TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

ASSIGNMENT 02.02

HỆ ĐIỀU HÀNH

Lớp: 21CLC03

Sinh viên: Trần Nguyên Huân – 21127050

Nguyễn Hoàng Phúc – 21127671

Giảng viên hướng dẫn: Thái Hùng Văn, Đặng Trần Minh Hậu

THÀNH PHỐ HỒ CHÍ MINH - 2023

Mục lục

I. Bảng phân công công việc, mức độ hoàn thành

Họ và tên	MSSV	Công việc thực hiện	Mức độ hoàn thành
Trần Nguyên Huân	21127050	- Nghiên cứu, trình bày phần Deadlock Prevention, Deadlock Avoidance	100%
Nguyễn Hoàng Phúc	21127671	- Nghiên cứu, trình bày phần Deadlock Detection, Recovery from Deadlock	100%

II. Tổng quan về Deadlock

- Deadlock là một trạng thái của hệ thống trong đó các tiến trình hoặc luồng đang chờ đợi tài nguyên từ các tiến trình khác mà không ai sẵn sàng nhượng bộ tài nguyên của mình. Deadlock là một vấn đề phức tạp trong hệ thống máy tính và có thể xảy ra khi các tiến trình hoặc luồng cố gắng truy cập cùng một tài nguyên hoặc tài nguyên khác nhau mà không thể hoàn tất nhiệm vụ của chúng.

- Về tổng quan, có bốn phương pháp chính được sử dụng để giải quyết vấn đề Deadlock

1. Phớt lờ vấn đề Deadlock. bằng cách không để ý nó hay không quan tâm đến nó. (thuật toán con đà điểu)
2. Nhận diện và phục hồi. Cho phép deadlock có thể xảy ra, nếu có thì nhận biết Deadlock và tìm cách khắc phục.
3. Phòng tránh Deadlock bằng cách cấp phát tài nguyên một cách cẩn thận
4. Ngăn chặn, bằng cách không phép một trong bốn điều kiện để dẫn đến Deadlock có thể thỏa mãn.

- Trong báo cáo này chúng ta chỉ tập trung 3 phương pháp giải quyết Deadlock: ngăn chặn Deadlock, phòng tránh Deadlock, nhận diện và phục hồi từ Deadlock

III. Các phương pháp chính để giải quyết Deadlock

III.1. Ngăn chặn Deadlock (Deadlock Prevention)

- Có 4 điều kiện thỏa mãn cùng lúc để xảy ra Deadlock:

1. Độc quyền. Mỗi tài nguyên chỉ được cấp cho duy nhất một tiến trình tại một thời điểm, hoặc không cấp cho tiến trình nào hết.
2. Giữ và chờ: Tiến trình đang giữ tài nguyên và yêu cầu thêm tài nguyên mới
3. Không thu hồi. Hệ thống không thể thu hồi tài nguyên cấp cho một tiến trình nào đó, trừ khi tiến trình này trả lại tài nguyên
4. Vòng tròn chờ. Tồn tại một chuỗi vòng tròn các tiến trình chờ đợi tài nguyên của tiến trình kế tiếp trong chuỗi.

➔ Để tránh Deadlock, chúng ta cần ngăn chặn ít nhất một trong bốn điều kiện trên. Tuy nhiên, để hiểu rõ hơn về mỗi điều kiện, chúng ta cần xem xét chúng một cách cụ thể và độc lập.

III.1.1. Loại trừ lẫn nhau (Mutual exclusion)

Mutual exclusion là một trạng thái trong hệ thống khi hai hoặc nhiều tiến trình đồng thời yêu cầu truy cập vào cùng một tài nguyên và không thể thực hiện được vì tài nguyên đó chỉ có thể được sử dụng bởi một tiến trình tại một thời điểm. Để tránh tình trạng mutual exclusion, cần thỏa mãn các điều kiện sau đây:

