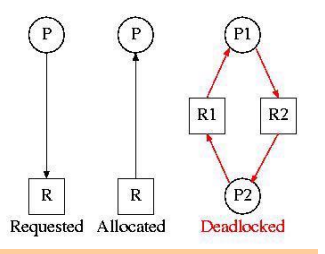
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ ĐÔNG Á
EAST ASIA UNIVERSITY OF TECHNOLOGY

Chương 4

4.3 Tắc nghẽn (Deadlocks)



Nội dung:

- Mô hình hệ thống
- Mô tả tắc nghẽn
- Các phương pháp xử lý tắc nghẽn
- Ngăn ngừa tắc nghẽn
- Tránh khỏi tắc nghẽn
- Phát hiện tắc nghẽn
- Phục hồi từ tắc nghẽn
- Phương pháp kết hợp xử lý tắc nghẽn

Mục tiêu

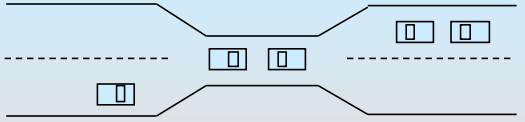
- Mô tả tắc nghẽn, tình trạng ngăn cản các tiến trình đồng thời hoàn thành tác vụ
- Giới thiệu các phương pháp khác nhau để ngăn ngừa hoặc tránh khỏi tắc nghẽn trong một hệ thống máy tính.

Vấn đề tắc nghẽn (Deadlock)

- Trong môi trường đa chương trình, một số tiến trình có thể tranh nhau một số tài nguyên hạn chế.
- Một tiến trình yêu cầu các tài nguyên, nếu tài nguyên không thể đáp ứng tại thời điểm đó thì tiến trình sẽ chuyển sang trạng thái chờ.
- Các tiến trình chờ có thể sẽ không bao giờ thay đổi lại trạng thái được vì các tài nguyên mà nó yêu cầu bị giữ bởi các tiến trình chờ khác.

⇒ ví dụ: tắc nghẽn trên cầu

Ví dụ qua cầu



- Hai (hay nhiều hơn) ô tô đối đầu nhau trên một cây cầu hẹp chỉ đủ độ rộng cho một chiếc.
- Mỗi đoạn của cây cầu có thể xem như một tài nguyên.
- Nếu tắc nghẽn xuất hiện, nó có thể được giải quyết nếu một hay một số ô tô lùi lại nhường đường rồi tiến sau.

4.3.1. Mô hình hệ thống

- Các loại tài nguyên $R_1, R_2, \ldots, R_m$

  Các chu kỳ CPU, không gian bộ nhớ, các tệp, các thiết bị vào-ra
- Mỗi loại tài nguyên R_i có W_i cá thể (instance).
  - vd: hệ thống có 2 CPU, có 5 máy in

  $\Rightarrow$ có thể đáp ứng yêu cầu của nhiều tiến trình hơn
- Mỗi tiến trình sử dụng tài nguyên theo các bước sau:
  1. **yêu cầu** (request): nếu yêu cầu không được giải quyết ngay (vd khi tài nguyên đang được tiến trình khác sử dụng) thì tiến trình yêu cầu phải đợi cho đến khi nhận được tài nguyên.
  2. **sử dụng** (use)
  3. **giải phóng** (release)

4.3.2. Mô tả tắc nghẽn

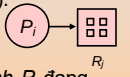
tắc nghẽn có thể xảy ra nếu 4 điều kiện sau đồng thời tồn tại:

- **Ngăn chặn lẫn nhau:** tại một thời điểm, chỉ một tiến trình có thể sử dụng một tài nguyên.
- **Giữ và đợi:** một tiến trình đang giữ ít nhất một tài nguyên và đợi để nhận được tài nguyên khác đang được giữ bởi tiến trình khác.
- **Không có ưu tiên:** một tài nguyên chỉ có thể được tiến trình (tự nguyện!) giải phóng khi nó đã hoàn thành công việc.
- **Chờ đợi vòng tròn:** tồn tại một tập các tiến trình chờ đợi $\{P_0, P_1, \ldots, P_n, P_0\}$
  - P_0 đang đợi tài nguyên bị giữ bởi P_1,
  - P_1 đang đợi tài nguyên bị giữ bởi P_2, **...**
  - P_{n-1} đang đợi tài nguyên bị giữ bởi P_n,
  - và P_n đang đợi tài nguyên bị giữ bởi P_0.

