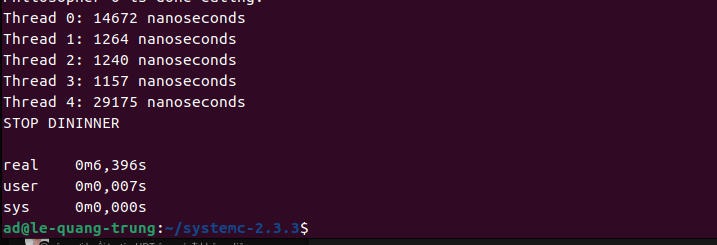
Trong trường hợp này, mỗi triết gia chỉ cầm 1 chiếc đũa trong cùng một thời điểm, do đó không có triết gia nào có thể thực hiện hoạt động do không thể thỏa mãn yêu cầu có hai chiếc đũa, nên dẫn đến trong trường hợp này hiện tượng Deaadlock xảy ra.

Mô phỏng: ta bắt đầu với 5 nhà triết gia và với 50 phần ăn. Như ta thấy có 2 kết quả cho bài toán trên, viêc có xảy ra deadlock hay ko phụ thuộc vào từng trường hợp và từng máy khác nhau.

Text

Description automatically generatedText

Description automatically generated

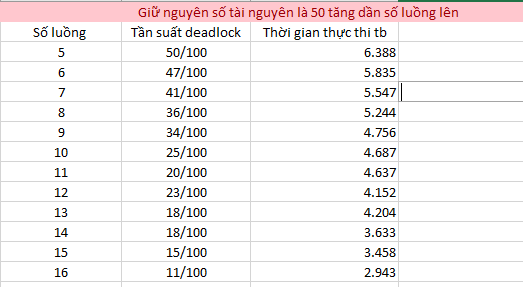


### ***Thay đổi số thread.***

Ta tăng dần số lượng thread lên và giữ nguyên số tài nguyên để kiểm tra chương trình từ đó thống kê kết quả sau:

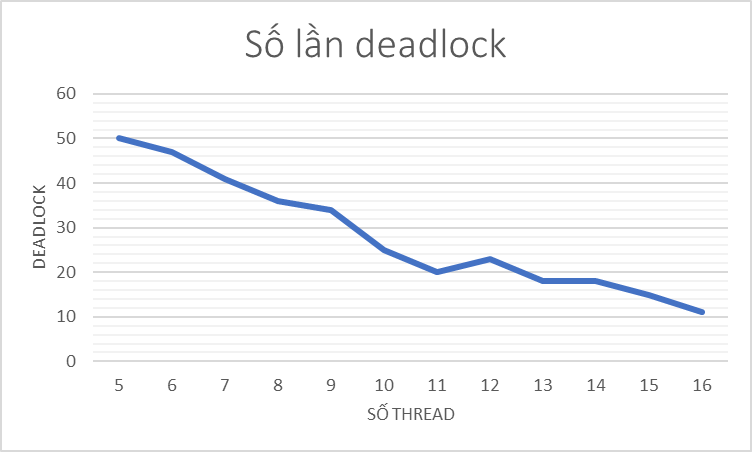
Giữ nguyên số lượng tài nguyên là 50 và tăng số thread lên.

|  |  |  |
| --- | --- | --- |
| **Số thread** | **Số lần deadlock** | **Thời gian thực thi** |
| 5 | 4/10 | 6.239s |
| 6 | 3/10 | 6.105s |
| 7 | 1/10 | 5.252s |
| 8 | 1/10 | 4.907s |
| 9 | 1/10 | 4.312s |
| 10 | 1/20 | 4.144s |



**Bảng 1.1: Bảng kết quả đo thời gian chạy của deadlock và số thread**

**Hình 2.2: Biểu đồ thể hiện kết quả sự phụ thuộc của deadlock và số thread.**

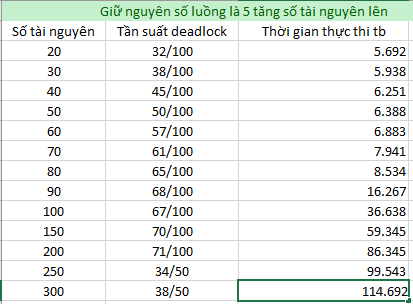


### ***Thay đổi số tài nguyên.***

Tăng số lượng tài nguyên lên và giữ nguyên số luồng.