- Tài nguyên phải được cấp cho một tiến trình duy nhất vào một thời điểm: Điều này đảm bảo rằng chỉ có một tiến trình được truy cập vào tài nguyên đó tại một thời điểm, do đó tránh được trường hợp hai tiến trình yêu cầu truy cập vào tài nguyên cùng một lúc.
- Tiến trình không được giữ tài nguyên mà không sử dụng: Điều này đảm bảo rằng tài nguyên sẽ được sử dụng bởi các tiến trình khác khi nó không được sử dụng bởi tiến trình hiện tại. Nếu tiến trình giữ tài nguyên mà không sử dụng, nó sẽ gây lãng phí tài nguyên và có thể làm giảm hiệu suất của hệ thống.
- Tiến trình không được giữ tài nguyên trong khi đợi tài nguyên khác: Điều này đảm bảo rằng các tiến trình không sẽ giữ tài nguyên mà không sử dụng, trong khi đang chờ đợi tài nguyên khác. Nếu các tiến trình giữ các tài nguyên mà không sử dụng trong khi đang chờ đợi các tài nguyên khác, nó có thể gây deadlock.

- Các kĩ thuật để tránh mutual exclusion xảy ra trong hệ thống

- Sử dụng khóa (lock): Phương pháp này sử dụng một biến khóa (lock) để đồng bộ hóa truy cập vào tài nguyên chung. Khi một tiến trình muốn sử dụng tài nguyên, nó phải yêu cầu khóa trước khi truy cập. Nếu khóa đang được sử dụng bởi một tiến trình khác, tiến trình

đang yêu cầu phải đợi cho đến khi khóa được giải phóng. Khi một tiến trình đã sử dụng xong tài nguyên, nó sẽ giải phóng khóa để cho tiến trình khác sử dụng.

- Sử dụng biến cờ (flag): Phương pháp này sử dụng một biến cờ (flag) để đồng bộ hóa truy cập vào tài nguyên chung. Khi một tiến trình muốn sử dụng tài nguyên, nó sẽ yêu cầu sử dụng biến cờ và đặt giá trị của biến này thành true để báo hiệu rằng tài nguyên đang được sử dụng bởi tiến trình này. Khi tiến trình đã sử dụng xong tài nguyên, nó sẽ đặt giá trị của biến cờ thành false để cho phép các tiến trình khác sử dụng.
- Sử dụng semaphores
  - Semaphore là một biến đồng bộ hóa được sử dụng để đồng bộ hóa truy cập vào tài nguyên. Semaphore bao gồm một giá trị nguyên và hai hoạt động đồng bộ hóa là P (proberen) và V (verhogen). Khi một tiến trình muốn truy cập tài nguyên, nó sẽ gọi hoạt động P để giảm giá trị của semaphore. Nếu giá trị của semaphore là 0, tiến trình sẽ bị khóa cho đến khi semaphore trở thành khả dụng.
  - Khi tiến trình sử dụng xong tài nguyên, nó sẽ gọi hoạt động V để tăng giá trị của semaphore lên một đơn vị, cho phép các tiến trình khác sử dụng tài nguyên. Semaphore còn được sử dụng để đồng bộ hóa truy cập vào các khu vực nhớ, thực hiện đồng bộ hóa giữa các tiến trình, xử lý báo hiệu và nhiều hơn nữa.
  - Có hai loại semaphore chính là binary semaphore và counting semaphore. Binary semaphore có giá trị chỉ bằng 0 hoặc 1 và được sử dụng để đồng bộ hóa truy cập vào tài nguyên đơn lẻ. Counting semaphore có giá trị bằng số nguyên không âm và được sử dụng để đồng bộ hóa truy cập vào nhiều tài nguyên.
  - Tuy nhiên, sử dụng semaphores cần phải chú ý để tránh các vấn đề như deadlock hoặc livelock. Chẳng hạn, nếu một tiến trình yêu cầu semaphore và bị khóa, nhưng không giải phóng semaphore, các tiến trình khác sẽ bị chặn và hệ thống sẽ bị deadlock. Hoặc nếu một tiến trình giải phóng semaphore quá nhanh, semaphore sẽ trở nên không đồng bộ và gây ra livelock. Do đó, việc sử dụng semaphores cần được thực hiện cẩn thận và có kế hoạch đầy đủ để tránh các vấn đề trên.

