CHƯƠNG 7. TẮT NGHẼN

7.1. Mô hình

Một hệ thống bao gồm một số lượng tài nguyên hữu hạn được phân phối cho một số luồng cạnh tranh. Các tài nguyên có thể được phân chia thành một số loại, mỗi loại bao gồm một số cá thể (instance) giống hệt nhau. Chẳng hạn như CPU, tệp và thiết bị I/O. Nếu một hệ thống có bốn CPU, thì loại tài nguyên CPU có bốn cá thể. Nếu một luồng yêu cầu một cá thể của kiểu tài nguyên, thì hệ thống cấp phát một cá thể của kiểu tài nguyên đó.

Các công cụ đồng bộ hóa khác nhau chẳng hạn như khóa mutex và semaphores, cũng là tài nguyên hệ thống; và trên các hệ thống máy tính hiện đại, chúng là những nguyên nhân phổ biến nhất gây ra tắc nghẽn. Một khóa thường được liên kết với một cấu trúc dữ liệu cụ thể - nghĩa là, một khóa có thể được sử dụng để bảo vệ quyền truy cập vào hàng đợi, khóa khác để bảo vệ quyền truy cập vào danh sách được liên kết, v.v. Vì lý do đó, mỗi phiên bản của một khóa thường được gán cho lớp tài nguyên riêng của nó.

Một luồng phải yêu cầu một tài nguyên trước khi sử dụng nó và phải giải phóng tài nguyên sau khi sử dụng nó. Một luồng có thể yêu cầu nhiều tài nguyên để thực hiện nhiệm vụ của nó. Rõ ràng, số lượng tài nguyên được yêu cầu không được vượt quá tổng số tài nguyên hiện có trong hệ thống.

Trong chế độ hoạt động bình thường, một luồng chỉ có thể sử dụng tài nguyên theo trình tự sau:

1. Yêu cầu. Luồng yêu cầu tài nguyên. Nếu yêu cầu không thể được cấp ngay lập tức (ví dụ: nếu một khóa mutex hiện đang được giữ bởi một luồng khác), thì luồng yêu cầu phải đợi cho đến khi nó có thể nhận được tài nguyên.

2. Sử dụng. Luồng có thể hoạt động trên tài nguyên (ví dụ: nếu tài nguyên là khóa mutex, luồng có thể truy cập vào miền găng).

3. Giải phóng. Luồng giải phóng tài nguyên.

Khi một luồng sử dụng tài nguyên do nhân quản lý, hệ điều hành sẽ kiểm tra để

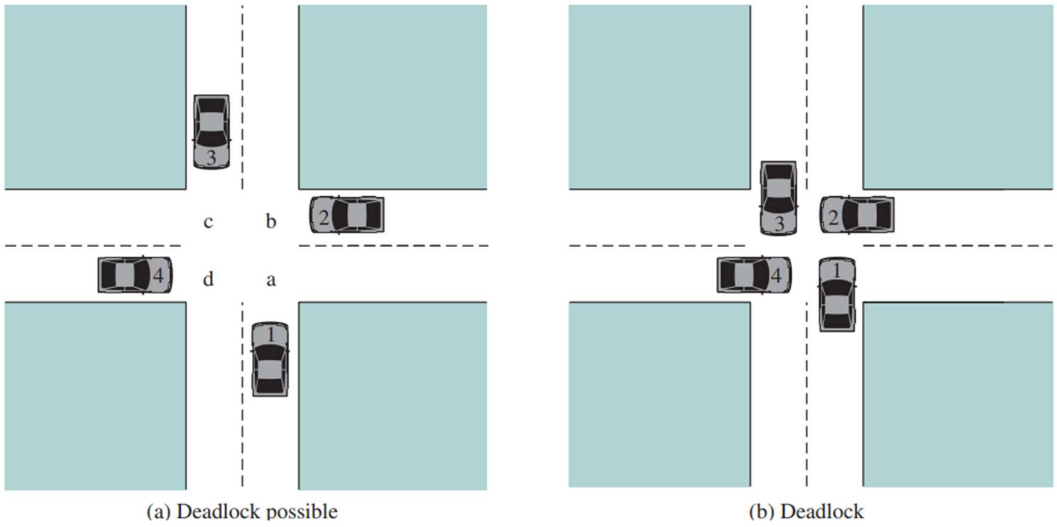
đảm bảo rằng luồng đã yêu cầu và đã được cấp phát tài nguyên. Một bảng hệ thống ghi lại xem mỗi tài nguyên là tự do hay đã cấp phát. Đối với mỗi tài nguyên được cấp phát, bảng cũng ghi lại luồng mà nó được cấp phát. Nếu một luồng yêu cầu một tài nguyên hiện được cấp phát cho một luồng khác, nó có thể được thêm vào một hàng đợi các luồng đang chờ tài nguyên này.

Một tập các luồng ở trạng thái tắc nghẽn khi mọi luồng trong tập đó đang chờ một sự kiện mà gây ra bởi luồng khác trong tập hợp. Các sự kiện này chủ yếu là nhận và giải phóng tài nguyên. Các tài nguyên thường logic (không phải vật lí) , chẳng hạn như khóa mutex, semaphores và tệp.

7.2. Các ví dụ tắc nghẽn

7.2.1. Tắc nghẽn giao thông

Một loại tắc nghẽn ta hay gặp trong thực tế đó là tắc nghẽn giao thông trong các thành phố lớn. Ví dụ như Hình 7-1.



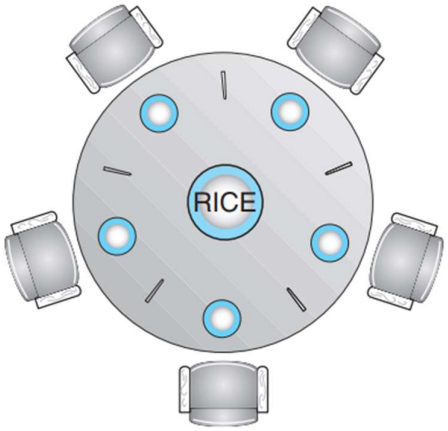
Hình 7-1. *Tắc nghẽn giao thông; (a) có thể tắc nghẽn, (b) tắc nghẽn.*

Trong hình trên ta thấy xe 1 chờ xe 2, xe 2 chờ xe 3, xe 3 chờ xe 4 và xe 4 lại chờ xe 1.

7.2.2. Bài toán các nhà triết học ăn tối

Hãy xét năm triết gia dành cả đời để suy nghĩ và ăn uống. Các nhà triết học chia sẻ một chiếc bàn tròn được bao quanh bởi năm chiếc ghế, mỗi chiếc thuộc về một triết gia. Chính giữa bàn là một bát cơm, trên bàn được bày năm chiếc đũa đơn (Hình 7-2). Khi một triết gia suy nghĩ, ông ấy không tương tác với các đồng nghiệp của mình. Đôi khi, một nhà triết học cảm thấy đói và cố gắng nhặt hai chiếc đũa gần nhất (chiếc đũa

nằm giữa ông ấy và những người hàng xóm bên trái và bên phải). Một triết gia có thể chỉ nhặt một chiếc đũa mỗi lần. Rõ ràng là ông ta không thể nhặt một chiếc đũa đã có trong tay của một người hàng xóm. Khi một triết gia đói có cả hai chiếc đũa cùng một lúc, ông ta ăn mà không cần nhả đũa ra. Khi ăn xong, ông ta đặt cả hai đũa xuống và bắt đầu suy nghĩ lại. Bài toán nhà triết học ăn uống được coi là một bài toán đồng bộ cổ điển không phải vì tầm quan trọng thực tế của nó mà vì nó là một ví dụ của một nhóm các bài toán điều khiển đồng thời. Nó là một đại diện đơn giản về sự cần thiết phải phân bổ một số tài nguyên giữa một số tiến trình theo cách không có tắc nghẽn và không bị chết đói.



Hình 7-2. *Mô tả tình huống 5 nhà triết học ăn tối.*

7.2.3. Tắc nghẽn trong hệ thống đệm

Hệ thống lưu đệm cải thiện thông lượng hệ thống bằng cách tách chương trình khỏi các thiết bị chậm như máy in. Ví dụ: nếu một chương trình gửi dòng tới máy in phải đợi từng trang được in trước khi có thể truyền trang tiếp theo thì chương trình sẽ thực thi chậm. Để tăng tốc độ thực thi chương trình, hệ thống lưu đệm sẽ định tuyến các trang đầu ra tới một thiết bị nhanh hơn nhiều, chẳng hạn như đĩa cứng, nơi chúng được lưu trữ tạm thời cho đến khi có thể in.