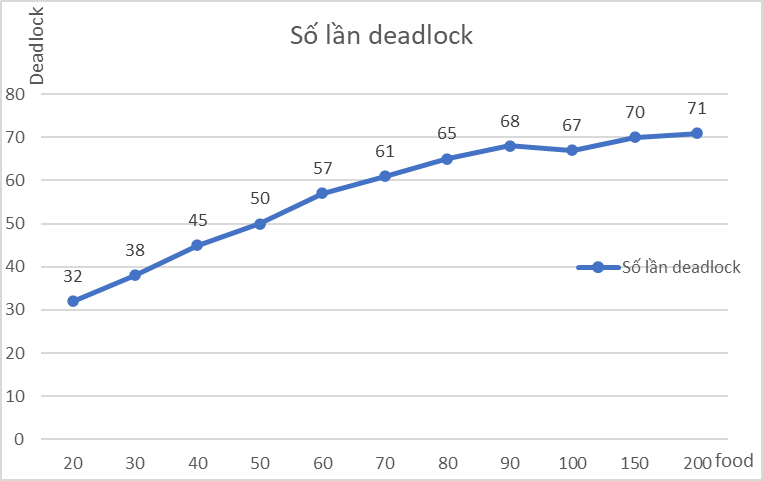
Ta giữ nguyên số luồng là 5 và tăng dần số lượng tài nguyên lên

|  |  |  |
| --- | --- | --- |
| **Số lượng thức ăn** | **Số lần deadlock** | **Thời gian thực thi TB** |
| 50 | 4/10 | 6.239s |
| 80 | 5/10 | 8.534s |
| 100 | 5/10 | 25.244s |
| 150 | 7/10 | 56.628s |



**Bảng 1.2: Bảng kết quả đo thời gian chạy của deadlock và số tài nguyên**

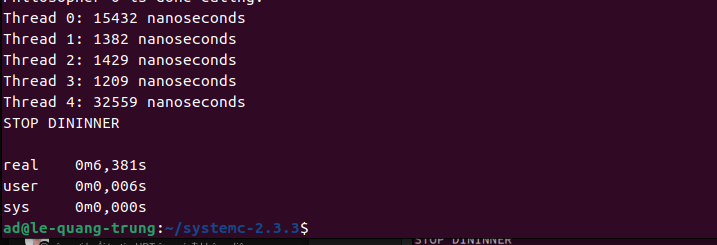
**Hình 2.3: Biểu đồ thể hiện kết quả sự phụ thuộc của deadlock và số tài nguyên (số food).**



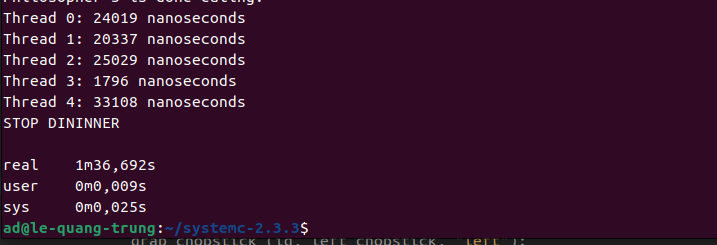
Số lần deadlock phụ thuộc số tài nguyên( thứ ăn).

* Kết quả một số lần chạy thử:

Food = 50.



Food = 200.



Khi cho Food = 1000 thì chạy được kết quả với thời gian thực thi khá dài.

Text

Description automatically generated with medium confidence

* Nhận xét:
* Khi tăng số tài nguyên (số food) lên thì ta thấy thời gian thực thi của chương trình cũng sẽ tăng lên và số lần bị deadlock cũng sẽ tăng lên . Từ đó ta thấy thời gian thực thi của chương trình và số lần bị deadlock có phụ thuộc vào số tài nguyên (số food).