III.1.2. Giữ và chờ (Hold and Wait)

- Để đảm bảo rằng điều kiện "hold-and-wait" không bao giờ xảy ra trong hệ thống, chúng ta phải đảm bảo rằng khi một tiến trình yêu cầu một tài nguyên, nó không giữ bất kỳ tài nguyên nào khác. Một giao thức mà chúng ta có thể sử dụng là yêu cầu mỗi tiến trình yêu cầu và được cấp phát tất cả các tài nguyên của nó trước khi nó bắt đầu thực thi. Chúng ta có thể thực hiện điều này bằng cách yêu cầu các lệnh hệ thống yêu cầu tài nguyên cho một tiến trình đứng trước tất cả các lệnh hệ thống khác.

- Một giao thức thay thế cho phép một tiến trình chỉ yêu cầu các tài nguyên khi nó không có bất kỳ tài nguyên nào. Một tiến trình có thể yêu cầu một số tài nguyên và sử dụng chúng. Trước khi nó có thể yêu cầu bất kỳ tài nguyên bổ sung nào, nó phải giải phóng tất cả các tài nguyên mà nó đang được cấp phát.

- Để minh họa sự khác biệt giữa hai giao thức này, chúng ta xem xét một tiến trình sao chép dữ liệu từ ổ đĩa DVD đến một tập tin trên đĩa cứng, sắp xếp tập tin đó và sau đó in kết quả ra máy in. Nếu tất cả các tài nguyên phải được yêu cầu ở đầu tiên của quá trình, thì tiến trình phải yêu cầu ban đầu ổ đĩa DVD, tập tin đĩa cứng và máy in. Nó sẽ giữ máy in trong suốt quá trình thực thi, mặc dù nó chỉ cần máy in ở cuối.

- Phương pháp thứ hai cho phép tiến trình chỉ yêu cầu ban đầu ổ đĩa DVD và tập tin đĩa cứng. Nó sao chép từ ổ đĩa DVD vào đĩa cứng và sau đó giải phóng cả ổ đĩa DVD và tập tin đĩa cứng. Sau đó, tiến trình phải yêu cầu tập tin đĩa cứng và máy in. Sau khi sao chép tập tin đĩa cứng vào máy in, nó giải phóng hai tài nguyên này và kết thúc.

- Có hai nhược điểm chính của cả hai giao thức này. Thứ nhất, sử dụng tài nguyên có thể thấp, vì các tài nguyên có thể được cấp phát nhưng không sử dụng trong một khoảng thời gian dài. Trong ví dụ được đưa ra, chúng ta có thể giải phóng ổ đĩa DVD và tệp đĩa, và sau đó yêu cầu tệp đĩa và máy in, chỉ nếu chúng ta có thể đảm bảo dữ liệu của chúng tôi sẽ vẫn nằm trên tệp đĩa. Nếu không, chúng ta phải yêu cầu tất cả các tài nguyên ở đầu tiên cho cả hai giao thức.

- Thứ hai, đó là đói đến tài nguyên. Một quá trình cần nhiều tài nguyên phổ biến có thể phải đợi vô thời hạn, vì ít nhất một trong số các tài nguyên mà nó cần luôn được phân bổ cho một quá trình khác.

III.1.3. Không có quyền ưu tiên (No Preemption)

- Để tránh No Preemption, ta cần giải pháp để có thể thu hồi các tài nguyên đang được sử dụng bởi một tiến trình để có thể cấp cho một tiến trình khác, và giải pháp này được gọi là preemption.

- Có nhiều cách để thực hiện preemption, một số phương pháp được sử dụng phổ biến như:

1. Priority Inversion Protocol: Trong giao thức này, tiến trình có ưu tiên cao hơn sẽ được cấp tài nguyên trước, nhưng nếu một tiến trình có ưu tiên thấp đang sử dụng tài nguyên mà tiến trình ưu tiên cao cần sử dụng, tiến trình ưu tiên thấp sẽ giữ tài nguyên đó và tiến trình ưu tiên cao sẽ bị chặn (blocked) đến khi tiến trình ưu tiên thấp hoàn thành việc sử dụng tài nguyên.
2. Aging: Đây là một kỹ thuật giúp tránh tình trạng tiến trình bị đóng băng (starvation) bằng cách tăng độ ưu tiên của các tiến trình đã chờ đợi một thời gian dài.