Hệ thống lưu trữ có thể dễ bị bế tắc. Một số hệ thống lưu trữ yêu cầu phải có đầu ra hoàn chỉnh từ một chương trình trước khi bắt đầu in. Một số công việc đã hoàn thành một phần tạo các trang vào tệp bộ đệm có thể bị bế tắc nếu dung lượng của đĩa đầy trước khi công việc nào hoàn thành. Người dùng hoặc quản trị viên hệ thống có thể loại bỏ một hoặc nhiều công việc để cung cấp đủ dung lượng lưu trữ cho các công việc còn lại hoàn thành.

Thông thường, quản trị viên hệ thống chỉ định dung lượng cho việc lưu trữ các tệp. Một cách để làm cho ít có khả năng xảy ra bế tắc hơn là cung cấp nhiều không gian cho

việc lưu trữ các tệp so với mức cần thiết. Giải pháp này có thể không khả thi nếu không gian ở mức cao. Một giải pháp phổ biến hơn là hạn chế bộ đệm đầu vào để chúng không chấp nhận các lệnh in bổ sung khi các tệp đệm bắt đầu đạt đến ngưỡng bão hòa nào đó, chẳng hạn như đầy 75%. Điều này có thể làm giảm thông lượng hệ thống nhưng đó là cái giá phải trả để giảm khả năng xảy ra bế tắc.

Các hệ thống ngày nay phức tạp hơn. Nó có thể cho phép tiến trình in bắt đầu trước khi công việc hoàn thành vì vậy một tệp lưu trữ đầy hoặc gần đầy có thể bắt đầu làm giải phóng trong khi công việc vẫn đang thực thi. Khái niệm này đã được áp dụng để phát trực tuyến các clip âm thanh và video, trong đó âm thanh và video bắt đầu phát trước khi các clip được tải xuống đầy đủ. Trong nhiều hệ thống, việc phân bổ không gian lưu trữ đã được thực hiện năng động hơn, do đó nếu không gian hiện có bắt đầu lấp đầy thì có thể có thêm không gian trống.

7.3. Đặc tính của tắc nghẽn

7.3.1. Điều kiện xuất hiện tắc nghẽn

Một tình huống tắc nghẽn có thể phát sinh nếu bốn điều kiện sau đây đồng thời tồn tại trong một hệ thống:

1. Loại trừ lẫn nhau. Ít nhất một tài nguyên phải được giữ ở chế độ không thể chia sẻ; nghĩa là mỗi lần chỉ có một luồng có thể sử dụng tài nguyên. Nếu một luồng khác yêu cầu tài nguyên đó, thì luồng yêu cầu phải chờ cho đến khi tài nguyên đó đã được giải phóng.

2. Giữ và chờ đợi. Một luồng phải đang nắm giữ ít nhất một tài nguyên và chờ đợi để có được các tài nguyên bổ sung hiện đang được các luồng khác nắm giữ.

3. Độc quyền. Các tài nguyên độc quyền; nghĩa là, một tài nguyên chỉ có thể được giải phóng một cách tự nguyện bởi luồng đang giữ nó, sau khi luồng đó đã hoàn thành nhiệm vụ của nó.

4. Vòng chờ. Tập hợp các luồng $\{T_0, T_1, ..., T_n\}$ chờ. T_0 đang chờ tài nguyên do T_1 giữ, T_1 đang chờ tài nguyên do T_2 giữ, ..., T_{n-1} đang chờ tài nguyên do T_n giữ và T_n đang chờ tài nguyên do T_0 giữ.

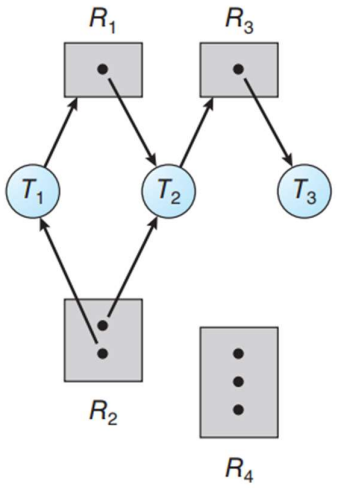
7.3.2. Đồ thị cấp phát tài nguyên

Tắc nghẽn có thể được mô tả chính xác hơn dưới dạng đồ thị có hướng được gọi là đồ thị phân bổ tài nguyên hệ thống. Đồ thị này bao gồm một tập các đỉnh V và một tập các cạnh E. Tập các đỉnh V được chia thành hai loại nút khác nhau: $T = \{T_1, T_2, ..., T_n\}$, tập hợp bao gồm tất cả các luồng hoạt động trong hệ thống và $R = \{R_1, R_2, ..., R_m\}$,

tập hợp bao gồm tất cả các loại tài nguyên trong hệ thống.

Một cạnh có hướng từ luồng T_i đến kiểu tài nguyên R_j được ký hiệu là $T_i \rightarrow R_j$; nó biểu thị rằng luồng T_i đã yêu cầu một thể hiện của loại tài nguyên R_j và hiện đang đợi tài nguyên đó. Một cạnh có hướng từ kiểu tài nguyên R_j đến luồng T_i được ký hiệu là $R_j \rightarrow T_i$; nó biểu thị rằng một thể hiện của kiểu tài nguyên R_j đã được cấp phát cho luồng T_i. Một cạnh có hướng $T_i \rightarrow R_j$ được gọi là một cạnh yêu cầu; một cạnh có hướng $R_j \rightarrow T_i$ được gọi là một cạnh gán.

Về cơ bản, chúng ta biểu diễn mỗi luồng T_i dưới dạng hình tròn và mỗi loại tài nguyên R_j dưới dạng hình chữ nhật. Ví dụ đơn giản, đồ thị phân bổ tài nguyên trong Hình 7-3 minh họa tình huống deadlock. Vì kiểu tài nguyên R_j có thể có nhiều hơn một thể hiện, chúng ta biểu diễn mỗi thể hiện đó như một dấu chấm trong hình chữ nhật. Lưu ý rằng cạnh yêu cầu trỏ đến hình chữ nhật R_j, trong khi cạnh gán cũng phải trỏ đến một trong các dấu chấm trong hình chữ nhật.



Hình 7-3. *Đồ thị cấp phát tài nguyên.*

Khi luồng T_i yêu cầu một thể hiện của kiểu tài nguyên R_j, một cạnh yêu cầu sẽ được chèn vào đồ thị phân bổ tài nguyên. Khi yêu cầu này có thể được thực hiện, cạnh yêu cầu được chuyển đổi ngay lập tức thành một cạnh gán. Khi luồng không cần truy cập tài nguyên nữa, nó sẽ giải phóng tài nguyên. Kết quả là, cạnh gán bị xóa.

Biểu đồ phân bổ tài nguyên trong Hình 7-3 mô tả tình huống sau:

- Tập hợp T, R và E:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$

◦ E = {$T_1 \rightarrow R_1$, $T_2 \rightarrow R_3$, $R_1 \rightarrow T_2$, $R_2 \rightarrow T_2$, $R_2 \rightarrow T_1$, $R_3 \rightarrow T_3$}

• Các thể hiện tài nguyên:

◦ Một thể hiện của loại tài nguyên R_1.

◦ Hai thể hiện của loại tài nguyên R_2.

◦ Một thể hiện của loại tài nguyên R_3.

◦ Ba thể hiện của loại tài nguyên R_4.

• Trạng thái luồng:

◦ Luồng T_1 đang giữ một thể hiện của loại tài nguyên R_2 và đang chờ một thể hiện của loại tài nguyên R_1.

◦ Luồng T_2 đang giữ một thể hiện của R_1 và một thể hiện của R_2 và đang đợi một thể hiện của R_3.

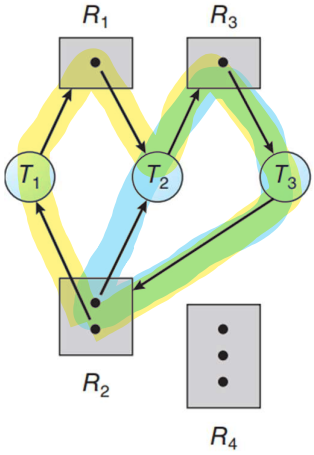
◦ Luồng T_3 đang giữ một thể hiện của R_3.

Đưa ra định nghĩa về đồ thị phân bổ tài nguyên, có thể chỉ ra rằng, nếu đồ thị không chứa chu trình, thì không có luồng nào trong hệ thống bị tắc nghẽn. Nếu đồ thị có chứa một chu trình, thì có thể tồn tại một tắt nghẽn.

Nếu mỗi loại tài nguyên có chính xác một thể hiện, thì một chu trình ngụ ý rằng một tắc nghẽn đã xảy ra. Nếu chu trình chỉ liên quan đến một tập hợp các loại tài nguyên, mỗi loại chỉ có một thể hiện duy nhất, thì một tắc nghẽn đã xảy ra. Mỗi luồng tham gia vào chu trình bị khóa. Trong trường hợp này, một chu trình trong đồ thị vừa là điều kiện cần vừa là điều kiện đủ để tồn tại deadlock.

