Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Bế tắc (Deadlock)

2

Định nghĩa

- Bế tắc là tình huống xuất hiện khi hai hay nhiều "hành động" phải chờ một hoặc nhiều hành động khác để kết thúc, nhưng không bao giờ thực hiện được
- Máy tính: Bế tắc là tình huống xuất hiện khi hai tiến trình phải chờ đợi nhau giải phóng tài nguyên hoặc nhiều tiến trình chờ sử dụng các tài nguyên theo một "vòng tròn" (circular chain)

3

Hai con dê qua cầu: Bế tắc

4

Bế tắc giao thông tại ngã tư

5

Bế tắc trong máy tính

- Tiến trình A:

```
{
   …
   Khóa file F1;
   ...
   Mở file F2;
   …
   Đóng F1 (mở khóa F1);
}
```

- Tiến trình B

```
{
   …
   Khóa file F2;
   ...
   Mở file F1;
   …
   Đóng F1 (mở khóa F1);
}
```

6

Qui trình sử dụng tài nguyên

- Một tiến trình thường sử dụng tài nguyên theo các bước tuần tự sau:
  - Xin phép sử dụng (request)
  - Sử dụng tài nguyên (use)
  - Giải phóng tài nguyên sau khi sử dụng (release)

7

Điều kiện cần để có bế tắc

- Bế tắc xuất hiện nếu 4 điều kiện sau xuất hiện đồng thời (điều kiện cần):
  - C1: Loại trừ lẫn nhau (mutual exclusion)
  - C2: Giữ và chờ (hold and wait)
  - C3: Không có đặc quyền (preemption)
  - C4: Chờ vòng (circular wait)

8

C1: Loại trừ lẫn nhau

- Một tài nguyên bị chiếm bởi một tiến trình, và không tiến trình nào khác có thể sử dụng tài nguyên này

9

C2: Giữ và chờ

- Một tiến trình giữ ít nhất một tài nguyên và chờ một số tài nguyên khác rồi để sử dụng. Các tài nguyên này đang bị một tiến trình khác chiếm giữ

10

C3: Không có đặc quyền

- Tài nguyên bị chiếm giữ chỉ có thể rỗi khi tiến trình "tự nguyện" giải phóng tài nguyên sau khi đã sử dụng xong.

11

C3: Chờ vòng

- Một tập tiến trình $\{P_0, P_1, ..., P_n\}$ có xuất hiện điều kiện "chờ vòng" nếu P_0 chờ một tài nguyên do P_1 chiếm giữ, P_1 chờ một tài nguyên khác do P_2 chiếm giữ, ..., P_{n-1} chờ tài nguyên do P_n chiếm giữ và P_n chờ tài nguyên do P_0 chiếm giữ

12

Đồ thị cấp phát tài nguyên

- Thuật ngữ: Resource allocation graph
- Để mô tả một cách chính xác bế tắc, chúng ta sử dụng đồ thị có hướng gọi là "đồ thị cấp phát tài nguyên" $G=(V, E)$ với V là tập đỉnh, E là tập cung
- V được chia thành hai tập con $P=\{P_0, P_1, ..., P_n\}$ là tập các tiến trình trong hệ thống và $R=\{R_0, R_1, ..., R_m\}$ là tập các loại tài nguyên trong hệ thống thỏa mãn $P \cup R=V$ và $P \cap R= \varnothing$

13

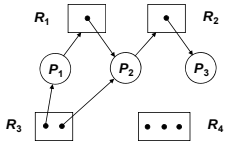
Đồ thị cấp phát tài nguyên

- Cung có hướng từ tiến trình P_i đến tài nguyên R_j, ký hiệu là $P_i \rightarrow R_j$ có ý nghĩa: Tiến trình P_i yêu cầu một *thể hiện* của R_i. Ta gọi $P_i \rightarrow R_j$ là *cung yêu cầu* (*request edge*)
- Cung có hướng từ tài nguyên R_j đến tiến trình P_i ký hiệu là $R_j \rightarrow P_i$ có ý nghĩa: Một thể hiện của tài nguyên Rj đã được cấp phát cho tiến trình Pi. Ta gọi $R_j \rightarrow P_i$ là *cung cấp phát* (*asignment edge*)