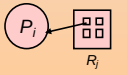
Biểu đồ phân phối tài nguyên

Một tập các đỉnh V và một tập các cạnh E.

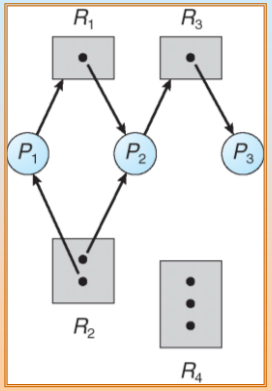
- V được chia thành 2 loại:
  - $P = \{P_1, P_2, \ldots, P_n\}$, tập tất cả các tiến trình.
  - $R = \{R_1, R_2, \ldots, R_m\}$, tập các loại tài nguyên.
    - Mỗi cá thể là một hình vuông bên trong
- cạnh yêu cầu – cạnh có hướng $P_i \rightarrow R_j$. *(tiến trình P_i đang đợi nhận một hay nhiều cá thể của tài nguyên R_j).*



- cạnh chỉ định – cạnh có hướng $R_j \rightarrow P_i$. *(tiến trình P_i đang giữ một hay nhiều cá thể của tài nguyên R_j).*



Vd đồ thị phân phối tài nguyên không chu trình

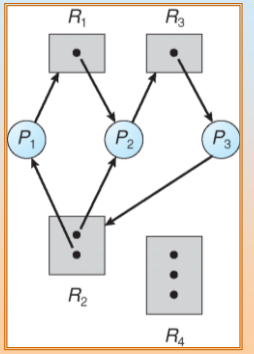


Nếu đồ thị không chu trình thì sẽ không có tiến trình nào bị tắc nghẽn

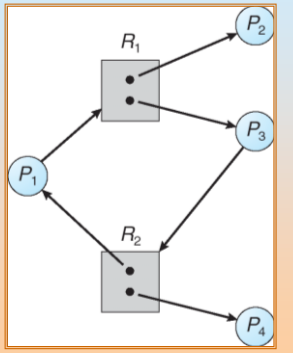
Nếu đồ thị có chu trình thì ***có thể*** tồn tại tắc nghẽn

Vd đồ thị phân phối tài nguyên có chu trình

tắc nghẽn



Không tắc nghẽn: P_4 hoặc P_2 có thể kết thúc, khiến P_1 và P_3 kết thúc được.



Kết luận về đồ thị

- Nếu đồ thị không chu trình
  ⇒ không xảy ra tắc nghẽn.
- Nếu đồ thị có chu trình ⇒
  - nếu mỗi loại tài nguyên chỉ một cá thể thì ***chắc chắn*** xảy ra tắc nghẽn.
  - nếu mỗi loại tài nguyên có một vài cá thể thì ***có thể*** xảy ra tắc nghẽn.

4.3.3. Các phương pháp xử lý tắc nghẽn

- Sử dụng một phương thức để ngăn ngừa hoặc tránh xa, đảm bảo rằng hệ thống sẽ không bao giờ đi vào trạng thái tắc nghẽn.
- Cho phép hệ thống đi vào trạng thái tắc nghẽn rồi khôi phục lại.
- Bỏ qua vấn đề này và vờ như tắc nghẽn không bao giờ xuất hiện trong hệ thống. Giải pháp này được sử dụng trong hầu hết các HĐH, bao gồm cả UNIX.

4.3.4. Ngăn ngừa tắc nghẽn

Ngăn cản các cách tạo yêu cầu: đảm bảo ít nhất một trong bốn điều kiện không thể xuất hiện

- **Ngăn cản lẫn nhau** – đảm bảo là hệ thống không có các file không thể chia sẻ.
  - một tiến trình không bao giờ phải chờ tài nguyên có thể chia sẻ
    - vd: read-only files
  - một số tài nguyên là không thể chia sẻ
    - vd: chế độ toàn màn hình
- **Giữ và đợi** – phải đảm bảo rằng mỗi khi một tiến trình yêu cầu một tài nguyên, nó không giữ bất kỳ tài nguyên nào khác.
  - Đòi hỏi tiến trình yêu cầu và được phân phối tất cả các tài nguyên của nó trước khi nó bắt đầu thực hiện, hoặc chỉ cho phép tiến trình yêu cầu các tài nguyên khi nó không giữ tài nguyên nào cả.
  - Hiệu quả sử dụng tài nguyên thấp, có thể xảy ra starvation.