Nếu mỗi loại tài nguyên có một số thể hiện, thì một chu kỳ không nhất thiết ngụ ý rằng một tắc nghẽn đã xảy ra. Trong trường hợp này, một chu trình trong đồ thị là điều kiện cần nhưng không phải là điều kiện đủ để tồn tại tắc nghẽn.

Để minh họa khái niệm này, chúng ta quay lại đồ thị phân bổ tài nguyên được mô tả trong Hình 7-3. Giả sử rằng luồng T_3 yêu cầu một thể hiện của loại tài nguyên R_2. Vì hiện tại không có thể hiện tài nguyên nào, chúng ta thêm một cạnh yêu cầu $T_3 \rightarrow R_2$ vào đồ thị (Hình 7-4).



Hình 7-4. *Đồ thị cấp phát tài nguyên có tắc nghẽn.*

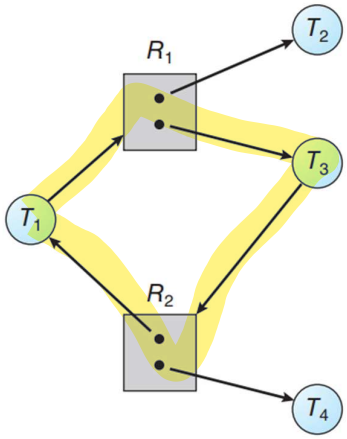
Tại thời điểm này, tồn tại hai chu trình trong hệ thống:

$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$.

$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$.

Các luồng T_1, T_2 và T_3 bị khóa. Luồng T_2 đang đợi tài nguyên R_3, được giữ bởi luồng T_3. Luồng T_3 đang đợi luồng T_1 hoặc luồng T_2 giải phóng tài nguyên R_2. Ngoài ra, luồng T_1 đang chờ luồng T_2 giải phóng tài nguyên R_1.

Xét ví dụ biểu đồ phân bổ tài nguyên trong Hình 7-5.



Hình 7-5. *Đồ thị cấp phát tài nguyên có chu trình nhưng không tắc nghẽn.*

Trong ví dụ này, chúng ta cũng có một chu trình: $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$. Tuy nhiên, không có tắc nghẽn. Ta thấy luồng T_4 có thể giải phóng thể hiện của loại tài nguyên R_2. Tài nguyên này sau đó có thể được phân bổ cho T_3, phá vỡ chu trình.

Tóm lại, nếu đồ thị phân bổ tài nguyên không có chu trình thì hệ thống không ở trạng thái tắc nghẽn. Nếu có một chu trình, thì hệ thống có thể ở trạng thái tắc nghẽn

hoặc không. Quan sát này rất quan trọng khi chúng ta giải quyết vấn đề tắc nghẽn.

7.4. Ngăn ngừa tắc nghẽn

Như chúng ta đã lưu ý trong Phần 7.3.1, để xảy ra tắc nghẽn, cả bốn điều kiện cần thiết phải xuất hiện. Bằng cách đảm bảo rằng ít nhất một trong những điều kiện này không thể xảy ra, chúng ta có thể ngăn chặn sự xuất hiện của tắc nghẽn.

7.4.1. Loại trừ lẫn nhau

Điều kiện để xuất hiện tình huống loại trừ lẫn nhau, đó là, ít nhất một tài nguyên phải không chia sẻ được. Các tài nguyên có thể chia sẻ không yêu cầu quyền truy cập loại trừ lẫn nhau và do đó không thể liên quan đến tắc nghẽn. Tệp chỉ đọc là một ví dụ điển hình về tài nguyên có thể chia sẻ. Nếu một số luồng cố gắng mở tệp chỉ đọc cùng một lúc, chúng có thể được cấp quyền truy cập đồng thời vào tệp. Một luồng không cần phải đợi một tài nguyên có thể chia sẻ được. Tuy nhiên, nói chung, chúng ta không thể ngăn chặn tắc nghẽn bằng cách loại bỏ điều kiện loại trừ lẫn nhau, bởi vì bản chất một số tài nguyên không thể chia sẻ được. Ví dụ, một khóa mutex không thể được chia sẻ đồng thời bởi một số luồng.

7.4.2. Giữ và đợi

Để đảm bảo rằng điều kiện giữ và chờ không bao giờ xảy ra trong hệ thống, chúng ta phải đảm bảo rằng, bất cứ khi nào một luồng yêu cầu một tài nguyên, nó không giữ bất kỳ tài nguyên nào khác. Một giao thức mà chúng ta có thể sử dụng bắt buộc mỗi luồng yêu cầu và được cấp phát tất cả tài nguyên của nó trước khi nó bắt đầu thực thi. Tất nhiên, điều này là không thực tế đối với hầu hết các ứng dụng do tính chất động của việc yêu cầu tài nguyên.

Một giao thức thay thế cho phép một luồng chỉ yêu cầu tài nguyên khi nó không có. Một luồng 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 hiện được cấp phát. Cả hai giao thức này đều có hai nhược điểm chính. Thứ nhất, việc sử dụng tài nguyên có thể thấp, vì tài nguyên có thể được phân bổ nhưng không được sử dụng trong một thời gian dài.

Ví dụ: một luồng có thể được cấp phát một khóa mutex cho toàn bộ quá trình thực thi của nó, nhưng chỉ yêu cầu nó trong một thời gian ngắn. Thứ hai, có thể xảy ra chết đói. Một luồng cần một số tài nguyên phổ biến có thể phải chờ vô thời hạn, bởi vì ít nhất một trong các tài nguyên mà nó cần luôn được phân bổ cho một số luồng khác.

7.4.3. Không ưu tiên (độc quyền)

Điều kiện cần thiết thứ ba đối với tắc nghẽn là độc quyền các tài nguyên đã được cấp phát. Để đảm bảo rằng điều kiện này không tồn tại, chúng ta có thể sử dụng giao thức sau. Nếu một luồng đang giữ một số tài nguyên và yêu cầu một tài nguyên khác mà không thể cấp phát ngay cho nó (tức là luồng phải đợi), thì tất cả các tài nguyên mà luồng hiện đang nắm giữ sẽ nhường lại (không độc quyền chiếm giữ). Nói cách khác, các tài nguyên này được giải phóng một cách ngầm định. Các tài nguyên được không độc quyền được thêm vào danh sách các tài nguyên mà luồng đang đợi. Luồng sẽ chỉ được khởi động lại khi nó có thể lấy lại các tài nguyên cũ, cũng như các tài nguyên mới mà nó đang yêu cầu.

Ngoài ra, nếu một luồng yêu cầu một số tài nguyên, trước tiên chúng ta kiểm tra xem chúng có sẵn hay không. Nếu có thì cấp phát nó. Nếu không, chúng ta kiểm tra xem chúng có được cấp phát cho một số luồng khác đang chờ tài nguyên bổ sung hay không. Nếu có, chúng ta giành các tài nguyên mong muốn từ luồng chờ và phân bổ chúng cho luồng yêu cầu. Nếu các tài nguyên không có sẵn hoặc không được giữ bởi một luồng đang chờ, thì luồng yêu cầu phải đợi. Trong khi chờ đợi, một số tài nguyên của nó có thể được nhường lại cho những luồng khác yêu cầu chúng. Một luồng chỉ có thể được khởi động lại khi nó được cấp phát tài nguyên mới mà nó đang yêu cầu và khôi phục bất kỳ tài nguyên nào đã được nhường trong khi chờ đợi.

Giao thức này thường được áp dụng cho các tài nguyên mà trạng thái của chúng có thể dễ dàng lưu và khôi phục sau này, chẳng hạn như thanh ghi CPU và các giao dịch cơ sở dữ liệu. Nó thường không thể áp dụng cho các tài nguyên như khóa mutex và semaphores (chính xác là loại tài nguyên mà tắc nghẽn thường xảy ra nhất).

7.4.4. Chờ vòng tròn

Ba lựa chọn được trình bày cho đến nay để ngăn chặn tắc nghẽn nói chung là không thực tế trong hầu hết các tình huống. Tuy nhiên, điều kiện thứ tư và cũng là điều kiện cuối cùng cho tắc nghẽn - điều kiện chờ vòng tròn - tạo cơ hội cho một giải pháp thực tế bằng cách làm mất hiệu lực của một trong những điều kiện cần thiết. Một cách để đảm bảo điều kiện này không bao giờ đúng là áp đặt thứ tự tổng thể của tất cả các loại tài nguyên và yêu cầu mỗi luồng yêu cầu tài nguyên theo thứ tự liệt kê ngày càng tăng.