14

Đồ thị cấp phát tài nguyên

- Ký hiệu hình vẽ:
  - P_i là hình tròn
  - R_i là các hình chữ nhật với mỗi chấm bên trong là số lượng các thể hiện của tài nguyên
- Minh họa đồ thị cấp phát tài nguyên:



15

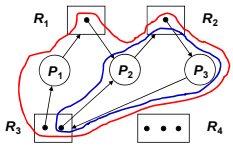
Đồ thị cấp phát tài nguyên

- Nếu không có chu trình trong đồ thị cấp phát tài nguyên: Không có bế tắc. Nếu có chu trình: *Có thể* xảy ra bế tắc.
- Nếu trong một chu trình trong đồ thị cấp phát tài nguyên, mỗi loại tài nguyên chỉ có đúng một thể hiện: Bế tắc đã xảy ra (Điều kiện cần và đủ)
- Nếu trong một chu trình trong đồ thị cấp phát tài nguyên một số tài nguyên có nhiều hơn một thể hiện: Có thể xảy ra bế tắc (Điều kiện cần nhưng không đủ)

16

Ví dụ chu trình dẫn đến bế tắc

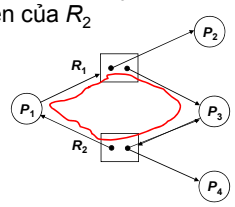
- Giả sử P_3 yêu cầu một thể hiện của R_3
- Khi đó có 2 chu trình xuất hiện:
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_1$, và
  - $P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2$
- Khi đó các tiến trình P_1, P_2, P_3 bị bế tắc



17

Ví dụ chu trình không dẫn đến bế tắc

- Chu trình: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Bế tắc không xảy ra vì P_4 có thể giải phóng một thể hiện tài nguyên R_2 và P_3 sẽ được cấp phát một thể hiện của R_2



18

Các phương pháp xử lý bế tắc

Các phương pháp xử lý bế tắc

- Một cách tổng quát, có 3 phương pháp:
  - Sử dụng một giao thức để hệ thống không bao giờ rơi vào trạng thái bế tắc: *Deadlock prevention* (*ngăn chặn bế tắc*) hoặc *Deadlock avoidance* (*tránh bế tắc*)
  - Có thể cho phép hệ thống bị bế tắc, phát hiện bế tắc và khắc phục nó
  - Bỏ qua bế tắc, xem như bế tắc không bao giờ xuất hiện trong hệ thống (*Giải pháp này dùng trong nhiều hệ thống, ví dụ Unix, Windows*!!)

Ngăn chặn bế tắc (Deadlock prevention)

Giới thiệu

- Ngăn chặn bế tắc (deadlock prevention) là phương pháp xử lý bế tắc, không cho nó xảy ra bằng cách làm cho ít nhất một điều kiện cần của bế tắc là C1, C2, C3 hoặc C4 không được thỏa mãn (không xảy ra)
- Ngăn chặn bế tắc theo phương pháp này có tính chất tĩnh (statically)

Ngăn chặn "loại trừ lẫn nhau"

- C1 (Loại trừ lẫn nhau): là điều kiện bắt buộc cho các tài nguyên không sử dụng chung được → *Khó làm cho C1 không xảy ra* vì các hệ thống luôn có các tài nguyên không thể sử dụng chung được

Ngăn chặn "giữ và chờ"

- C2 (Giữ và chờ): Có thể làm cho C2 không xảy ra bằng cách đảm bảo:
  - Một tiến trình luôn yêu cầu cấp phát tài nguyên chỉ khi nó không chiếm giữ bất kỳ một tài nguyên nào, hoặc
  - Một tiến trình chỉ thực hiện khi nó được cấp phát toàn bộ các tài nguyên cần thiết

Ngăn chặn "không có đặc quyền"

- Để ngăn chặn không cho điều kiện này xảy ra, có thể sử dụng giao thức sau:
  - Nếu tiến trình P (đang chiếm tài nguyên R_1, ..., R_{n-1}) yêu cầu cấp phát tài nguyên R_n nhưng không được cấp phát ngay (có nghĩa là P phải chờ) thì tất cả các tài nguyên R_1, ..., R_{n-1} phải được "thu hồi"
  - Nói cách khác, R_1, ..., R_{n-1} phải được "giải phóng" một cách áp đặt, tức là các tài nguyên này phải được đưa vào danh sách các tài nguyên mà P đang chờ cấp phát.