### **Phương pháp giải quyết bài toán.**

1. ***Cho một tiến trình ngủ***: Chọn 1 ông triết gia với id bất kỳ, cho ông ta ngủ trong một khoảng thời gian sleep\_time đủ lớn, để khi tất cả những ông triết gia còn lại thực hiện xong hành động của mình thì ông triết gia được chọn mới tỉnh dậy là thực hiện nốt hành động của mình. Nhờ có thời gian ngủ sleep\_time này thì hiện tượng xảy ra deadlock không còn xảy ra. Tuy nhiên, thời gian ngủ là bao nhiêu là đủ cho bài toán 5 ông triết gia.

* Với Thread = 5 và số tài nguyên là 50
* Sơ đồ thuật toán:

Diagram

Description automatically generated

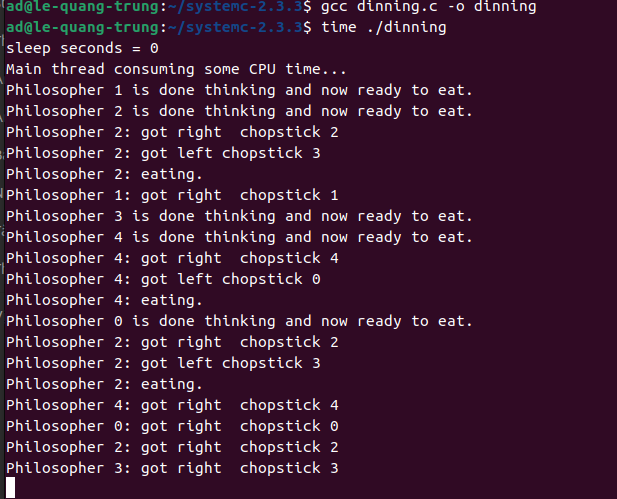
**Hình 2.4: Sơ đồ khối giải pháp deadlock**

* Kết quả mô phỏng:

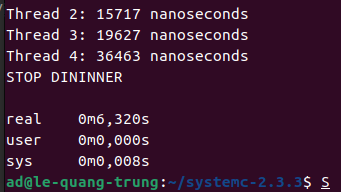
|  |  |
| --- | --- |
| **Thời gian thực thi trung bình(s)** | **Thời gian ngủ (s).** |
| 6.439 | 0 |
| 6.383 | 6 |
| 10.014 | 10 |
| 15.013 | 15 |

**Bảng 2.1 Kết quả đo thời gian ngủ**

* ***Sleep\_seconds = 0 (s).***

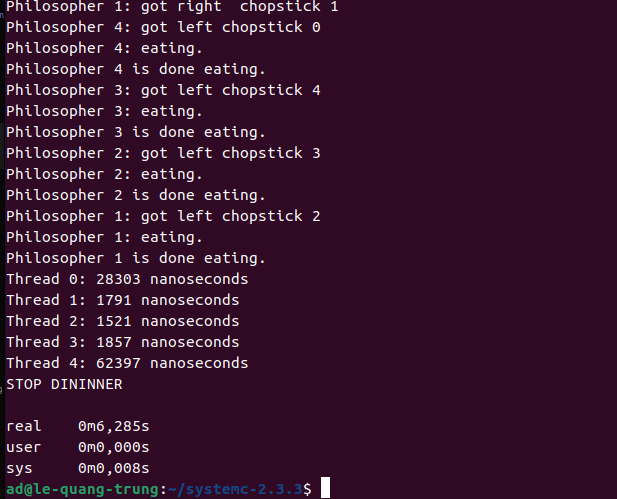
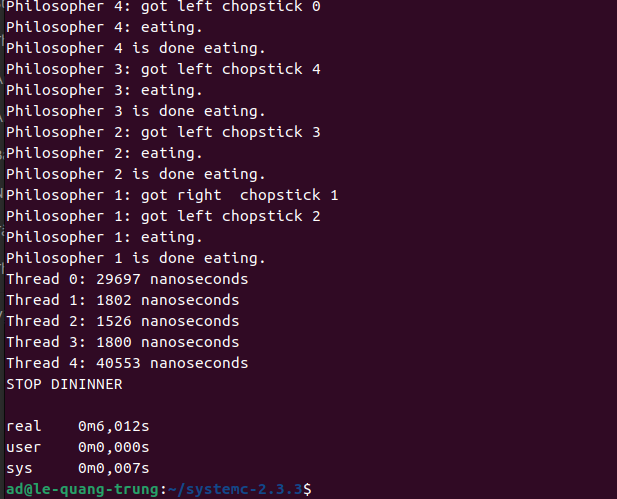