3. Deadline-Monitoring Protocol: Trong giao thức này, hệ thống sẽ kiểm tra các tiến trình đang chờ tài nguyên xem chúng có thể hoàn thành trước một deadline nhất định hay không. Nếu không thể hoàn thành trong thời gian đó, tiến trình sẽ bị gián đoạn (interrupt) để tài nguyên của nó được thu hồi và cấp cho các tiến trình khác.

- Tuy nhiên, mỗi phương pháp đều có nhược điểm riêng. Ví dụ, trong Priority Inversion Protocol, tiến trình ưu tiên cao có thể bị chặn bởi một tiến trình ưu tiên thấp, làm giảm hiệu quả của hệ thống. Trong khi đó, trong Aging, độ ưu tiên của một tiến trình có thể quá cao sau một thời gian dài chờ đợi, dẫn đến ưu tiên của tiến trình này vượt qua các tiến trình khác. Trong Deadline-Monitoring Protocol, nếu deadline được đặt quá ngắn, nhiều tiến trình sẽ bị gián đoạn, gây giảm hiệu suất của hệ thống. Ngược lại, nếu deadline được đặt quá dài, các tiến trình khác sẽ phải chờ đợi quá lâu để sử dụng tài nguyên.

II.1.4. Vòng tròn chờ (Circular Wait)

- Có nhiều cách để tránh trạng thái circular wait:

1. Ordering resources: Xếp hàng đợi các tài nguyên theo một thứ tự cố định và yêu cầu các tiến trình tuân theo thứ tự này khi yêu cầu tài nguyên. Ví dụ, nếu tài nguyên A phải được sử dụng trước tài nguyên B, thì bất kỳ tiến trình nào yêu cầu tài nguyên A cũng phải yêu cầu tài nguyên B trước khi được cấp phát tài nguyên A. Phương pháp này có thể đảm bảo không xảy ra trạng thái circular wait nhưng cần thiết phải biết trước thứ tự ưu tiên của các tài nguyên.

➔ Phương pháp này đảm bảo không xảy ra trạng thái circular wait nhưng yêu cầu biết trước thứ tự ưu tiên của các tài nguyên và có thể gây ra hiệu suất kém nếu thứ tự này không được thiết lập đúng cách.

2. Resource allocation denial: Từ chối cấp phát tài nguyên nếu việc này sẽ dẫn đến trạng thái circular wait. Tuy nhiên, phương pháp này có thể gây ra tình trạng hạn chế sử dụng tài nguyên và có thể dẫn đến hiệu suất kém nếu các tiến trình không thể tiếp tục hoạt động vì không thể cấp phát tài nguyên.
   ➔ Phương pháp này đảm bảo không xảy ra trạng thái circular wait nhưng có thể dẫn đến hạn chế sử dụng tài nguyên và gây ra hiệu suất kém nếu các tiến trình không thể tiếp tục hoạt động vì không thể cấp phát tài nguyên.

3. Resource preemption: Giải phóng các tài nguyên của một tiến trình và cấp phát lại cho các tiến trình khác nếu cần thiết để tránh trạng thái circular wait. Tuy nhiên, phương pháp này có thể gây ra tình trạng độc quyền tài nguyên, khi một tiến trình có thể giữ các tài nguyên lâu hơn so với các tiến trình khác.
   ➔ Phương pháp này giải quyết trạng thái circular wait bằng cách giải phóng các tài nguyên của một tiến trình và cấp phát lại cho các tiến trình khác. Tuy nhiên, phương pháp này có thể gây ra tình trạng độc quyền tài nguyên và làm giảm hiệu suất nếu các tiến trình phải chờ đợi quá lâu để có thể sử dụng tài nguyên.