25

Ngăn chặn "không có đặc quyền": mã lệnh

Tiến trình P yêu cầu cấp phát tài nguyên R_1, ..., R_{n-1}

if (R_1, ..., R_{n-1} rỗi)

then cấp phát tài nguyên cho P

else if ($\{R_i...R_j\}$ được cấp phát cho Q và Q đang trong trạng thái chờ một số tài nguyên S khác)

then thu hồi $\{R_i...R_j\}$ và cấp phát cho P

else đưa P vào trạng thái chờ tài nguyên R_1, ..., R_{n-1}

26

Ngăn chặn "chờ vòng"

- Một giải pháp ngăn chặn chờ vòng là đánh số thứ tự các tài nguyên và bắt buộc các tiến trình yêu cầu cấp phát tài nguyên theo số thứ tự tăng dần
- Giả sử có các tài nguyên $\{R_1, ..., R_n\}$. Ta gán cho mỗi tài nguyên một số nguyên dương duy nhất qua một ánh xạ 1-1

  $f : R \rightarrow N$, với N là tập các số tự nhiên

  Ví dụ: f(ổ cứng) = 1, f(băng từ) = 5, f(máy in) = 11

27

Ngăn chặn "chờ vòng"

- Giao thức ngăn chặn chờ vòng:
  - Khi tiến trình P không chiếm giữ tài nguyên nào, nó có thể yêu cầu cấp phát nhiều *thể hiện* của *một* tài nguyên R_i bất kỳ
  - Sau đó P chỉ có thể yêu cầu các thể hiện của tài nguyên R_j nếu và chỉ nếu $f(R_j) > f(R_i)$. Một cách khác, nếu P muốn yêu cầu cấp phát tài nguyên R_j, nó đã giải phóng tất cả các tài nguyên R_i thỏa mãn $f(R_i) \geq f(R_j)$
  - Nếu P cần được cấp phát nhiều loại tài nguyên, P phải *lần lượt* yêu cầu các thể hiện của *từng* tài nguyên đó

28

Chứng minh giải pháp ngăn chặn chờ vòng

- Sử dụng chứng minh phản chứng
- Giả sử giải pháp ngăn chặn gây ra chờ vòng $\{P_0, P_1, ..., P_n\}$ trong đó P_i chờ tài nguyên R_i bị chiếm giữ bởi $P_{(i+1) \bmod n}$
- Vì P_{i+1} đang chiếm giữ R_i và yêu cầu R_{i+1}, do đó $f(R_i) < f(R_{(i+1) \bmod n})$ $\forall i$, có nghĩa là ta có:
  - $f(R_0) < f(R_1) < f(R_2) < .. < f(R_n) < f(R_0)$
  - Mâu thuẫn! $\rightarrow$ Giải pháp được chứng minh

29

Ưu nhược điểm của ngăn chặn giải pháp bế tắc

- Ưu điểm: ngăn chặn bế tắc (deadlock prevention) là phương pháp tránh được bế tắc bằng cách làm cho điều kiện cần không được thỏa mãn
- Nhược điểm:
  - Giảm khả năng tận dụng tài nguyên và giảm thông lượng của hệ thống
  - Không mềm dẻo

30

Tránh bế tắc (Deadlock avoidance)

Giới thiệu

- Tránh bế tắc là phương pháp sử dụng thêm các thông tin về phương thức yêu cầu cấp phát tài nguyên để ra quyết định cấp phát tài nguyên sao cho bế tắc không xảy ra.
- Có nhiều thuật toán theo hướng này
- Thuật toán đơn giản nhất và hiệu quả nhất là: Mỗi tiến trình *P* đăng ký số thể hiện của mỗi loại tài nguyên mà *P* sẽ sử dụng. Khi đó hệ thống sẽ có đủ thông tin để xây dựng thuật toán cấp phát không gây ra bế tắc