Ngăn ngừa tắc nghẽn (tiếp)

- **Không có ưu tiên**
  - Nếu một tiến trình đang giữ một số tài nguyên và yêu cầu tài nguyên khác mà không thể được phân phối ngay cho nó thì tất cả các tài nguyên nó đang giữ được giải phóng.
  - Các tài nguyên ưu tiên được thêm vào danh sách tài nguyên dành cho tiến trình đang chờ đợi.
  - Tiến trình sẽ được khởi động lại chỉ khi nó có thể lấy lại các tài nguyên cũ của nó cũng như các tài nguyên mới mà nó đang yêu cầu.
- **Chờ đợi vòng tròn** – áp dụng một thứ tự tuyệt đối cho tất cả các loại tài nguyên: mỗi loại được gắn một số nguyên
  - mỗi tiến trình yêu cầu các tài nguyên theo thứ tự tăng dần: chỉ có thể nhận được tài nguyên có trọng số cao hơn của bất kỳ tài nguyên nào nó đang giữ
  - $\Rightarrow$ Muốn có tài nguyên i, tiến trình phải giải phóng tất cả các tài nguyên có trọng số $j > i$ (nếu có)

4.3.5. Tránh khỏi tắc nghẽn

Yêu cầu HĐH phải có một số thông tin ưu tiên

- Mô hình hữu dụng nhất và đơn giản nhất yêu cầu mỗi tiến trình công bố *số lượng tài nguyên lớn nhất* của mỗi loại mà nó có thể cần đến.
- Giải thuật tránh tắc nghẽn luôn kiểm tra trạng thái phân phối tài nguyên để đảm bảo rằng sẽ không bao giờ có tình trạng chờ đợi vòng tròn.
- Trạng thái phân phối tài nguyên được xác định bởi số tài nguyên khả dụng và đã được phân phối cũng như sự yêu cầu tối đa từ các tiến trình.

4.3.5.1. Safe State

- Một trạng thái là an toàn nếu hệ thống có thể phân phối các tài nguyên cho mỗi tiến trình mà vẫn tránh được tắc nghẽn.
- Khi một tiến trình yêu cầu một tài nguyên còn rỗi, hệ thống phải quyết định liệu phân phối ngay lập tức có làm cho hệ thống mất an toàn hay không?
- Hệ thống ở trong trạng thái an toàn nếu tồn tại một chuỗi an toàn của tất cả các tiến trình.

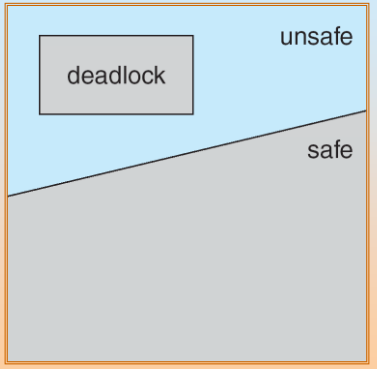
Safe State (tiếp)

- Chuỗi <P1, P2, …, Pn> là an toàn nếu với mỗi Pi, tài nguyên mà nó yêu cầu có thể được cung cấp bởi tài nguyên khả dụng hiện tại và các tài nguyên đang được giữ bởi Pj, với j<i.
  - Nếu tài nguyên Pi cần đang bị Pj giữ thì nó có thể đợi cho đến khi tất cả các Pj kết thúc.
  - Khi Pj kết thúc, Pi có thể giành được các tài nguyên cần thiết, thực hiện, rồi trả lại các tài nguyên đó và kết thúc.
  - Khi Pi kết thúc, P(i+1) có thể giành được tài nguyên cần thiết, …

Safe State: thực tế dễ nhận

- Nếu hệ thống ở trạng thái an toàn ⇒ không có tắc nghẽn.
- Nếu hệ thống ở trạng thái không an toàn ⇒ có thể có tắc nghẽn.
- Sự tránh khỏi tắc nghẽn ⇒ đảm bảo rằng hệ thống sẽ không bao giờ bước vào trạng thái không an toàn.
  - Mỗi loại tài nguyên có một cá thể: giải thuật đồ thị phân phối tài nguyên.
  - Mỗi loại tài nguyên có nhiều cá thể: giải thuật chủ nhà băng.