4. Wait-for graph and detection: Sử dụng đồ thị chờ đợi để phát hiện trạng thái circular wait và giải quyết vấn đề bằng cách tìm kiếm chu trình trong đồ thị và giải phóng tài nguyên. Phương pháp này hiệu quả trong việc giải quyết trạng thái circular wait nhưng tốn nhiều thời gian và tài nguyên tính toán.
   ➔ Phương pháp này hiệu quả trong việc giải quyết trạng thái circular wait nhưng tốn nhiều thời gian và tài nguyên tính toán để phát hiện và giải quyết vấn đề. Bên cạnh đó, phương pháp này có thể dẫn đến tình trạng chậm hơn nếu cần phải tạo và xử lý các đồ thị chờ đợi lớn.

III.2. Phòng tránh Deadlock (Deadlock Avoidance)

III.2.1. Thuật toán Resource-Allocation-Graph (RAG)

- Thuật toán Resource-Allocation-Graph là một trong những phương pháp được sử dụng trong việc phát hiện và tránh deadlock trong hệ thống máy tính. Thuật toán này sử dụng đồ thị tài nguyên (Resource Allocation Graph) để mô hình hóa việc sử dụng tài nguyên và quan hệ giữa các tiến trình.

- Các bước của thuật toán Resource-Allocation-Graph như sau:

1) Tạo đồ thị chờ đợi tài nguyên (Resource Allocation Graph) bằng cách tạo một đỉnh (vertex) cho mỗi tiến trình (process) và một đỉnh cho mỗi tài nguyên (resource). Tạo một cạnh (edge) từ mỗi tiến trình đến tài nguyên mà nó đang yêu cầu, và tạo một cạnh từ mỗi tài nguyên đến tiến trình nào đang giữ tài nguyên đó.
2) Kiểm tra xem đồ thị có chứa chu trình không. Nếu có chu trình, điều này cho thấy rằng hệ thống đang ở trạng thái deadlock.
3) Tìm các tiến trình có thể được giải phóng tài nguyên mà chúng đang giữ. Nếu có ít nhất một tiến trình có thể được giải phóng tài nguyên, hệ thống không đang ở trạng thái deadlock.
4) Nếu không có tiến trình nào có thể được giải phóng tài nguyên, hệ thống đang ở trạng thái deadlock và không thể tiếp tục hoạt động.
5) Nếu hệ thống đang ở trạng thái deadlock, giải quyết vấn đề bằng cách giải phóng một số tài nguyên. Có thể giải phóng tài nguyên bằng cách tạm dừng một số tiến trình hoặc giải phóng các tài nguyên được sử dụng không hiệu quả.
6) Sau khi đã giải phóng tài nguyên, kiểm tra lại đồ thị chờ đợi tài nguyên để đảm bảo rằng không còn chu trình và hệ thống có thể tiếp tục hoạt động.

➔ Thuật toán Resource-Allocation-Graph giúp cho việc phát hiện trạng thái deadlock trở nên dễ dàng hơn, tuy nhiên, nó không phải là phương pháp hoàn hảo và có thể bị sai sót trong một số trường hợp đặc biệt. Ngoài ra, nó cũng không giải quyết được các trường hợp xảy ra deadlock mà không có chu trình trong đồ thị tài nguyên.

III.2.2. Giải thuật nhà băng cho một loại tài nguyên

- Để tránh Deadlock, Dijkstra (1965) đã đề xuất một giải thuật cho nhà băng để kiểm tra tính an toàn của việc cho mượn tiền cho khách hàng. Giả sử có một nhà băng với bốn khách hàng và mỗi khách hàng được cấp một hạn mức tín dụng. Giải thuật sẽ

kiểm tra xem việc cho mượn tiền có đảm bảo rằng mỗi khách hàng đều có thể mượn tối đa theo hạn mức của họ không. Nếu việc cho mượn tiền có thể dẫn đến tình trạng không an toàn (nhà băng hết tiền cho mượn trong khi khách hàng chưa vay đủ hạn mức của họ), thì yêu cầu sẽ bị từ chối.