Giới thiệu

- Các thuật toán như vậy kiểm tra *trạng thái cấp phát tài nguyên* một cách "động" để đảm bảo điều kiện chờ vòng không xảy ra
- Trạng thái cấp phát tài nguyên được xác định bởi số lượng tài nguyên rỗi, số lượng tài nguyên đã cấp phát và số lượng lớn nhất các yêu cầu cấp phát tài nguyên của các tiến trình
- Hai thuật toán sẽ nghiên cứu: *Thuật toán đồ thị cấp phát tài nguyên* và thuật toán *banker*

Trạng thái an toàn (safe-state)

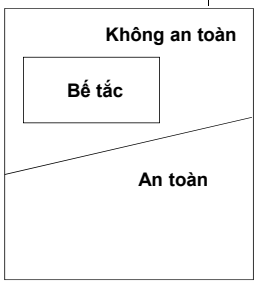
- Một trạng thái (cấp phát tài nguyên) được gọi là an toàn nếu hệ thống có thể cấp phát tài nguyên cho các tiến trình theo một thứ tự nào đó mà vẫn tránh được bế tắc, hay
- Hệ thống ở trong trạng thái an toàn nếu và chỉ nếu tồn tại một *thứ tự an toàn* (safe-sequence)

Thứ tự an toàn

- Thứ tự các tiến trình $<P_1,...,P_n>$ gọi là một thứ tự an toàn (safe-sequence) cho trạng thái cấp-phát hiện-tại nếu với mỗi P_i, yêu cầu cấp phát tài nguyên của P_i vẫn có thể được thỏa mãn căn cứ vào trạng thái của:
  - Tất cả các tài nguyên rỗi hiện có, và
  - Tất cả các tài nguyên đang bị chiếm giữ bởi tất cả các P_j $\forall j<i$.

Các trạng thái an toàn, không an toàn và bế tắc

- Trạng thái an toàn không là trạng thái bế tắc
- Trạng thái bế tắc là trạng thái không an toàn
- Trạng thái không an toàn có thể là trạng thái bế tắc hoặc không



Ví dụ trạng thái an toàn, bế tắc

- Xét một hệ thống có 12 tài nguyên là 12 băng từ và 3 tiến trình P_0, P_1, P_2 với các yêu cầu cấp phát:
  - P_0 yêu cầu nhiều nhất 10 băng từ
  - P_1 yêu cầu nhiều nhất 4 băng từ
  - P_2 yêu cầu nhiều nhất 9 băng từ
- Giả sử tại một thời điểm t_0, P_0 đang chiếm 5 băng từ, P_1 và P_2 mỗi tiến trình chiếm 2 băng từ. Như vậy có 3 băng từ rỗi

37

Ví dụ trạng thái an toàn, bế tắc

	Yêu cầu nhiều nhất	Yêu cầu hiện tại
P_0	10	5
P_1	4	2
P_2	9	2

- Tại thời điểm t_0, hệ thống ở trạng thái an toàn
- Thứ tự $<P_1, P_0, P_2>$ thỏa mãn điều kiện an toàn
- Giả sử ở thời điểm t_1, P_2 có yêu cầu và được cấp phát 1 băng từ: Hệ thống không ở trạng thái an toàn nữa... -> quyết đinh cấp tài nguyên cho P_2 là sai.

38

Thuật toán đồ thị cấp phát tài nguyên

- Giả sử các tài nguyên chỉ có 1 thể hiện
- Sử dụng đồ thị cấp phát tài nguyên như ở slide 16 và thêm một loại cung nữa là *cung báo trước* (claim)
- Cung báo trước $P_i \rightarrow R_j$ chỉ ra rằng P_i có thể yêu cầu cấp phát tài nguyên R_j, được biểu diễn trên đồ thị bằng các đường nét đứt
- Khi tiến trình P_i yêu cầu cấp phát tài nguyên R_j, đường nét đứt trở thành đường nét liền

39

Thuật toán đồ thị cấp phát tài nguyên

- Chú ý rằng các tài nguyên phải được thông báo trước khi tiến trình thực hiện
- Các cung báo trước sẽ phải có trên đồ thị cấp phát tài nguyên
- Tuy nhiên có thể giảm nhẹ điều kiện: cung thông báo $P_i \rightarrow R_j$ được thêm vào đồ thị nếu tất cả các cung gắn với P_i đều là cung thông báo