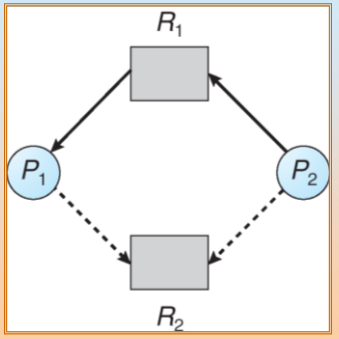
Safe, Unsafe, Deadlock State



4.3.5.2. Giải thuật đồ thị phân phối tài nguyên

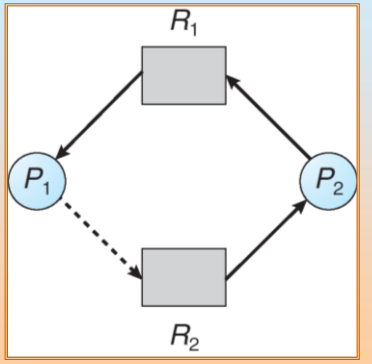
- *Cạnh muốn yêu cầu* (*claim edge*) $P_i \rightarrow R_j$: tiến trình P_j có thể yêu cầu tài nguyên R_j; được biểu diễn bởi một đường đứt nét.
- Cạnh muốn yêu cầu biến thành cạnh yêu cầu (request edge) khi một tiến trình yêu cầu một tài nguyên.
- Khi tài nguyên được một tiến trình giải phóng, cạnh yêu cầu trở lại thành cạnh muốn yêu cầu.
- Hệ thống ở trong trạng thái an toàn miễn là đồ thị không chứa chu trình nào.
  - Chúng ta coi các cạnh muốn yêu cầu như là các cạnh yêu cầu

Giải thuật đồ thị phân phối tài nguyên tránh khỏi tắc nghẽn



Giả sử P_2 yêu cầu R_2. Dù R_2 vẫn đang tự do, chúng ta vẫn không thể phân phối nó cho P_2 vì sẽ tạo ra một chu trình → hệ thống trong trạng thái không an toàn. Nếu P_1 yêu cầu R_2 và P_2 yêu cầu R_1 thì tắc nghẽn sẽ xuất hiện.

Trạng thái không an toàn trong đồ thị phân phối tài nguyên



4.3.5.3. Giải thuật chủ nhà băng

- Có tên như trên vì giải thuật này có thể được sử dụng trong hệ thống nhà băng để đảm bảo rằng nhà băng không bao giờ phân phối quá số tiền khả dụng của nó đến mức mà nó có thể thỏa mãn mọi yêu cầu từ các khách hàng.
- Khi một tiến trình mới đi vào hệ thống, nó phải khai báo số lượng tối đa cá thể của mỗi loại tài nguyên mà nó có thể cần đến. Số này có thể vượt quá tổng tài nguyên trong hệ thống.
- Khi user yêu cầu tài nguyên, hệ thống phải xác định liệu sự phân phối có giữ hệ thống trong trạng thái an toàn không:
  - Nếu có → phân phối tài nguyên
  - Nếu không → tiến trình phải chờ đến khi các tiến trình khác giải phóng đủ tài nguyên.

Giải thuật chủ nhà băng: cấu trúc dữ liệu

- ***n*** = số tiến trình.
- ***m*** = số loại tài nguyên.
- ***Available***: vectơ độ dài *m* – các tài nguyên khả dụng của mỗi loại.
  - Available[*j*] = *k*: có *k* cá thể của loại tài nguyên R_j là khả dụng.
- ***Max***: *ma trận n x m:* xác định số tối đa yêu cầu của mỗi tiến trình.
  - *Max*[*i,j*] = *k*: tiến trình P_i có thể yêu cầu tối đa tối đa k cá thể của loại tài nguyên R_j.
- ***Allocation***: *ma trận n* x *m:* xác định số tài nguyên mỗi loại hiện đang phân phối cho mỗi tiến trình.
  - Allocation[*i,j*] = *k*: P_i hiện đang được phân phối *k* cá thể của R_j.
- ***Need***: *ma trận n* x *m: xác định số tài nguyên còn thiếu cho mỗi tiến trình*
  - *Need*[*i,j*] = *k*: P_i có thể cần *k* cá thể nữa của R_j :

    Need[*i,j*] = *Max*[*i,j*] - *Allocation* [*i,j*].