Bị deadlock



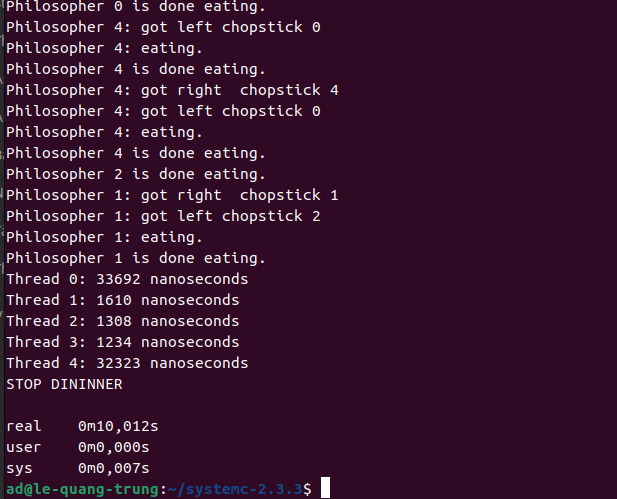
**Hình 2.5: Kết quả chạy khi sleep\_seconds = 0 (s)**

* ***Sleep\_seconds = 6 (s).***



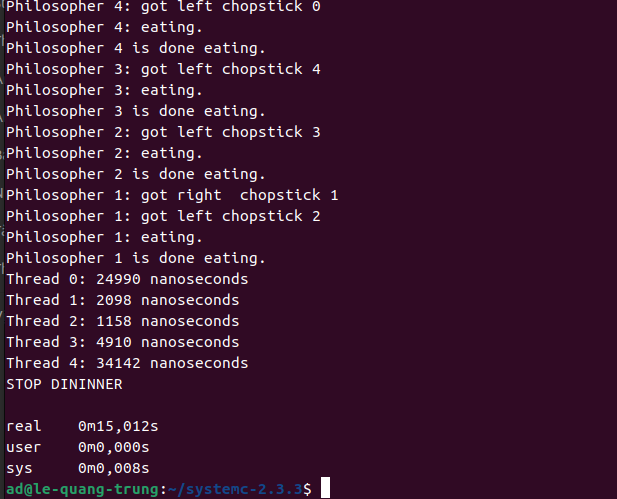
**Hình 2.6: Kết quả chạy khi sleep\_seconds = 6 (s)**

* ***Sleep\_seconds = 10 (s).***



**Hình 2.7: Kết quả chạy khi sleep\_seconds = 10 (s)**

* ***Sleep\_seconds = 15 (s).***



**Hình 2.8: Kết quả chạy khi sleep\_seconds = 15 (s)**

* Nhận xét:

Từ các kết quả đo trên, ta có thể rút ra được một số nhận xét như sau:

• Khi sleep\_seconds = 0 (tức là philo #1 không sleep) xảy ra deadlock (vẫn có trường hợp không xảy ra deadlock)

• Việc deadlock không xảy ra nếu thời gian sleep đủ lớn , tuy nhiên thời gian thực thi bằng khoảng thời gian sleep.

• Việc xảy ra deadlock là ngẫu nhiên, nó phụ thuộc vào nhiều yếu tố.