40

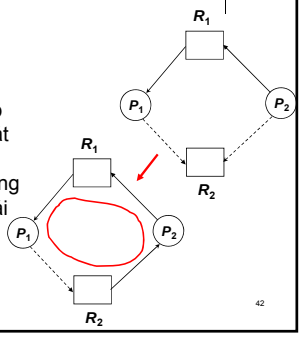
Thuật toán đồ thị cấp phát tài nguyên

- Giả sử P_i yêu cầu cấp phát R_j. Yêu cầu này chỉ có thể được chấp nhận nếu ta chuyển cung báo trước $P_i \rightarrow R_j$ thành cung cấp phát $R_j \rightarrow P_i$ và không tạo ra một chu trình
- Chúng ta kiểm tra bằng cách sử dụng thuật toán phát hiện chu trình trong đồ thị: Nếu có n tiến trình trong hệ thống, thuật toán phát hiện chu trình có độ phức tạp tính toán $O(n^2)$
- Nếu không có chu trình: Cấp phát->trạng thái an toàn, ngược lại: Trạng thái không an toàn

41

Ví dụ

- Giả sử P_1 yêu cầu cấp phát R_2
- Mặc dù R_2 rỗi nhưng chúng ta không thể cấp phát R_2, vì nếu cấp phát ta sẽ có chu trình trong đồ thị và gây ra chờ vòng → Hệ thống ở trạng thái không an toàn



42

Thuật toán banker

- Thuật toán đồ thị phân phối tài nguyên không áp dụng được cho các hệ thống có những tài nguyên có nhiều thể hiện
- Thuật toán *banker* được dùng cho các hệ có tài nguyên nhiều thể hiện, nó kém hiệu quả hơn thuật toán đồ thị phân phối tài nguyên
- Thuật toán banker có thể dùng trong ngân hàng: Không bao giờ cấp phát tài nguyên (tiền) gây nên tình huống sau này không đáp ứng được nhu cầu của tất cả các khách hàng

43

Banker cần có dữ liệu vào:

- Mỗi tiến trình có thể xin cấp phát bao nhiêu thể hiện của mỗi loại tài nguyên?
- Mỗi tiến trình đang nắm giữ bao nhiêu thể hiện của mỗi loại tài nguyên?
- Bao nhiêu thể hiện của mỗi loại tài nguyên của hệ thống hiện đang rỗi (có thể sử dụng được)?

44

Banker cấp phát tài nguyên khi

- Request≤ Max
- Request≤ Available

45

Ký hiệu dùng trong banker

- Tài nguyên rỗi: Vector *m* thành phần *Available*, *Available*[*j*]=*k* nghĩa là có *k* thể hiện của R_j rỗi
- Max: Ma trận *nxm* xác định yêu cầu tài nguyên max của mỗi tiến trình. *Max*[*i*][*j*]=*k* có nghĩa là tiến trình P_i yêu cầu nhiều nhất *k* thể hiện của tài nguyên R_j.

46

Ký hiệu dùng trong banker

- Cấp phát: Ma trận *nxm* xác định số thể hiện của các loại tài nguyên đã cấp phát cho mỗi tiến trình. *Allocation*[*i*][*j*]=*k* có nghĩa là tiến trình P_i được cấp phát *k* thể hiện của R_j.
- Cần thiết: Ma trận *nxm* chỉ ra số lượng thể hiện của các tài nguyên mỗi tiến trình cần cấp phát tiếp. *Need*[*i*][*j*]=*k* có nghĩa là tiến trình P_i còn có thể cần thêm *k* thể hiện nữa của tài nguyên R_j.

47

Ký hiệu dùng trong banker

- Số lượng và giá trị các biến trên biến đổi theo trạng thái của hệ thống
- Qui ước: Nếu hai vector *X*, *Y* thỏa mãn *X*[*i*]≤*Y*[*i*] ∀*i* thì ta ký hiệu *X*≤*Y*.
- Giả sử *Work* và *Finish* là các vector *m* và *n* thành phần.
- *Request*[*i*] là vector yêu cầu tài nguyên của tiến trình P_i. *Request*[*i*][*j*]=*k* có nghĩa là tiến trình P_i yêu cầu *k* thể hiện của tài nguyên R_j