Giải thuật chủ nhà băng: Kiểm tra an toàn

Tư tưởng: chúng ta tìm một chuỗi an toàn. Nếu tìm được, trạng thái là an toàn, trái lại trạng thái là không an toàn.

1. Gán *Work* và *Finish* là các vectơ độ dài m và n. Khởi tạo:
   Work := *Available*
   Finish[*i*] := (*Allocation*$_i$ == 0) với i = 1,2, …, n.
2. Tìm i thỏa mãn cả 2 điều kiện:
   (a) *Finish*[*i*] = *false* và
   (b) *Need*$_i \le$ *Work*
   Nếu không có i như vậy, nhảy đến bước 4.
   Tìm P_i chưa kết thúc và có nhu cầu không vượt quá khả dụng, nếu có thì phân phối tài nguyên cho nó.
3. *Work* := *Work* + *Allocation*$_i$
   Finish[*i*] := *true*
   nhảy đến bước 2.
   Lượng khả dụng được cộng thêm số tài nguyên đã phân phối cho P_i vì P_i đã có đủ tài nguyên để thực hiện rồi kết thúc
4. Nếu *Finish*[*i*] = true với $\forall$i thì hệ thống ở trạng thái an toàn.

Giải thuật yêu cầu tài nguyên cho tiến trình P_i

Request = vectơ yêu cầu cho tiến trình P_i. Nếu *Request*$_i$ [*j*] = *k* thì tiến trình P_i muốn *k* cá thể của loại tài nguyên R_j.

1. Nếu *Request*$_i \le$ *Need*$_i$, chuyển sang bước 2. Trái lại, dựng lên trạng thái lỗi vì tiến trình đã vượt quá yêu cầu tối đa của nó.
2. Nếu *Request*$_i \le$ *Available*, chuyển sang bước 3. Trái lại P_i phải đợi vì tài nguyên chưa sẵn sàng.
3. Giả vờ phân phối các tài nguyên cho P_i bằng cách sửa trạng thái như sau:

   Available = *Available* – *Request*$_i$;

   Allocation$_i$ = *Allocation*$_i$ + *Request*$_i$;

   Need$_i$ = *Need*$_i$ – *Request*$_i$;

   - *Nếu an toàn* $\Rightarrow$ *phân phối tài nguyên cho* P_i.
   - *Nếu không an toàn* $\Rightarrow$ P_i *phải đợi, và trạng thái phân phối tài nguyên cũ được phục hồi.*

Ví dụ giải thuật chủ nhà băng

- 5 tiến trình P_0 … P_4;
  3 loại tài nguyên A (10 cá thể), B (5 cá thể), và C (7 cá thể).
- Giả sử tại thời điểm T_0:

= Max - Allocation

	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2
P_1	3 2 2	2 0 0	1 2 2	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

= 10 – (0+2+3+2+0)
= W_A - Σ Allocation$_A$

- Hệ thống có đang ở trạng thái an toàn?

Ví dụ (tiếp)

- Áp dụng giải thuật kiểm tra an toàn:

1. Work = Available, Finish[i] = false với mọi i vì Allocation[i] $\neq$ 0
2. Tìm thấy i=1 thỏa mãn Need$_1$ (1 2 2) $\le$ Work (3 3 2)
3. Work = Work + Allocation$_1$ = (3 3 2) + (2 0 0) = (5 3 2)
   Finish[1] = true (đánh dấu tiến trình P1 kết thúc được)
2. Tìm thấy i=3 thỏa mãn Need$_3$ (0 1 1) $\le$ Work (5 3 2)
3. Work = Work + Allocation$_3$ = (5 3 2) + (2 1 1) = (7 4 3)
   Finish[3] = true

   ……..

- Hệ thống đang ở trạng thái an toàn vì chuỗi <P_1, P_3, P_4, P_2, P_0> thỏa mãn các điều kiện an toàn.