	Mượn	Tối đa
A	0	5
B	0	4
C	0	3
D	0	6

Bảng 1: còn dư 10 (trạng thái an toàn)

	Mượn	Tối đa
A	1	5
B	1	4
C	2	3
D	4	6

Bảng 2: còn dư 2 (trạng thái an toàn)

	Mượn	Tối đa
A	1	5
B	2	4
C	2	3
D	4	6

Bảng 3: còn dư 1 (trạng thái không an toàn)

- Nhà băng biết được không phải lúc nào mọi người cũng mượn hết số tín dụng của họ (18) vì vậy nhà băng chỉ có tối đa 10. Tình huống trong **bảng 1** rõ ràng là an toàn, mọi người có thể mượn được tối đa. Tình huống **bảng 2** cũng là tình huống an toàn, vì nhà băng có thể cho C mượn 2 để C thực thi xong và trả lại 4. Lúc này hệ thống có thể thỏa mãn B và D. Khi hoặc D trả lại thì hệ thống có thể thỏa mãn yêu cầu của A. Tuy nhiên ở tình huống **bảng 3**, khi B yêu cầu thêm 1 và nhà băng đồng ý thì hệ thống lại không an toàn. Vì giả sử lúc này cả bốn khách hàng yêu cầu mượn tối đa. Thì hệ thống không thể thỏa mãn khách hàng nào, dẫn đến Deadlock.

III.2.3. Giải thuật nhà băng cho trường hợp tổng quát (nhiều loại tài nguyên)

- Thuật toán đồ thị cấp phát tài nguyên (resource-allocation-graph algorithm) không áp dụng được cho hệ thống cấp phát tài nguyên có nhiều phiên bản của từng loại tài nguyên. Thuật toán tránh bế tắc mà chúng tôi mô tả tiếp theo có thể áp dụng cho một hệ thống như vậy nhưng kém hiệu quả hơn sơ đồ đồ thị phân bổ tài nguyên. Thuật toán này được biết đến là thuật toán banker. Tên được đặt như vậy vì thuật toán có thể được sử dụng trong một hệ thống ngân hàng để đảm bảo rằng ngân hàng không bao giờ phân phát tiền sẵn có của nó theo cách mà nó không còn có thể đáp ứng nhu cầu của tất cả khách hàng của mình.

- Khi một tiến trình mới vào hệ thống, nó phải khai báo giá trị tối đa số lượng phiên bản của từng loại tài nguyên mà nó có thể cần. Con số này có thể không vượt quá tổng số tài nguyên trong hệ thống. Khi người dùng yêu cầu một tập hợp các tài nguyên trong hệ thống, hệ thống phải xác định xem liệu việc phân bổ các tài nguyên này tài nguyên sẽ rời khỏi hệ thống trong trạng thái an toàn hay không. Nếu an toàn, các tài nguyên sẽ được phân bổ; nếu không, tiến trình phải chờ cho đến khi một số quy trình khác phát hành đủ tài nguyên.

- Một số cấu trúc dữ liệu phải được duy trì đẻ cài đặt thuật toán banker. Các cấu trúc dữ liệu này mã hóa trạng thái cấp phát tài nguyên hệ thống. Chúng ta cần các cấu trúc dữ liệu sau, trong đó n là số tiến trình trong hệ thống và m là số loại tài nguyên:

  - **Available:** Một vector kích thước m để chỉ số tài nguyên có sẵn của mỗi loại. Nếu Available[j] bằng k, thì k tài nguyên của loại Rj có sẵn.
  - **Max:** Một ma trận n x m định nghĩa nhu cầu tối đa có thể đạt được của từng tiến trình. Nếu Max[i][j] bằng k, thì tiến trình Pi có thể yêu cầu tối đa k tài nguyên của loại Rj.
  - **Allocation:** Một ma trận n x m định nghĩa số tài nguyên của từng loại hiện tại phân bổ cho từng tiến trình. Nếu Allocation[i][k] bằng k, thì tiến trình Pi được phân bổ k tài nguyên loại Rj.
  - **Need:** Một ma trận n x m chỉ ra nhu cầu tài nguyên còn lại của mỗi quá trình. Nếu Need[i][j] bằng k, thì tiến trình Pi có thể cần hơn k tài nguyên của loại Rj để hoàn thành nhiệm vụ của nó. Chú ý rằng Need[i][j] bằng [i][j] – Allocation[i][j].