48

Thuật toán trạng thái an toàn

1. Khởi tạo *Work*=*Available* và *Finish*[*i*]=false $\forall i$=1..*n*
2. Tìm *i* sao cho *Finish*[*i*]==false và *Need*[*i*]≤*Work*
   Nếu không tìm được *i*, chuyển đến bước 4
3. *Work*=*Work*+*Allocation*[i], *Finish*[*i*]=true
   Chuyển đến bước 2
4. Nếu *Finish*[*i*]==true $\forall i$ thì hệ thống ở trạng thái an toàn

- Độ phức tạp tính toán của thuật toán trạng thái an toàn: $O(m.n^2)$

49

Thuật toán yêu cầu tài nguyên

1. Nếu *Request*[*i*]≤*Need*[*i*], chuyển đến bước 2
   Ngược lại thông báo lỗi (không có tài nguyên rỗi)
2. Nếu *Request*[*i*]≤*Available*, chuyển đến bước 3.
   Ngược lại P_i phải chờ vì không có tài nguyên
3. Nếu việc thay đổi trạng thái giả định sau đây:
   Available=*Availalble*-*Request*[i]
   Allocation=*Allocation*+*Request*[i]
   Need[i]=*Need*[i]-*Request*[i]
   đưa hệ thống vào trạng thái an toàn thì cấp phát tài nguyên cho P_i, ngược lại P_i phải chờ *Request*[*i*] và trạng thái của hệ thống được khôi phục như cũ

50

Ví dụ banker

- Xét một hệ thống các tiến trình và tài nguyên như sau:

	Allocation			*Max*			*Available*			*Need*		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

51

Ví dụ banker

- Hệ thống hiện đang ở trạng thái an toàn
- Thứ tự <P_1,P_3,P_4,P_2,P_0> thỏa mãn tiêu chuẩn an toàn
- Giả sử P_1 có yêu cầu: *Request*[1]=(1,0,2)
- Để quyết định xem có cấp phát tài nguyên theo yêu cầu này không, trước hết ta kiểm tra *Request*[1]≤*Available*: (1,0,2)<(3,3,2): Đúng
- Giả sử yêu cầu này được cấp phát, khi đó trạng thái giả định của hệ thống là:

52

Ví dụ banker

	Allocation			*Need*			*Available*		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

- Thực hiện thuật toán trạng thái an toàn và thấy rằng thứ tự <P_1,P_3,P_4,P_0,P_2>. Do đó có thể cấp phát tài nguyên cho P_1 ngay.

53

Ví dụ banker

- Tuy nhiên, nếu hệ thống ở trạng thái sau thì
  - Yêu cầu (3,3,0) của P4 không thể cấp phát ngay vì các tài nguyên không rỗi
  - Yêu cầu (0,2,0) của P0 cũng không thể cấp phát ngay vì mặc dù các tài nguyên rỗi nhưng việc cấp phát sẽ làm cho hệ thống rơi vào trạng thái không an toàn
- **Bài tập**: Thực hiện kiểm tra và quyết định cấp phát hai yêu cầu trên

54

Phát hiện bế tắc

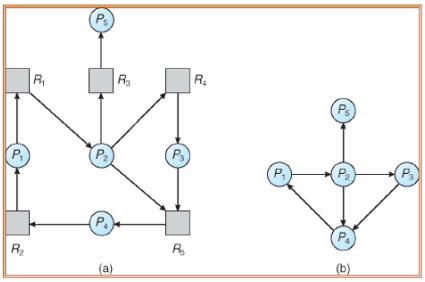
- Nếu không áp dụng phòng tránh hoặc ngăn chặn bế tắc thì hệ thống có thể bị bế tắc
- Khi đó:
  - Cần có thuật toán kiểm tra trạng thái để xem có bế tắc xuất hiện hay không
  - Thuật toán khôi phục nếu bế tắc xảy ra