Ví dụ P_1 yêu cầu (1,0,2)

- Kiểm tra Request ≤ $Need_1$ ↔ (1,0,2) ≤ (1,2,2) ⇒ true.
- Kiểm tra Request ≤ Available ↔ (1,0,2) ≤ (3,3,2) ⇒ true. Giả vờ đáp ứng yêu cầu, hệ thống sẽ đến trạng thái sau:

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Việc thực hiện giải thuật kiểm tra an toàn cho thấy chuỗi <P1, P3, P4, P0, P2> vẫn thỏa mãn các yêu cầu an toàn → có thể chấp nhận ngay yêu cầu từ P1
- Có thể chấp nhận yêu cầu (3,3,0) từ P4?
- Có thể chấp nhận yêu cầu (0,2,0) từ P0?

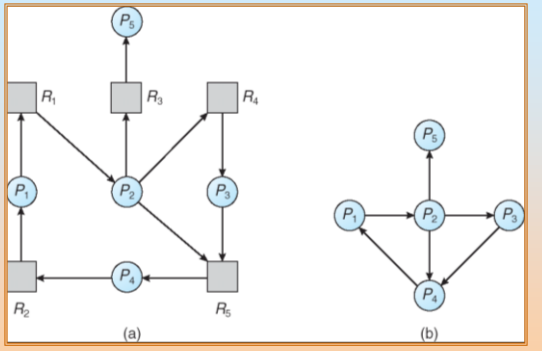
4.3.6. Phát hiện tắc nghẽn

- Nếu một hệ thống không thể thực hiện được việc ngăn ngừa hay tránh xa tắc nghẽn thì tắc nghẽn có thể xuất hiện. Trong môi trường này, hệ thống phải cung cấp:
  - Giải thuật phát hiện tắc nghẽn
  - Giải thuật phục hồi từ tắc nghẽn

Mỗi loại tài nguyên có 1 cá thể

- Khi tất cả tài nguyên chỉ có một cá thể, giải thuật xác định tắc nghẽn sử dụng một biến thể của đồ thị phân phối tài nguyên, bằng cách bỏ đi các nút của loại tài nguyên và bỏ đi các cạnh thích hợp ⇒ đồ thị *wait-for*
  - Các nút là các tiến trình.
  - $P_i \rightarrow P_j$ nếu P_i đang đợi P_j.
- Định kỳ sử dụng giải thuật tìm kiếm chu trình trong đồ thị. Giải thuật đó đòi hỏi n^2 phép toán, với n là số đỉnh trong đồ thị: có chu trình → có thể có tắc nghẽn.

Resource-Allocation Graph and Wait-for Graph



Đồ thị phân phối tài nguyên Đồ thị wait-for tương ứng

Mỗi loại tài nguyên có nhiều cá thể

- *Available:* vectơ độ dài *m* xác định số tài nguyên khả dụng của mỗi loại.
- *Allocation:* ma trận *n* x *m* xác định các tài nguyên của mỗi loại hiện đang được phân phối cho mỗi tiến trình.
- *Request:* ma trận *n* x *m* xác định yêu cầu hiện tại của mỗi tiến trình. Nếu *Request* [*i,j*] = *k*, thì tiến trình P_i đang yêu cầu *k* cá thể nữa của loại tài nguyên R_j.

Giải thuật phát hiện tắc nghẽn

1. Gán *Work* và *Finish* là các vectơ độ dài *m* và *n*. Khởi tạo:
   (a) *Work* := *Available*
   (b) For *i* := 1 to n do If $Allocation_i \neq 0$ then *Finish*[i] := false
   Else *Finish*[i] := *true*
2. Tìm chỉ số *i* thỏa mãn cả 2 điều kiện:
   (a) *Finish*[*i*] = *false*
   (b) $Request_i \leq Work$
   nếu không tồn tại *i*, nhảy sang bước 4.
3. *Work* := *Work* + $Allocation_i$
   Finish[*i*] := *true*
   nhảy sang bước 2.
4. Nếu *Finish*[*i*] = true với ∀i thì hệ thống không có tắc nghẽn
   Nếu *Finish*[*i*] = false, với một số i, $1 \leq i \leq n$, thì P_i bị tắc nghẽn, hệ thống ở trong trạng thái tắc nghẽn.