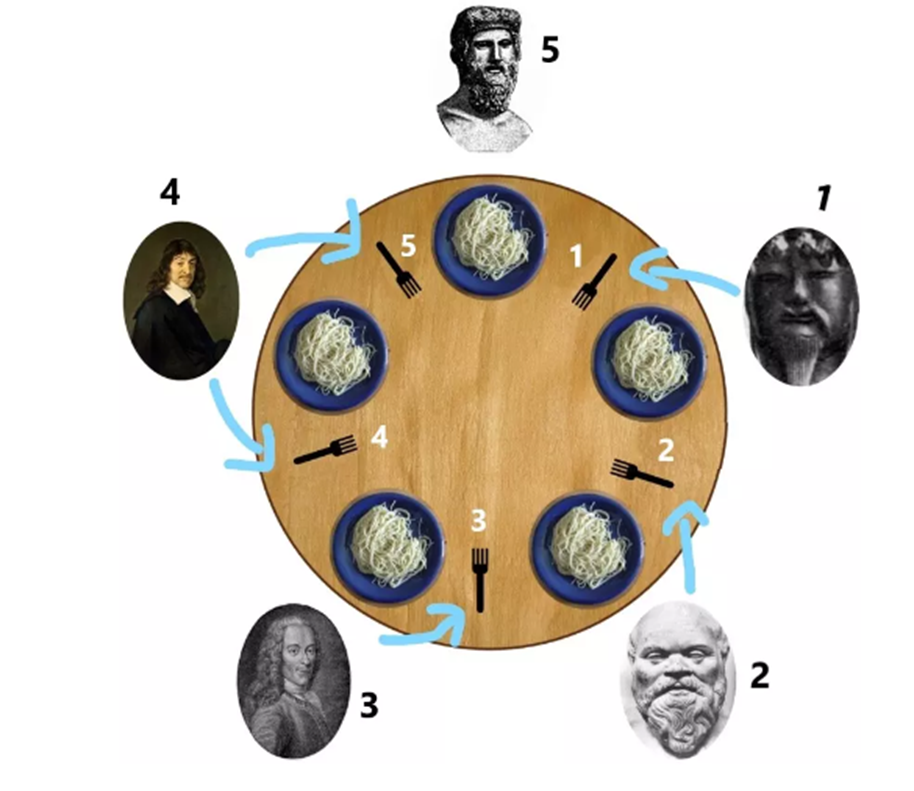
* Như vậy, deadlock xảy ra phụ thuộc vào :

1. Thời gian ngủ của Philosopher

2. Cơ chế lập lịch của các thread trong multithreading.

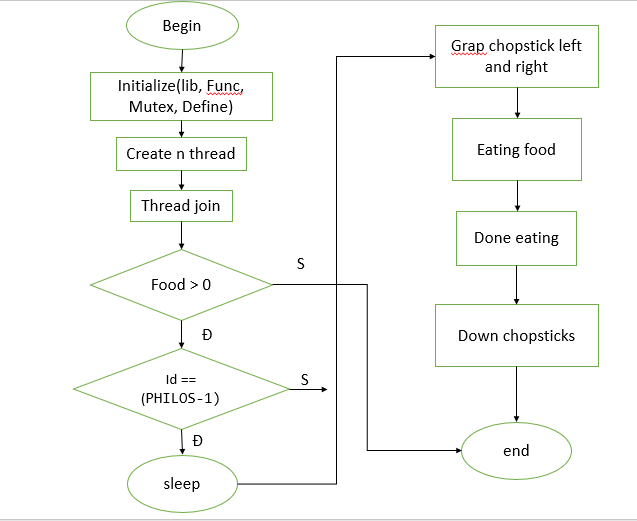
3. Thời gian thực thi của các thread không đồng đều ở các thời điểm giống nhau từ đó suy ra việc xảy ra deadlock là ngẫu nhiên.

1. ***Phương pháp phân cấp tài nguyên.***

******

Mỗi chiếc dĩa sẽ được đánh số từ 1 - 5, và mỗi triết gia sẽ luôn ưu tiên giành lấy chiếc dĩa có số nhỏ nhất trong 2 chiếc cần giành (Triết gia 1 lấy chiếc dĩa 1, triết gia 2 lấy chiếc dĩa 2, …). Như vậy sau khi 4 vị triết gia đã lấy chiếc dĩa của mình, triết gia số 5 sẽ không thể lấy chiếc dĩa nào (vì chiếc dĩa nhỏ nhất số 1 bên cạnh ông đã bị lấy). Nhờ đó, vị triết gia số 4 có thể lấy chiếc dĩa số 5 và bắt đầu ăn.

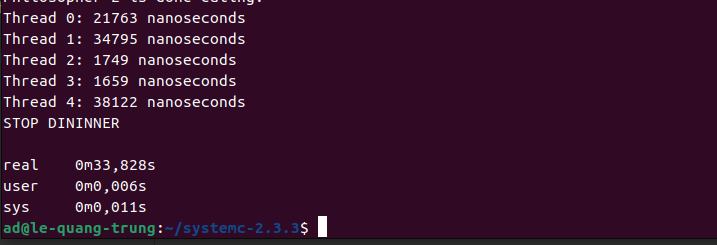
* Sơ đồ giải thuật thuật toán phân cấp tài nguyên.