- Để đơn giản hóa việc trình bày thuật toán banker, giả sử X và Y là các vector kích thước n. Chúng ta nói rằng $X \leq Y$ nếu và chỉ nếu $X[i] \leq Y[i]$ với mọi i = 1, 2,

…, n. Ví dụ nếu X = (1,7,3,2) và Y = (0,3,2,1) thì Y ≤ X. Ngoài ra, Y < X nếu Y ≤ X và Y ≠ X.

- Chúng ta có thể coi mỗi hàng trong ma trận Allocation và Need là các vector và gọi chúng là $Allocation_i$ và $Need_i$. Vector $Allocation_i$ chỉ định các tài nguyên hiện được phân bổ cho tiến trình P_i; vector $Need_i$ chỉ định tài nguyên bổ sung rằng P_i vẫn có thể yêu cần để hoàn thành nhiệm vụ của mình.

- Một số nghiên cứu đã chỉ ra rằng về lý thuyết thuật toán rất tuyệt, nhưng về thực tế thì nó không sử dụng được. Lý do chính là các tiến trình khó mà biết được nó cần bao nhiêu tài nguyên thì mới thực thi được. Thêm vào đó số lượng các tiến trình thay đổi theo thời gian. Các tài nguyên có thể bị hư hỏng một cách bất ngờ. Như vậy trong thực tế, nếu có thì rất ít hệ điều hành sử dụng thuật toán nhà băng để phòng tránh Deadlock.

III.3. Nhận diện Deadlock (Deadlock Detection)

III.3.1. Nhận diện Deadlock: mỗi loại tài nguyên chỉ có một

- Trong hệ thống này, chúng ta vẽ đồ thị cấp phát và chiếm giữ tài nguyên như đề xuất của Holt ở phần mô hình hóa Deadlock. Rồi chúng ta kiểm tra xem đồ thi có tồn tại mạch vòng nào hay không. Nếu có tồn tại thì hệ thống bị Deadlock. Sau đây chúng em sẽ trình bày một số thuật toán đơn giản để kiểm tra mạch vòng. Đây là các bước tuần tự của thuật toán:

1) Đồ thị có N nút, khởi hành tại nút N
2) Khởi động danh sách L là một danh sách rỗng, tất cả các cạnh không đánh dấu.
3) Thêm nút hiện tại vào danh sách L, kiểm tra xem nút đã tồn tại trước đó chưa. Nếu có thì đồ thị chứa mạch vòng và kết thúc.
4) Tại nút hiện hành, kiểm tra xem có cạnh nào xuất phát từ nút này mà chưa đánh dấu hay không. Nếu có thì qua bước 5, còn không thì qua bước 6.
5) Ngẫu nhiên chọn một cạnh như trên và đánh dấu nó, cập nhập lại nút hiện hành mới và quay lại bước 3.
6) Loại bỏ nút này, và trờ về nút liền trước, xem nút này là nút hiện hành, về bước 3. Nếu nút này là nút xét đầu tiên thì đồ thị không có mạch vòng, thuật toán kết thúc.

➔ Thuật toán ở trên sử dụng ý tưởng của thuật toán tìm kiếm chiều sâu (depth-first-search). Giải thuật này còn xa với thuật toán tối ưu. Tuy nhiên cũng chứng minh được có tồn tại thuật toán nhận diện Deadlock.

III.3.2. Nhận diện Deadlock: mỗi loại tài nguyên có nhiều tài nguyên

- Khi mỗi loại tài nguyên có nhiều tài nguyên thì chúng ta sử dụng thuật toán ma trận (matrix-based) để nhận diện deadlock của hệ thống có n tiến trình. Gọi P_1 cho đến P_n. Gọi m là số loại tài nguyên, E_i là tài nguyên loại i, $E_i = 2$ nghĩa là loại này có hai tài nguyên (ví dụ máy có hai đầu đọc CD). E là vector tài nguyên của hệ thống.

- Tại một thời điểm, Gọi $A(A_1,A_2,\ldots,A_n)$ là vector các tài nguyên chưa cấp phát. Nếu tất cả các tài nguyên loại i đã được cấp phát thì Ai = 0.