Độ phức tạp tính toán của giải thuật là $O(m \times n^2)$

Ví dụ giải thuật phát hiện tắc nghẽn

- 5 tiến trình P_0 … P_4;
  3 loại tài nguyên: *A* (7 cá thể), *B* (2 cá thể), và *C* (6 cá thể).
- Giả sử hệ thống tại thời điểm T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Chuỗi <P_0, P_2, P_3, P_1, P_4> sẽ cho kết quả *Finish*[*i*] = true với tất cả các *i* → không có tắc nghẽn.

Ví dụ (tiếp)

- P_2 yêu cầu thêm 1 cá thể loại *C (0 0 1)*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- Trạng thái của hệ thống?
  - Có thể phục hồi các tài nguyên bị giữ bởi tiến trình P_0 khi nó kết thúc, nhưng không đủ tài nguyên để hoàn thành các tiến trình khác.
  - tắc nghẽn xuất hiện, gồm các tiến trình P_1, P_2, P_3, và P_4.

Cách sử dụng giải thuật

- Thời điểm và mức thường xuyên cần đến giải thuật phụ thuộc:
  - tắc nghẽn có khả năng thường xuyên xảy ra như thế nào?
  - Có bao nhiêu tiến trình bị tác động khi tắc nghẽn xuất hiện
- Nếu giải thuật phát hiện tắc nghẽn ít được sử dụng, có thể có nhiều chu trình trong biểu đồ tài nguyên và do đó ta không thể tìm được những tiến trình nào "gây ra" tắc nghẽn.
- Nếu phát hiện được tắc nghẽn, chúng ta cần phục hồi lại bằng một trong hai cách:
  - Dừng các tiến trình
  - Buộc chúng phải giải phóng tài nguyên (ưu tiên trước)

4.3.7. Phục hồi từ tắc nghẽn
4.3.7.1. Dừng tiến trình

- Hủy bỏ tất cả các tiến trình bị tắc nghẽn (có *Finish[i] = false*).
- Hủy bỏ một tiến trình tại một thời điểm đến khi chu trình tắc nghẽn được loại trừ.
- Chúng ta nên chọn hủy bỏ theo trình tự nào?
  - Theo mức ưu tiên của tiến trình.
  - Theo thời gian tiến trình đã thực hiện, và thời gian cần thiết còn lại để hoàn thành.
  - Theo tài nguyên tiến trình đã sử dụng.
  - Theo tài nguyên tiến trình cần để hoàn thành.
  - Theo số tiến trình sẽ cần bị dừng.
  - Tiến trình là tiến trình tương tác hay tiến trình bó?

4.3.7.2. Ưu tiên trước tài nguyên

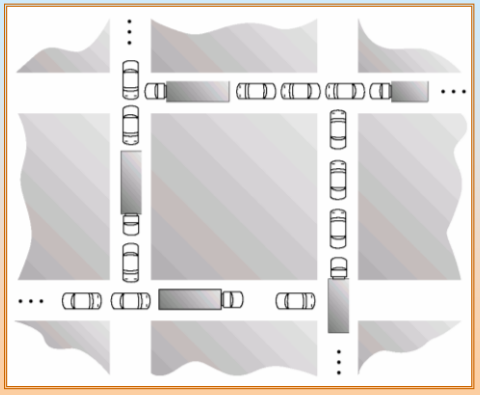
- Chọn một tiến trình nạn nhân dựa vào giá trị nhỏ nhất (mức ưu tiên, số tài nguyên đang dùng…).
- Rollback – quay lại trạng thái an toàn trước, khởi động lại tiến trình ở trạng thái đó.
- Starvation – 1 tiến trình có thể luôn bị chọn làm nạn nhân khiến nó không thể kết thúc. Phải đảm bảo rằng một tiến trình được chọn làm nạn nhân chỉ trong khoảng thời gian ngắn.
  - Giải pháp: Thêm các rollback vào yếu tố chi phí.

4.3.8. Phương pháp kết hợp xử lý tắc nghẽn

- Kết hợp 3 phương pháp cơ bản
  - ngăn ngừa - prevention
  - tránh khỏi - avoidance
  - phát hiện - detection

  tạo thành phương pháp tối ưu đối với mỗi tài nguyên trong hệ thống.
- Phân chia các tài nguyên thành các lớp theo thứ tự phân cấp.
- Sử dụng kỹ thuật thích hợp nhất để xử lý tắc nghẽn trong mỗi lớp.

Traffic Deadlock



BÀI TẬP