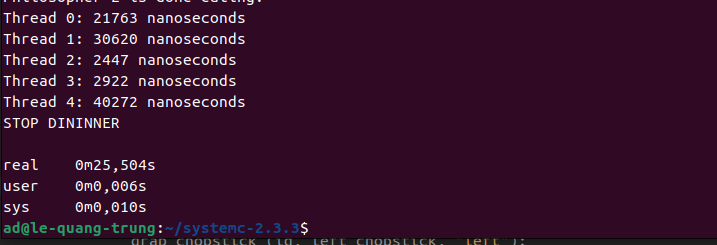


**Hình 2.9: Sơ đồ thuật toán phân cấp tài nguyên**

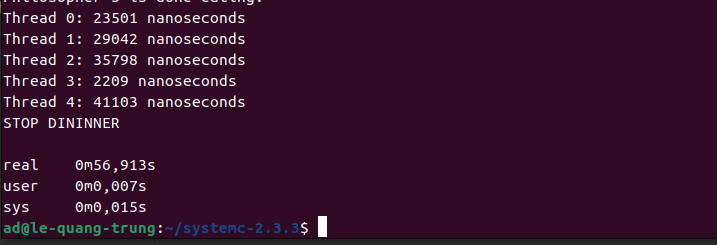
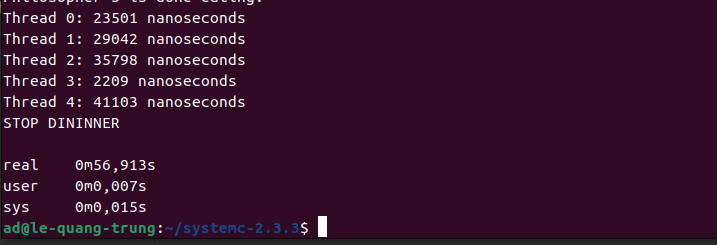
* Kêt quả:

Khi Food =200.





Khi Food =300.



Ta thấy thời gian chạy đã giảm đi nhiều và tỷ lệ bị deadlock cũng giảm đi đáng kể

* 1. ***Áp dụng kỹ thuật Processor Affinity vào bài toán Dining Philosopher*.**
* Sử dụng thước đo thời gian để đánh giá ký thuật và từ đó tìm ra thời gian ngủ thích hợp nhất để ko xảy ra deadlock.

*Ta tìm hiểu Processor Affinity là gì?*

Processor Affinity còn được gọi là CPU pinning, cho phép người dùng chỉ định tiến trình chỉ sử dụng một vài lõi. Về mặt kỹ thuật, bạn có thể ghép nối và hủy ghép nối một tiến trình hoặc luồng với CPU hoặc các CPU (ở đây có thể được gọi là lõi CPU). Processor Affinity hữu ích nếu bạn có một chương trình nặng như kết xuất video. Khi bạn dành riêng một lõi cho chương trình chỉnh sửa video, điều đó đảm bảo rằng lõi của bộ xử lý luôn dành riêng cho tác vụ. Việc này giúp cải thiện hiệu suất vì nó giảm vấn đề cache, do không có độ trễ với lõi chuyên dụng.

Tuy nhiên, điều này cũng có nghĩa là chương trình không thể sử dụng bất kỳ lõi nào khác ảnh hưởng đến cân bằng tải.

Process affinity: là thuộc tính của scheduler giúp liên kết hoặc hủy liên kết các process, khiến các process chỉ chạy trong một số CPU nhất định được phân bổ.

*a. Công cụ Taskset.*

Xử lý đa lõi có thể mang lại nhiều lợi ích và cả những hạn chế. Đôi khi, cần chỉ định thủ công lõi CPU cho các chương trình để đạt được hiệu suất tốt nhất. Trong Linux, công cụ chính cho mục đích này là taskset.“Taskset” là công cụ sử dụng để phân chia các CPU cho process chạy trên nó. Dựa trên pid của một process/task/thread thì taskset có thể phân bổ chúng vào các CPU nhất định.