- Chúng ta lập ra hai ma trận. Ma trận C, ma trận tài nguyên đã cấp phát, và ma trận R, ma trận yêu cầu tài nguyên.

- Thuật toán nhận diện Deadlock nhờ vào sự so sánh của các vector. Chúng ta nói vector $A \leq B$ nếu như tất cả các thành phần của vector A đều nhỏ hơn hoặc bằng thành phần tương ứng trong vector B.

- Khởi đầu các tiến trình đều không đánh dấu. Khi thuật toán thực thi xong tất cả các tiến trình được đánh dấu, nghĩa là có đủ tài nguyên để thực thi thì sẽ không bị Deadlock. Nếu có một tiến trình chưa đánh dấu nghĩ là hệ thống bị Deadlock. Thuật toán như sau:

1) Chọn một tiến trình chưa đánh dấu, Pi, sao cho vector tại dòng thứ i trong ma trận R nhỏ hơn hoặc bằng vector A.
2) Nếu có. Thì cập nhập lại vector A(cộng tài nguyên đang cấp cho tiến trình P_i, nghĩa là dòng thứ i trong ma trận C vào vector A. Đánh dấu tiến trình P_i.
3) Nếu không có tiến trình nào thỏa. Kết thúc.

- Khi thuật toán kết thúc, nếu có tồn tại bất kì tiến trình nào chưa được đánh dấu thì hệ thống Deadlock.

- Ở bước thứ 2 của thuật toán, khi một tiến trình có thể được thỏa mãn tất cả tài nguyên đang yêu cầu, thì chúng ta sẽ cấp tài nguyên đang yêu cầu cho chúng, và chúng thực thi xong trả lại toàn bộ tài nguyên cho hệ thống. Vì vậy chúng ta có thể cập nhập lại vector A bằng cách tăng thêm tài nguyên mà tiến trình này đang chiếm giữ (đã được cấp phát trước đây).

III.4. Phục hồi từ Deadlock (Recovery from Deadlock)

- Có 3 cách phục hồi từ Deadlock:

1) Phục hồi bằng cách thu hồi tài nguyên
   - Hệ thống sẽ thu hồi lại tài nguyên đã cấp cho một tiến trình và cấp cho một tiến trình khác để tiếp tục thực thi. Tài nguyên mới được sử dụng sẽ được giải phóng sau khi các tiến trình hoàn tất yêu cầu của chúng.
2) Phục hồi dựa vào rollback
   - Hệ thống tạo các điểm kiểm tra tại những khoảng thời gian cố định để lưu lại trạng thái của toàn hệ thống, bao gồm trạng thái của các tài nguyên.
   - Khi xảy ra tình trạng Deadlock, hệ thống quay lại vị trí điểm kiểm tra và cấp lại các tài nguyên đang chờ đợi từ đó. Tuy nhiên, việc thực hiện rollback có thể làm mất đi các thay đổi được thực hiện từ điểm kiểm tra đến thời điểm xảy ra Deadlock.
3) Phục hồi bằng cách kết thúc tiến trình
   - Đây là phương pháp đơn giản nhất, hệ thống sẽ chọn và kết thúc một tiến trình nằm trong vòng lặp Deadlock và cứ tiếp tục như vậy cho tới khi hết Deadlock. Tuy nhiên, phương pháp này có thể ảnh hưởng đến tính nhất quán của dữ liệu mà tiến trình đang truy cập, đặc biệt là trong trường hợp tiến trình đang thao tác với cơ sở dữ liệu.

IV. Nguồn tham khảo

(1): Sách: Operating-System-Concepts.9th.Edi.Abraham-Silberschatz.2012 (https://drive.google.com/drive/folders/1-bz_MZPNp8DHevgNVAYQs6Kj-65qmU4R)

(2): Thuật toán Nhà băng: https://en.wikipedia.org/wiki/Banker%27s_algorithm#:~:text=Banker's%20algorithm%20is%20a%20resource,conditions%20for%20all%20other%20pending

(3): Giáo trình Hệ điều hành – FIT.HCMUS (Trần Trung Dũng – Phạm Tuấn Sơn).