Để minh họa, ta đặt $R = \{R_1, R_2, ..., R_m\}$ là tập các loại tài nguyên. Chúng ta gán cho mỗi loại tài nguyên một số nguyên duy nhất, cho phép so sánh hai tài nguyên và xác định xem tài nguyên này đứng trước tài nguyên khác trong thứ tự của chúng. Về mặt hình thức, chúng ta định nghĩa hàm $F: R \rightarrow N$, trong đó N là tập các số tự nhiên.

Bây giờ xét giao thức sau để ngăn chặn tắc nghẽn: Mỗi luồng chỉ có thể yêu cầu

tài nguyên theo thứ tự liệt kê ngày càng tăng. Có nghĩa là, ban đầu một luồng có thể yêu cầu một phiên bản của tài nguyên - chẳng hạn như R_i. Sau đó, luồng có thể yêu cầu một thể hiện của tài nguyên R_j khi và chỉ khi $F(R_j) > F(R_i)$. Ví dụ: bằng cách sử dụng hàm được định nghĩa ở trên, một luồng muốn sử dụng cả mutex thứ nhất và mutex thứ hai cùng một lúc, trước tiên phải yêu cầu mutex thứ nhất và sau đó là mutex thứ hai. Ngoài ra, chúng ta có thể bắt buộc một luồng yêu cầu một thể hiện của tài nguyên R_j phải giải phóng bất kỳ tài nguyên nào R_i mà sao cho $F(R_i) \geq F(R_j)$. Cũng lưu ý rằng nếu cần một số thể hiện của cùng một loại tài nguyên, thì chỉ đưa ra một yêu cầu duy nhất cho tất cả chúng phải được đưa ra.

Nếu hai giao thức này được sử dụng, thì điều kiện chờ vòng tròn không thể xảy ra. Có thể chứng minh thực tế này bằng cách giả định rằng tồn tại một chờ vòng tròn (chứng minh phản chứng). Đặt tập hợp các luồng tham gia vào vòng chờ là $\{T_0, T_1, ..., T_n\}$, trong đó T_i đang đợi một tài nguyên R_i, được giữ bởi luồng T_{i+1}. T_n đang đợi tài nguyên R_n do T_0 nắm giữ. Sau đó, vì luồng T_{i+1} đang giữ tài nguyên R_i trong khi yêu cầu tài nguyên R_{i+1}, chúng ta phải có $F(R_i) < F(R_{i+1})$ $\forall i$. Nhưng điều kiện này có nghĩa là $F(R_0) < F(R_1) < ... < F(R_n) < F(R_0)$. Dẫn đến, $F(R_0) < F(R_0)$, điều này là không thể. Do đó, không thể có sự chờ đợi vòng tròn.

7.5. Tránh tắc nghẽn

Các thuật toán ngăn chặn tắc nghẽn, như đã thảo luận trong Phần 7.4, ngăn chặn các tắc nghẽn bằng cách đảm bảo rằng ít nhất một trong các điều kiện cần thiết để không thể xảy ra tắc nghẽn. Tuy nhiên, các nhược điểm của việc ngăn chặn tắc nghẽn bằng phương pháp này là hiệu suất sử dụng thiết bị thấp và giảm thông lượng hệ thống.

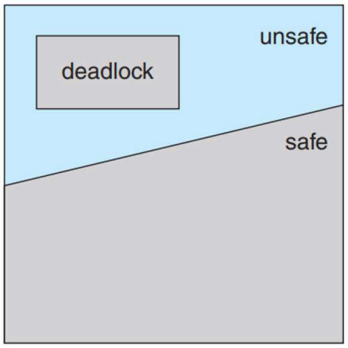
Một phương pháp thay thế để tránh tắc nghẽn là yêu cầu thông tin bổ sung về cách yêu cầu tài nguyên. Ví dụ, trong một hệ thống có tài nguyên R_1 và R_2, hệ thống có thể cần biết rằng luồng P sẽ yêu cầu R_1 đầu tiên và sau đó là R_2 trước khi giải phóng cả hai tài nguyên, trong khi luồng Q sẽ yêu cầu R_2 và sau đó là R_1. Với thông tin về chuỗi hoàn chỉnh các yêu cầu và giải phóng tài nguyên cho mỗi luồng, hệ thống có thể quyết định liệu luồng đó có nên đợi hay không để tránh tình trạng tắc nghẽn có thể xảy ra trong tương lai. Mỗi yêu cầu tài nguyên, hệ thống phải xem xét các tài nguyên hiện có, tài nguyên hiện được phân bổ cho mỗi luồng cũng như các yêu cầu và giải phóng trong tương lai của mỗi luồng.

Các thuật toán khác nhau sử dụng cách tiếp cận này khác nhau về số lượng và loại thông tin được yêu cầu. Mô hình đơn giản và hữu ích nhất yêu cầu mỗi luồng khai báo số lượng tài nguyên tối đa của mỗi loại mà nó có thể cần. Với thông tin tiên nghiệm này,

có thể xây dựng một thuật toán đảm bảo rằng hệ thống sẽ không bao giờ đi vào trạng thái tắc nghẽn. Một thuật toán tránh tắc nghẽn tự động kiểm tra trạng thái cấp phát tài nguyên để đảm bảo rằng điều kiện chờ vòng tròn không bao giờ có thể tồn tại. Trạng thái phân bổ tài nguyên được xác định bởi số lượng tài nguyên có sẵn và được phân bổ cũng như nhu cầu tối đa của các luồng. Trong các phần sau, chúng ta khám phá hai thuật toán tránh tắc nghẽn.

7.5.1. Trạng thái an toàn

Trạng thái là an toàn nếu hệ thống có thể phân bổ tài nguyên cho mỗi luồng (tối đa) theo một số thứ tự mà vẫn tránh được tắc nghẽn. Định nghĩa hình thức, một hệ thống ở trạng thái an toàn nếu tồn tại một chuỗi an toàn. Một chuỗi các luồng $<T_1, T_2, ..., T_n>$ ở trạng thái an toàn nếu đối với mỗi T_i, các yêu cầu tài nguyên mà T_i vẫn có thể được đáp ứng bởi các tài nguyên hiện có cộng với các tài nguyên được giữ bởi tất cả T_j, với $j<i$. Trong tình huống này, nếu tài nguyên mà T_i cần không có sẵn ngay lập tức, thì T_i có thể đợi cho đến khi tất cả các T_j đã hoàn thành. Khi chúng hoàn thành, T_i có thể lấy tất cả các tài nguyên cần thiết, hoàn thành nhiệm vụ được chỉ định, trả lại các tài nguyên đã được phân bổ và kết thúc. Khi T_i kết thúc, T_{i+1} có thể nhận được các tài nguyên cần thiết, v.v. Nếu không có trình tự nào như vậy tồn tại, thì trạng thái hệ thống được cho là không an toàn.



Hình 7-6. *Không gian trạng thái an toàn, không an toàn, tắc nghẽn.*

Trạng thái an toàn không phải là trạng thái tắc nghẽn. Ngược lại, trạng thái tắc nghẽn là trạng thái không an toàn. Tuy nhiên, không phải tất cả các trạng thái không an toàn đều là tắc nghẽn (Hình 7-6). Trạng thái không an toàn có thể dẫn đến tắc nghẽn. Miễn là trạng thái an toàn, hệ điều hành có thể tránh được trạng thái không an toàn (và tắc nghẽn).

Để minh họa, hãy xem xét một hệ thống có 12 tài nguyên và ba luồng: T_0, T_1 và T_2. Luồng T_0 yêu cầu 10 tài nguyên, luồng T_1 có thể cần đến 4 tài nguyên và luồng T_2

có thể cần đến 9 tài nguyên. Giả sử rằng, tại thời điểm t_0, luồng T_0 đang giữ 5 tài nguyên, luồng T_1 đang giữ 2 tài nguyên và luồng T_2 đang giữ 2 tài nguyên. (Do đó, có 3 tài nguyên đang rỗi).

	Nhu cầu tối đa	Hiện tại đang giữ
T_0	10	5
T_1	4	2
T_2	9	2

Tại thời điểm t_0, hệ thống ở trạng thái an toàn. Chuỗi $<T_1, T_0, T_2>$ thỏa mãn điều kiện an toàn. Luồng T_1 ngay lập tức có thể được cấp phát tất cả các tài nguyên của nó và sau đó trả lại chúng (khi đó hệ thống sẽ có 5 tài nguyên khả dụng); thì luồng T_0 có thể lấy tất cả các tài nguyên của nó và trả lại chúng (khi đó hệ thống sẽ có sẵn 10 tài nguyên); và cuối cùng luồng T_2 có thể lấy tất cả các tài nguyên của nó và trả lại chúng (khi đó hệ thống sẽ có sẵn tất cả 12 tài nguyên).