55

Tài nguyên chỉ có một thể hiện

- Sử dụng thuật toán đồ thị chờ: Đồ thị chờ có được từ đồ thị cấp phát tài nguyên bằng cách xóa các đỉnh tài nguyên và nối các cung liên quan
  - Cung $P_i \rightarrow P_j$ có nghĩa là P_i đang chờ P_j giải phóng tài nguyên mà P_i cần
  - Cung $P_i \rightarrow P_j$ tồn tại trong đồ thị chờ nếu và chỉ nếu đồ thị cấp phát tài nguyên tương ứng có hai cung $P_i \rightarrow R_q$ và $R_q \rightarrow P_j$ với R_q là tài nguyên
  - Hệ thống có bế tắc nếu đồ thị chờ có chu trình
  - Để phát hiện bế tắc: Cần cập nhật đồ thị chờ và thực hiện định kỳ thuật toán phát hiện chu trình

56



57

Tài nguyên có nhiều thể hiện

- *Available*: Vector *m* thành phần chỉ ra số lượng thể hiện của mỗi loại tài nguyên
- *Allocation*: Ma trận *nxm* xác định số thể hiện của mỗi loại tài nguyên đang được cấp phát cho các tiến trình
- *Request*: Ma trận *nxm* xác định yêu cầu hiện tại của mỗi tiến trình. Nếu *Request*[*i*][*j*]=*k* thì tiến trình P_i yêu cầu cấp phát *k* thể hiện của tài nguyên R_j.

58

Tài nguyên có nhiều thể hiện

1. Giả sử *Work* và *Finish* là các vector *m* và *n* thành phần. Khởi tạo *Work*=*Available*. Với mỗi *i*=0..*n*-1 gán *Finish*[*i*]=false nếu *Allocation*[*i*]≠0, ngược lại gán *Finish*[*i*]=true
2. Tìm *i* sao cho *Finish*[*i*]==false và *Request*[*i*]≤*Work*. Nếu không tìm thấy *i*, chuyển đến bước 4
3. *Work*=*Work*+*Allocation*, *Finish*[*i*]=true; chuyển đến bước 2
4. Nếu *Finish*[*i*]==false với 0≤*i*≤*n*-1 thì hệ thống đang bị bế tắc (và tiến trình P_i đang bế tắc).

- Độ phức tạp tính toán của thuật toán: $O(m.n^2)$

59

Sử dụng thuật toán phát hiện

- Tần suất sử dụng phụ thuộc:
  - Tần suất xảy ra bế tắc
  - Bao nhiêu tiến trình bị ảnh hưởng bởi bế tắc?
- Sử dụng thuật toán phát hiện:
  - Định kỳ: Có thể có nhiều chu trình trong đồ thị, không biết được tiến trình/request nào gây ra bế tắc
  - Khi có yêu cầu cấp phát tài nguyên: Tốn tài nguyên CPU

60

Khôi phục khi có bế tắc

- Kết thúc tiến trình:
  - Kết thúc toàn bộ các tiến trình bị bế tắc (1)
  - Kết thúc từng tiến trình và dừng quá trình này khi bế tắc chấm dứt (2)
- Tiến trình bị kết thúc ở (2) căn cứ vào:
  - Độ ưu tiên
  - Thời gian đã thực hiện và thời gian còn lại
  - Số lượng và các loại tài nguyên đã sử dụng
  - Các tài nguyên cần cấp phát thêm
  - Số lượng các tiến trình phải kết thúc
  - Tiến trình là tương tác hay xử lý theo lô (batch)

61

Khôi phục khi có bế tắc

- Giải phóng tài nguyên một cách bắt buộc (preemption):
  - Chọn tài nguyên nào và tiến trình nào để thực hiện?
  - Khôi phục trạng thái của tiến trình đã chọn ở (1) như thế nào?
  - Làm thế nào để tránh tình trạng một tiến trình luôn bị bắt buộc giải phóng tài nguyên?

62

Tóm tắt

- Khái niệm bế tắc
- Các điều kiện cần để có bế tắc
- Đồ thị phân phối tài nguyên
- Các phương pháp xử lý bế tắc: Ngăn chặn và tránh bế tắc (thuật toán đồ thị cấp phát tài nguyên và thuật toán banker)
- Khôi phục khi bế tắc đã xảy ra: Kết thúc tiến trình và preemption

63

Bài tập

- Thực hiện lại ví dụ phát hiện bế tắc ở trang 264 trong giáo trình
- Làm bài tập số 7.1 trong giáo trình (trang 268)
- Làm bài tập số 7.11 trong giáo trình (trang 270)

64