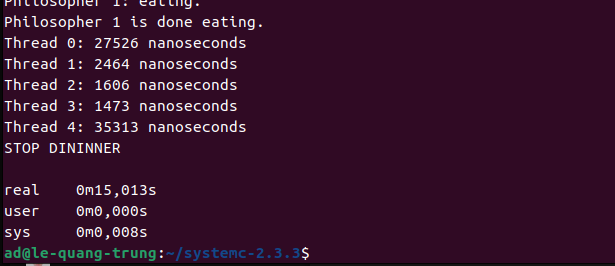
*b. Áp dụng vào bài toán.*

Ở đây, chương trình được chạy với sleep\_time = 15s.

* Với Thread = 5 và số tài nguyên là 50
  + Khi cho chương trình chạy trên 4 CPU (từ 0 đến 3)

hình ảnh

Kết quả:

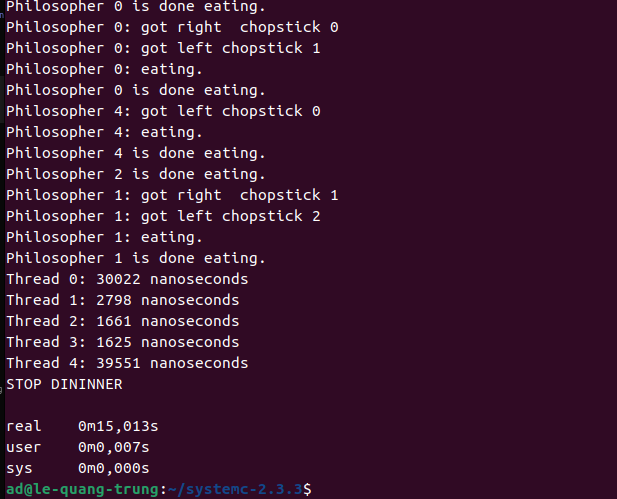


**Hình 2.10: Kết quả chạy lệnh taskset -c trên 4 CPU**

* + Khi cho chương trình chạy trên 3 CPU (từ 0 đến 2)



Kết quả:

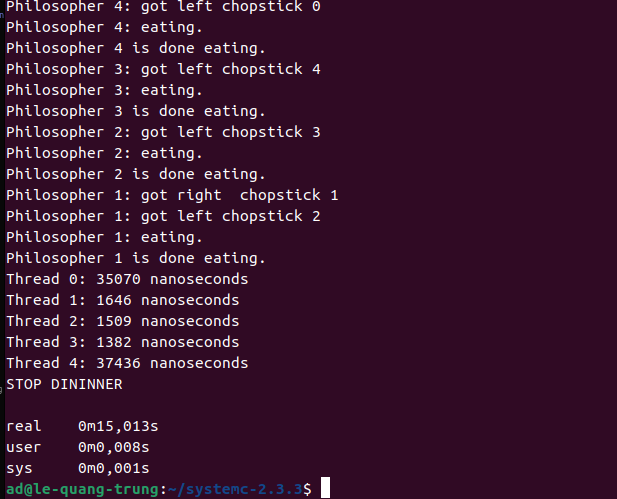


**Hình 2.11: Kết quả chạy lệnh taskset -c trên 3 CPU**

* + Khi cho chương trình chạy trên 2 CPU (từ 0 đến 1)



Kết quả:

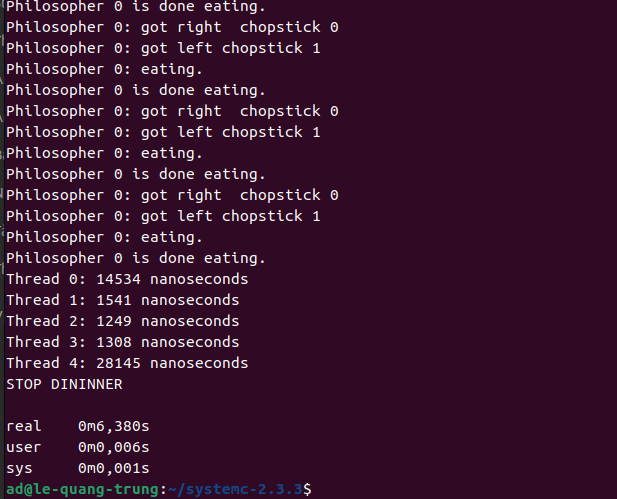


**Hình 2.12: Kết quả chạy lệnh taskset -c trên 2 CPU**

* + Khi cho chương trình chạy trên 1 CPU (là CPU 0)