Một hệ thống có thể đi từ trạng thái an toàn sang trạng thái không an toàn. Giả sử rằng, tại thời điểm t_1, luồng T_2 yêu cầu và được cấp phát thêm 1 tài nguyên. Hệ thống không còn ở trạng thái an toàn. Tại thời điểm này, chỉ luồng T_1 có thể được cấp phát tất cả các tài nguyên của nó. Khi nó trả về chúng, hệ thống sẽ chỉ có 4 tài nguyên khả dụng. Vì luồng T_0 đã được cấp phát 5 tài nguyên nhưng có tối đa 10 tài nguyên, nó có thể yêu cầu thêm 5 tài nguyên. Nếu như vậy, nó sẽ phải đợi, vì tài nguyên không có sẵn. Tương tự, luồng T_2 có thể yêu cầu 6 tài nguyên bổ sung và phải chờ, dẫn đến tắc nghẽn. Sai lầm của chúng ta là cấp cho luồng T_2 1 tài nguyên khác. Nếu chúng ta đã bắt T_2 đợi cho đến khi một trong các luồng khác kết thúc và giải phóng tài nguyên của nó, thì chúng ta có thể tránh được tắc nghẽn.

Với khái niệm về trạng thái an toàn, chúng ta có thể xác định các thuật toán tránh để đảm bảo rằng hệ thống sẽ không bao giờ bị tắc nghẽn. Ý tưởng chỉ đơn giản là đảm bảo rằng hệ thống sẽ luôn duy trì ở trạng thái an toàn. Ban đầu, hệ thống ở trạng thái an toàn. Bất cứ khi nào một luồng yêu cầu một tài nguyên hiện đang có sẵn, hệ thống phải quyết định xem liệu tài nguyên đó có thể được cấp phát ngay lập tức hay luồng phải đợi. Yêu cầu chỉ được cấp nếu việc phân bổ khiến hệ thống ở trạng thái an toàn.

7.5.2. Thuật toán đồ thị cấp phát tài nguyên

Nếu có một hệ thống cấp phát chỉ có một thể hiện của mỗi loại tài nguyên, ta có thể sử dụng một biến thể của đồ thị cấp phát tài nguyên được xác định trong Phần 7.3.2 để tránh tắc nghẽn. Ngoài các cạnh yêu cầu và cấp phát đã được mô tả, ta định nghĩa một loại cạnh mới, được gọi là cạnh đăng kí trước. Một cạnh đăng kí trước $T_i \rightarrow R_j$ chỉ ra rằng luồng T_i có thể yêu cầu tài nguyên R_j vào một thời điểm nào đó trong tương lai.

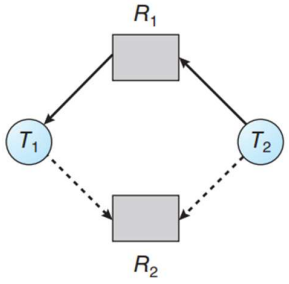
Cạnh này giống cạnh yêu cầu về hướng nhưng được biểu diễn trong biểu đồ bằng một đường nét đứt. Khi luồng T_i yêu cầu tài nguyên R_j, cạnh đăng kí trước $T_i \rightarrow R_j$ được chuyển đổi thành cạnh yêu cầu. Tương tự, khi một tài nguyên R_j được giải phóng bởi T_i, cạnh cấp phát $R_j \rightarrow T_i$ được chuyển đổi lại thành một đăng kí trước $T_i \rightarrow R_j$.

Lưu ý rằng các tài nguyên phải được xác nhận quyền sở hữu trước trong hệ thống. Nghĩa là, trước khi luồng T_i bắt đầu thực thi, tất cả các cạnh đăng kí trước của nó phải xuất hiện trong đồ thị phân bổ tài nguyên. Chúng ta có thể nới lỏng điều kiện này bằng cách chỉ cho phép thêm một cạnh đăng kí trước $T_i \rightarrow R_j$ vào đồ thị nếu tất cả các cạnh được liên kết với luồng T_i là các cạnh đăng kí trước.

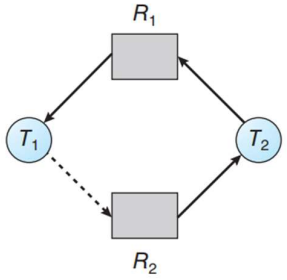
Bây giờ, giả sử rằng luồng T_i yêu cầu tài nguyên R_j. Yêu cầu chỉ có thể được đáp ứng nếu việc chuyển đổi cạnh yêu cầu $T_i \rightarrow R_j$ thành cạnh cấp phát $R_j \rightarrow T_i$ không dẫn đến việc hình thành một chu trình trong đồ thị phân bổ tài nguyên. Ta kiểm tra trạng thái an toàn bằng cách sử dụng thuật toán phát hiện chu trình. Thuật toán để phát hiện một chu trình trong đồ thị có độ phức tạp là $O(n^2)$, trong đó n là số luồng trong hệ thống.

Nếu không có chu trình nào tồn tại, thì việc phân bổ tài nguyên sẽ làm hệ thống ở trạng thái an toàn. Nếu có một chu trình, thì việc phân bổ sẽ đưa hệ thống vào trạng thái không an toàn. Khi đó, luồng T_i sẽ phải đợi các yêu cầu của nó được đáp ứng.

Để minh họa thuật toán này, xét đồ thị phân bổ tài nguyên ở Hình 7-7. Giả sử rằng T_2 yêu cầu R_2. Mặc dù R_2 hiện đang rỗi, nhưng không thể cấp phát nó cho T_2, vì hành động này sẽ tạo ra một chu trình trong đồ thị (Hình 7-8) và hệ thống sẽ ở trạng thái không an toàn. Nếu T_1 yêu cầu R_2 và T_2 yêu cầu R_1, thì xảy ra tắc nghẽn.



Hình 7-7. *Đồ thị cấp phát tài nguyên để tránh tắc nghẽn.*



Hình 7-8. *Trạng thái không an toàn trong đồ thị cấp phát.*

7.5.3. Thuật toán Banker

Thuật toán đồ thị cấp phát tài nguyên không áp dụng được cho hệ thống cấp phát tài nguyên có nhiều thể hiện của mỗi loại tài nguyên. Thuật toán tránh tắc nghẽn sau đây có thể áp dụng cho một hệ thống như vậy nhưng kém hiệu quả hơn so với đồ thị phân bổ tài nguyên. Thuật toán này thường được gọi là thuật toán của chủ ngân hàng (Banker). Tên này được chọn vì thuật toán có thể được sử dụng trong hệ thống ngân hàng để đảm bảo rằng ngân hàng không bao giờ phân bổ tiền mặt sẵn có của mình theo cách mà nó không còn có thể đáp ứng nhu cầu của tất cả khách hàng.

Khi một luồng mới vào hệ thống, nó phải khai báo số thể hiện tối đa của mỗi loại tài nguyên mà nó cần. Con số này không được vượt quá tổng số tài nguyên trong hệ thống. Khi người dùng yêu cầu một tập hợp tài nguyên, hệ thống phải xác định xem việc phân bổ các tài nguyên này có khiến hệ thống ở trạng thái an toàn hay không? Nếu an toàn, các tài nguyên sẽ được cấp phát; nếu không, luồng phải đợi cho đến khi một số luồng khác giải phóng đủ tài nguyên.

Một số cấu trúc dữ liệu được duy trì để cài đặt thuật toán Banker. Các cấu trúc dữ liệu này mã hóa trạng thái của hệ thống cấp phát tài nguyên. Ta cần các cấu trúc dữ liệu sau, trong đó n là số luồng trong hệ thống và m là số loại tài nguyên:

• `Available`: Một vectơ có độ dài `m` cho biết số lượng tài nguyên hiện có của mỗi loại. Nếu `Available[j] = k`, thì `k` thể hiện của loại tài nguyên R_j có sẵn.

• `Max`: Một ma trận `n×m` xác định nhu cầu tối đa của mỗi luồng. Nếu `Max[i][j] = k`, thì luồng T_i có thể yêu cầu tối đa `k` trường hợp của loại tài nguyên R_j.

• `Allocation`: Một ma trận `n×m` xác định số lượng tài nguyên của mỗi loại hiện được phân bổ cho mỗi luồng. Nếu `Allocation[i][j] = k`, thì luồng T_i hiện được cấp phát `k` thể hiện của loại tài nguyên R_j.

• `Need`: Một ma trận `n×m` cho biết nhu cầu tài nguyên còn lại của mỗi luồng. Nếu `Need[i][j] = k`, thì luồng T_i có thể cần thêm `k` thể hiện của loại tài nguyên R_j để hoàn thành nhiệm vụ của nó. Lưu ý rằng `Need[i][j] = Max[i][j] - Allocation[i][j]`.

Các cấu trúc dữ liệu này thay đổi theo thời gian cả về kích thước và giá trị.

Để đơn giản hóa việc trình bày thuật toán Banker, tiếp theo ta định nghĩa một số ký hiệu. Gọi X và Y là các vectơ có độ dài `n`. Ta có $X \leq Y$ khi và chỉ khi $X[i] \leq Y[i]$ $\forall i = 1, 2, ..., n$. Ví dụ, nếu $X = (1, 7, 3, 2)$ và $Y = (0, 3, 2, 1)$, thì $Y \leq X$. Ngoài ra, $Y < X$ nếu $Y \leq X$ và $Y \neq X$.

Ta có thể coi mỗi hàng trong ma trận `Allocation` và `Need` là vectơ và gọi chúng là $\texttt{Allocation}_i$ và $\texttt{Need}_i$. Vector $\texttt{Allocation}_i$ chỉ rõ các tài nguyên hiện được cấp cho luồng T_i; vectơ $\texttt{Need}_i$ chỉ rõ các tài nguyên bổ sung mà luồng T_i vẫn có thể yêu cầu để hoàn thành nhiệm vụ của nó.

a. Thuật toán kiểm tra trạng thái an toàn

Đây là thuật toán kiểm tra liệu hệ thống có ở trạng thái an toàn hay không. Thuật toán này mô tả như sau:

```
1. Gọi Work và Finish lần lượt là các vectơ có độ dài m và n. Khởi
tạo Work = Available và Finish[i] = false ∀i = 0..n-1.
2. Tìm một chỉ số i sao cho cả hai:
           a. Finish[i] == false, và:
           b. Needi ≤ Work.
   Nếu không có i như vậy tồn tại, hãy chuyển sang bước 4.
3. Work = Work + Allocationi
   Finish[i] = true
   Chuyển sang bước 2.
4. Nếu Finish[i] == true ∀i, thì hệ thống đang ở trạng thái an toàn.
```

Thuật toán có độ phức tạp $O(mn^2)$.

b. Thuật toán yêu cầu tài nguyên

Tiếp theo là thuật toán xác định liệu các yêu cầu có thể được cấp phát một cách an toàn hay không? Gọi $\texttt{Request}_i$ là vectơ yêu cầu cho luồng T_i. Nếu $\texttt{Request}_i\texttt{[j] = k}$, nghĩa là luồng T_i muốn k thể hiện của loại tài nguyên R_j. Khi luồng T_i yêu cầu tài nguyên, các hành động sau được thực hiện:

```
1. Nếu Requestesti ≤ Needi, chuyển sang bước 2. Nếu không, phát sinh
lỗi vì luồng yêu cầu vượt quá nhu cầu tối đa của nó.
2. Nếu Requesti ≤ Available, chuyển sang bước 3. Nếu không, Ti phải
đợi, vì tài nguyên không có sẵn.
3. Hệ thống cấp phát thử tài nguyên được yêu cầu đến luồng Ti bằng
cách sửa đổi trạng thái như sau:
           Available = Available – Requestesti
           Allocationi = Allocationi + Requesti
           Needi = Needi – Requestesti
Nếu kết quả trạng thái phân bổ tài nguyên này là an toàn, giao dịch
đã hoàn tất và luồng Ti được cấp phát tài nguyên của nó. Tuy nhiên,
nếu trạng thái mới không an toàn, thì Ti phải đợi Requesti và khôi
phục trạng thái cấp phát tài nguyên cũ.
```

c. Ví dụ minh họa

Để minh họa việc sử dụng thuật toán Banker, xét một hệ thống có 5 luồng từ T_0 đến T_4 và 3 loại tài nguyên A, B và C. Loại tài nguyên A có 10 thể hiện, tài nguyên B có 5 thể hiện và tài nguyên C có 7 thể hiện. Giả sử rằng bảng sau đây biểu diễn trạng thái hiện tại của hệ thống:

Luồng	Allocation			Max			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	5	3	7	4	3			
T_1	2	0	0	3	2	2	1	2	2			
T_2	3	0	2	9	0	2	6	0	0	3	3	2
T_3	2	1	1	2	2	2	0	1	1			
T_4	0	0	2	4	3	3	4	3	1			

Nội dung cột `Need` được định nghĩa là `Max - Allocation`.

Ta thấy hệ thống hiện đang ở trạng thái an toàn. Thật vậy, dãy <T_1, T_3, T_4, T_2, T_0> thỏa mãn các tiêu chí an toàn. Giả sử bây giờ luồng T_1 yêu cầu 1 thể hiện bổ sung của loại tài nguyên A và 2 thể hiện của loại tài nguyên C, vì vậy $\texttt{Request}_1 = (1, 0, 2)$. Để quyết định liệu yêu cầu này có thể được cấp ngay lập tức hay không, trước tiên ta kiểm tra xem $\texttt{Request}_1 \leq \texttt{Available}$, tức là $(1, 0, 2) \leq (3, 3, 2)$, là đúng. Sau đó, ta thử đáp ứng yêu cầu này và dẫn đến trạng thái mới sau:

Luồng	Allocation			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	4	3	2	3	0
T_1	3	0	2	0	2	0			
T_2	3	0	2	6	0	0			
T_3	2	1	1	0	1	1			
T_4	0	0	2	4	3	1			

Ta phải xác định xem trạng thái hệ thống mới này có an toàn hay không, bằng cách thực hiện thuật toán an toàn và nhận thấy rằng chuỗi <T_1, T_3, T_4, T_0, T_2> thỏa mãn yêu cầu an toàn. Do đó có thể đáp ứng ngay lập tức cấp yêu cầu của luồng T_1.

Tuy nhiên khi hệ thống ở trạng thái này, T_4 yêu cầu (3, 3, 0) không thể được cấp phát, vì tài nguyên không có sẵn. Hơn nữa, yêu cầu (0, 2, 0) của T_0 không thể được chấp nhận, ngay cả khi các tài nguyên có sẵn, vì trạng thái kết quả là không an toàn.

7.6. Phát hiện tắc nghẽn

Nếu một hệ thống không sử dụng thuật toán ngăn chặn tắc nghẽn hoặc tránh tắc nghẽn, thì tình huống tắc nghẽn có thể xảy ra. Trong môi trường này, hệ thống có thể

cung cấp:

• Một thuật toán kiểm tra trạng thái của hệ thống để xác định xem có xảy ra tắc nghẽn hay không?

• Một thuật toán để khôi phục từ tắc nghẽn.

Tiếp theo, ta thảo luận về hai yêu cầu này vì chúng liên quan đến các hệ thống chỉ có một thể hiện duy nhất của mỗi loại tài nguyên, cũng như các hệ thống có nhiều thể hiện của mỗi loại tài nguyên.

7.6.1. Kiểu một thể hiện cho mỗi loại tài nguyên

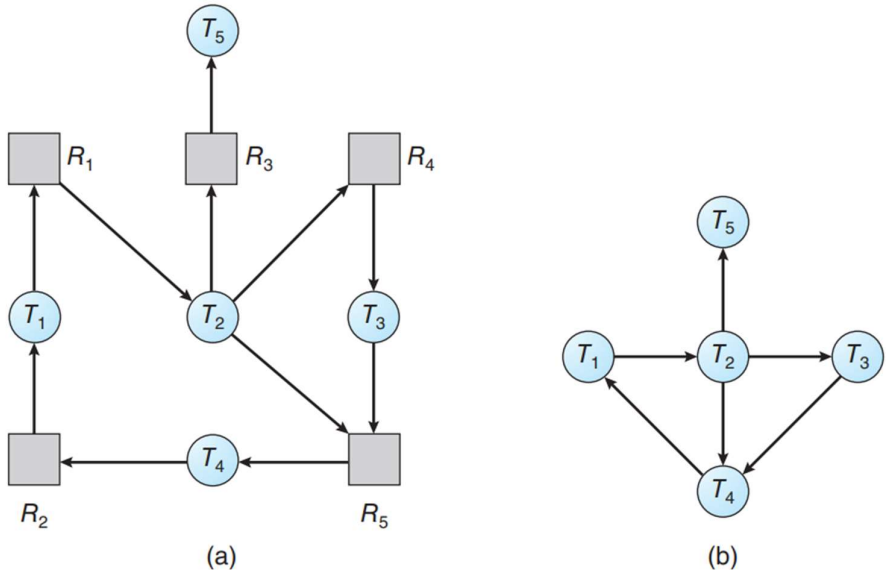
Nếu tất cả các tài nguyên chỉ có một thể hiện duy nhất, thì có thể xác định một thuật toán phát hiện tắc nghẽn sử dụng biến thể của đồ thị phân bổ tài nguyên, được gọi là đồ thị chờ đợi. Đồ thị này nhận được từ đồ thị phân bổ tài nguyên bằng cách loại bỏ các nút tài nguyên và thu gọn các cạnh thích hợp.