Kết quả:



**Hình 2.13: Kết quả chạy lệnh taskset -c trên 1 CPU**

Từ các kết quả trên ta có thể thấy được thời gian thực thi của 2 trường hợp là khác nhau khi chạy trên số lượng CPU khác nhau. Khi chạy với 1 CPU thì kết quả chạy ra là trung bình là 6,105 còn khi chạy nhiều hơn với 1 CPU thì kết quả sẽ gần bằng với thời gian ngủ.

Thực hiện đo đạc nhiều lần với sleep khác nhau ta thu được bảng kết quả như sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sleep** | **1 CPU** | **2 CPU** | **3 CPU** | **4 CPU** |
| 0 | 6.38 | 5.946 | 5.845 | 5.635 |
| 6 | 6.381 | 6.138 | 6.282 | 6.285 |
| 10 | 6.385 | 10.013 | 10.012 | 10.012 |
| 15 | 6.383 | 15.013 | 15.012 | 15.012 |

**Bảng 2.2 Kết quả đo với sleep\_time khác nhau**

* **Nhận xét:**

Từ bảng kết quả đo đạc ta đưa ra được kết luận:

• Khi cho chương trình chạy trên cả 4 nhân :

* + Nếu thời gian ngủ của philosopher đủ lớn thì không xảy ra deadlock tuy nhiên thời gian thực thi xấp xỉ bằng thời gian ngủ.

• Khi chia thành nhân riêng biệt để thực thi chương trình, nếu thời gian ngủ đủ lớn:

* + Không xảy ra deadlock.
  + Thời gian thực thi giảm nếu chỉ chạy trong 1 nhân (bé hơn thời gian ngủ).
  + Từ những kết quả thực tế khi đo đạc nhiều lần, nhận thấy thời gian thực thi tối ưu đạt xấp xỉ 6.384s (như vậy khi chọn thời gian ngủ, chúng ta nên lấy sleep tối thiểu là 6.5s).

# **Kết luận**

Qua quá trình tìm hiểu và thực hiện bài toán deadlock – cụ thể là bài toán Dinning philosopher, chúng em đã hiểu rõ hơn về việc sử dụng chương trình đa luồng trong khi dùng chung tài nguyên của hệ điều hành. Kết quả mà nhóm thực hiện được là đã tránh được việc deadlock xảy ra với giải pháp cho triết gia ngủ trong 1 khoảng thời gian đủ lớn. Khoảng thời gian đó được đo lại nhiều lần và cho một kết quả trung bình là 6.4s; phương pháp phân cấp tài nguyên giảm được việc deadlock và thời gian thực thi của nó.

Chúng em cảm ơn thầy hướng dẫn – thầy Hàn Huy Dũng đã tận tụy giúp đỡ chúng em trong quá trình thực hiện bài tập lớn. Sự giúp đỡ cùng với cách hướng dẫn chúng em tìm hiểu và thực hiện một cách khoa học sẽ là cơ sở, nền tảng để chúng em làm việc sau này.

# **6. Tài liệu tham khảo**

[1] [https://docs.oracle.com/cd/E19205-01/820-0619/geosb/index.htm**l**](https://docs.oracle.com/cd/E19205-01/820-0619/geosb/index.html)**.**

[2] <https://www.cs.rpi.edu/academics/courses/fall04/os/c10/>.

[3] <https://linuxhint.com/use-taskset-command/>.

[4] https://viblo.asia/p/go-concurrency-qua-cac-vi-du-phan-1-dining-philosophers-yMnKM0Gz57P#\_31-giai-phap-phan-cap-tai-nguyen-3.

[5] <https://c-program-example.com/2012/02/c-program-to-solve-dining-philosophers-problem.html>.

[6] <https://codecharms.me/posts/operating-system-the-dining-philosophers-problem>.

[7] <https://viblo.asia/p/go-concurrency-qua-cac-vi-du-phan-1-dining-philosophers-yMnKM0Gz57P#_31-giai-phap-phan-cap-tai-nguyen-3>.