Chính xác hơn, một cạnh từ T_i đến T_j trong đồ thị chờ đợi ngụ ý rằng luồng T_i đang chờ luồng T_j giải phóng một tài nguyên mà T_i cần. Một cạnh $T_i \rightarrow T_j$ tồn tại trong đồ thị chờ khi và chỉ khi đồ thị phân bổ tài nguyên tương ứng chứa hai cạnh $T_i \rightarrow R_q$ và $R_q \rightarrow T_j$ đối với tài nguyên R_q. Trong Hình 7-9 trình bày đồ thị phân bổ tài nguyên và đồ thị chờ tương ứng.

Một tắc nghẽn tồn tại trong hệ thống khi và chỉ khi đồ thị chờ chứa một chu trình. Để phát hiện các tắc nghẽn, hệ thống cần duy trì đồ thị chờ và định kỳ gọi thuật toán tìm kiếm một chu trình trong đồ thị. Thuật toán phát hiện một chu trình trong đồ thị có độ phức tạp $O(n^2)$, trong đó n là số đỉnh trong đồ thị.

7.6.2. Kiểu nhiều thể hiện cho một loại tài nguyên

Đồ thị chờ không áp dụng được cho hệ thống phân bổ tài nguyên có nhiều thể hiện của mỗi loại tài nguyên. Bây giờ ta chuyển sang thuật toán phát hiện tắc nghẽn có thể áp dụng cho một hệ thống như vậy. Thuật toán sử dụng một số cấu trúc dữ liệu thay đổi theo thời gian tương tự như cấu trúc được sử dụng trong thuật toán Banker:



Hình 7-9. *(a) Đồ thị cấp phát tài nguyên; (b) Đồ thị chờ tương ứng.*

• `Available`: Một vectơ có độ dài `m` cho biết số lượng tài nguyên hiện có của mỗi loại.

• `Allocation`: ma trận `n×m` xác định số lượng tài nguyên của mỗi loại hiện được phân bổ cho mỗi luồng.

• `Request`: ma trận `n×m` cho biết yêu cầu hiện tại của mỗi luồng. Nếu `Request[i][j]=k`, thì luồng T_i đang yêu cầu thêm `k` phiên bản của loại tài nguyên R_j.

Quan hệ ≤ giữa hai vectơ được định nghĩa như trong mục 7.5.3. Để đơn giản hóa ký hiệu, ta coi các hàng trong ma trận `Allocation` và `Request` là vectơ; và gọi chúng là $\texttt{Allocation}_i$ và $\texttt{Requestest}_i$. Thuật toán phát hiện được mô tả ở đây chỉ đơn giản là kiểm tra mọi trình tự phân bổ có thể có cho các luồng để hoàn thành công việc. So sánh thuật toán này với thuật toán Banker trong mục 7.5.3.

```
1. Đặt Work và Finish là vector có độ dài m và n tương ứng.
   Work = Available
      For i = 0 to n–1 do
           if Allocationi ≠ 0 then
                Finish[i] = false
           else
                Finish[i] = true.
2. Tìm chỉ số i thỏa hai điều kiện:
     a. Finish[i] == false
     b. Requesti ≤ Work
   Nếu không tồn tại số i, chuyển đến bước 4.
```

```
3. Work = Work + Allocationi
   Finish[i] = true
   Chuyển đến bước 2.
4. Nếu tồn tại i mà Finish[i] = false, thì hệ thống ở trạng thái tắc
nghẽn. Hơn nữa nếu Finish[i] = false thì luồng Ti bị khóa.
```

Thuật toán này có độ phức tạp là $O(m.n^2)$.

Chú ý: ta lấy lại tài nguyên của luồng T_i (ở bước 3) ngay sau khi xác định $Requestest_i \leq Work$ (ở bước 2b). Vì T_i hiện không tham gia vào tắc nghẽn (vì $Requestest_i \leq Work$). Vì vậy, giả định rằng T_i sẽ không cần thêm tài nguyên để hoàn thành nhiệm vụ của nó; do đó nó sẽ sớm trả lại tất cả các tài nguyên hiện có. Nếu giả định này là không chính xác, có thể xảy ra tắc nghẽn sau này. Tắc nghẽn đó sẽ được phát hiện vào lần tiếp theo khi gọi thuật toán phát hiện tắc nghẽn.

Để minh họa thuật toán này, xét một hệ thống có 5 luồng từ T_0 đến T_4 và 3 loại tài nguyên A, B và C. Loại tài nguyên A có 7 thể hiện, tài nguyên B có 2 thể hiện và tài nguyên C có 6 thể hiện. Bảng sau biểu diễn cho trạng thái hiện tại của hệ thống:

	Allocation			**Request**			**Available**		
Luồng	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

Ta khẳng định hệ thống không ở trạng thái tắc nghẽn. Thật vậy, nếu thực hiện thuật toán kiểm tra trạng thái tắc nghẽn, ta nhận được chuỗi $<T_0, T_2, T_3, T_1, T_4>$ cho kết quả là $Finish[i]=true \ \forall i$.

Giả sử bây giờ luồng T_2 thực hiện một yêu cầu bổ sung 1 thể hiện kiểu tài nguyên C. Ma trận Request được sửa đổi như sau:

	Request		
Luồng	*A*	*B*	*C*
T_0	0	0	0
T_1	2	0	2
T_2	0	0	1
T_3	1	0	0
T_4	0	0	2

Ta khẳng định hệ thống hiện đã bị tắc nghẽn. Mặc dù có thể lấy lại các tài nguyên do luồng T_0 đang giữ, nhưng số lượng tài nguyên có sẵn không đủ để thực hiện các yêu

cầu của các luồng khác. Do đó, tồn tại một tắc nghẽn, bao gồm các luồng T_1, T_2, T_3, T_4.

7.6.3. Cách sử dụng thuật toán phát hiện tắc nghẽn

Khi nào ta nên gọi thuật toán phát hiện tắc nghẽn? Câu trả lời phụ thuộc vào hai yếu tố sau:

1. Bao lâu thì một tắc nghẽn có khả năng xảy ra?

2. Có bao nhiêu luồng sẽ bị ảnh hưởng bởi tắc nghẽn khi nó xảy ra?

Nếu tắc nghẽn xảy ra thường xuyên, thì thuật toán phát hiện nên được gọi thường xuyên. Tài nguyên được phân bổ cho các luồng bị khóa sẽ không hoạt động cho đến khi tắc nghẽn bị phá vỡ. Hơn nữa, số lượng luồng tham gia vào chu trình tắc nghẽn có thể tăng lên.

Các tắc nghẽn chỉ xảy ra khi một số luồng đưa một yêu cầu không thể được cấp phát ngay lập tức. Yêu cầu này có thể là yêu cầu cuối cùng hoàn thành một chuỗi các luồng đang chờ. Do đó, ta có thể sử dụng thuật toán phát hiện tắc nghẽn mỗi khi một yêu cầu cấp phát không thể được đáp ứng ngay lập tức. Trong trường hợp này, có thể xác định không chỉ tập hợp các luồng bị khóa mà còn cả các luồng cụ thể đã "gây ra" tắc nghẽn đó. (Trong thực tế, mỗi luồng bị khóa là một liên kết chu trình trong biểu đồ tài nguyên, vì vậy tất cả chúng, cùng gây ra tắc nghẽn.) Nếu có nhiều loại tài nguyên khác nhau, một yêu cầu có thể tạo ra nhiều chu trình trong biểu đồ tài nguyên, mỗi chu trình được hoàn thành bởi yêu cầu gần đây nhất và "gây ra" bởi một luồng có thể nhận dạng.

Tất nhiên, việc sử dụng thuật toán phát hiện tắc nghẽn cho mọi yêu cầu tài nguyên sẽ phát sinh chi phí đáng kể thời gian tính toán. Một giải pháp thay thế ít tốn kém hơn là gọi thuật toán vào những khoảng thời gian xác định - ví dụ: một lần mỗi giờ hoặc bất cứ khi nào mức sử dụng CPU giảm xuống dưới 40 phần trăm. (Tắc nghẽn cuối cùng làm tê liệt thông lượng của hệ thống và khiến hiệu suất sử dụng CPU giảm xuống.) Nếu thuật toán phát hiện được gọi vào các thời điểm tùy ý, đồ thị tài nguyên có thể chứa nhiều chu trình. Trong trường hợp này thường không thể biết được luồng nào trong số nhiều luồng bị khóa đã "gây ra" sự tắc nghẽn.

7.7. Phục hồi trạng thái không tắc nghẽn

Khi một thuật toán phát hiện xác định có một tắc nghẽn tồn tại, một số giải pháp để khắc phục. Một khả năng là thông báo cho người vận hành biết một tắc nghẽn đã xảy ra và để người vận hành xử lý tắc nghẽn theo cách thủ công. Một khả năng khác là để hệ thống tự động khôi phục từ trạng thái tắc nghẽn. Có hai lựa chọn để phá vỡ tắc nghẽn.

Một, đơn giản là hủy bỏ một hoặc nhiều luồng để phá vỡ chờ vòng tròn. Cách khác, là lấy một số tài nguyên từ một hoặc nhiều luồng bị khóa.

7.7.1. Kết thúc tiến trình và luồng

Để loại bỏ các tắc nghẽn bằng cách hủy bỏ một tiến trình hoặc một luồng, ta sử dụng một trong hai phương pháp. Trong cả hai phương pháp, hệ thống thu hồi tất cả các tài nguyên được phân bổ cho các tiến trình đã kết thúc.

• Hủy bỏ tất cả các tiến trình bị khóa. Phương pháp này rõ ràng sẽ phá vỡ chu trình tắc nghẽn, nhưng phải trả giá rất đắt. Các tiến trình bị khóa có thể đã được tính toán trong một thời gian dài, và kết quả của các tính toán từng phần này phải bị loại bỏ và có thể sẽ phải tính toán lại sau này.

• Hủy bỏ một tiến trình tại một thời điểm cho đến khi chu trình tắc nghẽn được loại bỏ. Phương pháp này phát sinh chi phí đáng kể, vì sau khi mỗi tiến trình bị hủy bỏ, một thuật toán phát hiện tắc nghẽn phải được sử dụng để xác định xem có bất kỳ tiến trình nào vẫn bị khóa không.

Việc hủy bỏ một tiến trình có thể không dễ dàng. Nếu tiến trình đang trong quá trình cập nhật một tệp, việc chấm dứt nó có thể khiến tệp đó ở trạng thái không chính xác. Tương tự, nếu tiến trình đang trong quá trình cập nhật dữ liệu được chia sẻ trong khi giữ khóa mutex, hệ thống phải khôi phục trạng thái của khóa như khả dụng, mặc dù không có đảm bảo nào về tính toàn vẹn của dữ liệu được chia sẻ.

Nếu phương pháp chấm dứt từng phần được sử dụng, thì chúng ta phải xác định tiến trình (hoặc các tiến trình) bị khóa nào nên được chấm dứt. Quyết định này là một quyết định chính sách, tương tự như các quyết định lập lịch CPU. Câu hỏi cơ bản là một câu hỏi kinh tế; chúng ta nên hủy bỏ những tiến trình mà việc chấm dứt sẽ phải chịu chi phí tối thiểu. Thật không may, thuật ngữ chi phí tối thiểu không phải là một thuật ngữ chính xác. Nhiều yếu tố có thể ảnh hưởng đến tiến trình được chọn, bao gồm:

1. Mức độ ưu tiên của tiến trình là gì.

2. Tiến trình đã tính toán trong bao lâu và tiến trình sẽ tính toán bao lâu trước khi hoàn thành nhiệm vụ được chỉ định.

3. Tiến trình đã sử dụng bao nhiêu và loại tài nguyên nào (ví dụ: liệu các tài nguyên đó có đơn giản không để nhường quyền ưu tiên).

4. Tiến trình cần thêm bao nhiêu tài nguyên để hoàn thành.

5. Có bao nhiêu tiến trình sẽ cần được kết thúc.

7.7.2. Nhường tài nguyên

Để loại bỏ các tắc nghẽn bằng cách sử dụng quyền ưu tiên tài nguyên, ta liên tiếp loại bỏ một số tài nguyên từ các tiến trình và cung cấp các tài nguyên này cho các tiến trình khác cho đến khi chu trình tắc nghẽn bị phá vỡ.

Nếu cần phải có quyền ưu tiên tài nguyên để giải quyết các tắc nghẽn, thì cần giải quyết ba vấn đề:

1. Lựa chọn nạn nhân. Những tài nguyên nào và những tiến trình nào cần nhường ưu tiên? Như trong kết thúc tiến trình, ta phải xác định thứ tự ưu tiên để giảm thiểu chi phí. Các yếu tố chi phí có thể bao gồm các thông số như số lượng tài nguyên mà một tiến trình bị tắc nghẽn đang nắm giữ và lượng thời gian mà tiến trình đã tiêu tốn cho đến nay.

2. Hoàn vốn. Nếu ta loại trừ một tài nguyên khỏi một tiến trình, thì nên làm gì với tiến trình đó? Rõ ràng, nó không thể tiếp tục với quá trình thực hiện bình thường của nó; nó đang thiếu một số tài nguyên cần thiết. Ta phải khôi phục tiến trình về trạng thái an toàn nào đó và khởi động lại từ trạng thái đó. Nói chung, rất khó để xác định trạng thái an toàn là gì, giải pháp đơn giản nhất là khôi phục hoàn toàn: hủy bỏ tiến trình và sau đó khởi động lại nó. Mặc dù sẽ hiệu quả hơn nếu chỉ quay lại tiến trình trong chừng mực cần thiết để phá vỡ tắc nghẽn, nhưng phương pháp này yêu cầu hệ thống phải giữ thêm thông tin về trạng thái của tất cả các tiến trình đang chạy.

3. Bỏ đói. Làm thế nào để ta đảm bảo rằng bỏ đói sẽ không xảy ra? Đó là, làm thế nào ta có thể đảm bảo rằng các tài nguyên sẽ không phải lúc nào cũng được nhường từ tiến trình đó? Trong một hệ thống mà việc lựa chọn nạn nhân chủ yếu dựa trên các yếu tố chi phí, có thể xảy ra tiến trình tương tự luôn được chọn làm nạn nhân. Kết quả là, tiến trình này không bao giờ hoàn thành nhiệm vụ được chỉ định của nó, một tình huống chết đói mà bất kỳ hệ thống thực tế nào cũng phải giải quyết. Rõ ràng, chúng ta phải đảm bảo rằng một tiến trình có thể được chọn làm nạn nhân chỉ một (nhỏ) số lần hữu hạn. Giải pháp phổ biến nhất là bao gồm số lần hoàn vốn trong yếu tố chi phí.

7.8. Tóm tắt

• Tắc nghẽn xảy ra trong một tập hợp các tiến trình khi mọi tiến trình trong tập hợp đang chờ một sự kiện chỉ có thể do một tiến trình khác trong tập hợp gây ra.

• Có bốn điều kiện cần thiết cho tắc nghẽn: (1) loại trừ lẫn nhau, (2) giữ và chờ, (3) không có quyền ưu tiên (độc quyền), và (4) chờ vòng tròn. Chốt lại chỉ có thể xảy ra khi có đủ cả 4 điều kiện.

• Các tắc nghẽn có thể được mô hình hóa bằng các đồ thị phân bổ tài nguyên, trong đó một chu trình biểu thị tắc nghẽn.

• Có thể ngăn chặn tắc nghẽn bằng cách đảm bảo rằng một trong bốn điều kiện cần thiết để tắc nghẽn không thể xảy ra. Trong bốn điều kiện cần, loại bỏ sự chờ vòng tròn là cách tiếp cận thực tế duy nhất.

• Có thể tránh được tắc nghẽn bằng cách sử dụng thuật toán Banker, thuật toán này không cấp tài nguyên nếu như việc cấp như vậy sẽ dẫn hệ thống vào trạng thái không an toàn, nơi có thể xảy ra tắc nghẽn.

• Một thuật toán phát hiện tắc nghẽn có thể đánh giá các tiến trình và tài nguyên trên một hệ thống đang chạy để xác định xem một tập các tiến trình có ở trạng thái bị khóa hay không.

• Nếu tắc nghẽn xảy ra, hệ thống có thể cố gắng khôi phục từ tắc nghẽn bằng cách hủy bỏ một trong các tiến trình trong chờ vòng tròn hoặc lấy lại các tài nguyên đã được gán cho một tiến trình đã khóa.

7.9. Câu hỏi ôn tập

Câu 7.1. Liệt kê ba ví dụ về các trường hợp tắc nghẽn không liên quan đến môi trường hệ thống máy tính.

Câu 7.2. Giả sử rằng một hệ thống đang ở trạng thái không an toàn. Cho thấy rằng các luồng có thể hoàn thành quá trình thực thi của chúng mà không đi vào trạng thái bị khóa.

Câu 7.3. Xét một trạng thái sau của hệ thống:

	Allocation				**Max**				**Available**			
Luồng	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
T_0	0	0	1	2	0	0	1	2				
T_1	1	0	0	0	1	7	5	0				
T_2	1	3	5	4	2	3	5	6	1	5	2	0
T_3	0	6	3	2	0	6	5	2				
T_4	0	0	1	4	0	6	5	6				

a. Nội dung của ma trận Need là gì ?

b. Hệ thống có ở trạng thái an toàn hay không?

c. Giả sử luồng T_1 yêu cầu (0, 4, 2, 0), yêu cầu này có thể được đáp ứng ngay lập tức hay không?