Chương 5: Khoá chết - Deadock

Vấn đề khoá chết & Cách giải quyết khoá chết

Nội dung

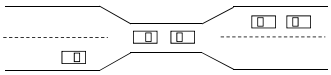
- Vấn đề DeadLock
- Mô hình hệ thống
- Mô tả deadlock
- Các phương pháp xử lý deadlock
- Ngăn ngừa deadlock
- Tránh khỏi deadlock
- Phát hiện deadlock
- Phục hồi từ deadlock
- Phương pháp kết hợp xử lý deadlock

1. Vấn đề DeadLock(1)

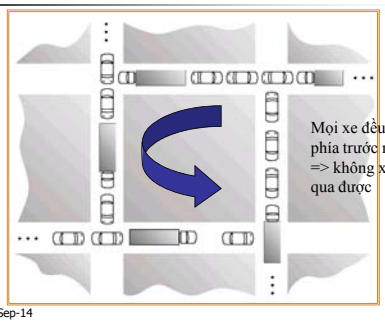
- Trong chế độ multiprogramming, một số tiến trình có thể tranh chấp một số tài nguyên hạn chế.
- Một tiến trình yêu cầu các tài nguyên, nếu tài nguyên không thể đáp ứng tại thời điểm đó thì tiến trình sẽ chuyển sang trạng thái **waiting**.
- Các tiến trình **waiting** có thể sẽ không bao giờ thay đổi lại trạng thái được vì các tài nguyên mà nó yêu cầu lại bị giữ bởi các tiến trình **waiting** khác.
- ***Deadlock***(khoá chết - bế tắc):
  - Là hiện tượng một tiến trình chiếm hữu tài nguyên lâu dài làm cho các tiến trình có nhu cầu sử dụng tài nguyên này luôn ở trạng thái **waiting**
  - Một **tiến trình deadlock** là tiến trình đợi một sự kiện không bao giờ xảy ra

1. Vấn đề DeadLock(2)

- Ví dụ: tắc nghẽn trên cầu
  - Hai (hay nhiều hơn) ô tô đi ngược chiều 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 deadlock xuất hiện, nó có thể được giải quyết nếu một hay một số ô tô lùi lại nhường đường rồi tiến sau.



1. Vấn đề DeadLock(3)

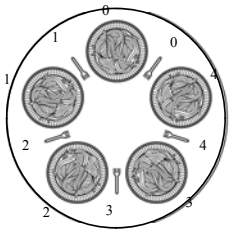


Mọi xe đều đợi xe phía trước mình đi => không xe nào qua được

1. Vấn đề DeadLock(4)

- Dining Philosophers:
  - N nhà triết học và N cái đĩa
  - Các nhà triết học ăn/suy nghĩ
  - Ăn cần 2 cái đĩa
  - Mỗi lần cầm lên 1 cái đĩa

⇒ **Deadlock**: mọi người cùng cầm đĩa bên tay trái trước và ngồi đợi… => chết đói!



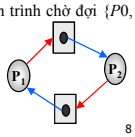
N=5

2. Mô tả hệ thống

- Các loại tài nguyên ký hiệu $R1, R2, \ldots, Rm$
  - R có thể là: các chu kỳ CPU, không gian bộ nhớ, file, I/O device
- Mỗi loại tài nguyên Ri có Wi cá thể (instance).
  - Ex: hệ thống có 2 CPU, có 5 máy in 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** tài nguyên (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** tài nguyên (use)
  3. **Giải phóng** tài nguyên (release)

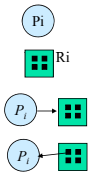
3. Mô tả DeadLock

- Deadlock có thể xảy ra nếu 4 điều kiện sau **đồng thời** tồn tại:
  1. **Ngăn chặn(loại trừ) lẫn nhau:** tại một thời điểm, chỉ một tiến trình có thể sử dụng một tài nguyên*(chỉ một tiến trình trong đoạn găng)*.
  2. **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.
  3. **Không có ưu tiên(độc quyền):** một tài nguyên chỉ có thể được tiến trình (tự nguyện!) giải phóng khi nó đã hoàn thành công việc.
  4. **Chờ đợi vòng tròn:** tồn tại một tập các tiến trình chờ đợi {*P*0, *P*1, …, *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, …
     - *Pn*–1 đang đợi tài nguyên bị giữ bởi *P*n,
     - và *P*n đang đợi tài nguyên bị giữ bởi *P*0.



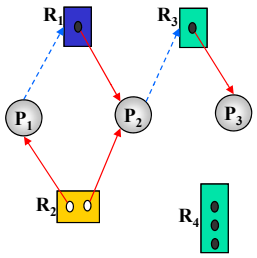
3.1. Biểu đồ phân phối tài nguyên – **RAG**(1)

- Một tập các đỉnh V và một tập các cạnh E.
  - V được chia thành 2 loại:
    - $P = \{P1, P2, \ldots, Pn\}$: tập tất cả các tiến trình.
    - $R = \{R1, R2, \ldots, Rm\}$: tập các loại tài nguyên.
  - Tập các cạnh E:
    - Mỗi cá thể là một hình vuông bên trong cạnh yêu cầu – cạnh có hướng $Pi \rightarrow Rj$. *(tiến trình Pi đang đợi nhận một hay nhiều cá thể của tài nguyên Rj).*
    - Cạnh chỉ định – cạnh có hướng $Ri \rightarrow Pi$. *(tiến trình Pi đang giữ một hay nhiều cá thể của tài nguyên Ri).*



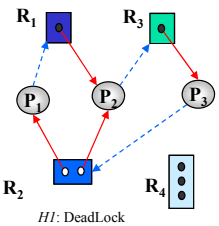
3.1. Biểu đồ phân phối tài nguyên – RAG(2): Minh hoạ

- Đồ thị phân phối tài nguyên không chu trình
  - Nếu đồ thị không chu trình thì sẽ không có tiến trình nào bị deadlock
  - Nếu đồ thị có chu trình thì *có thể* tồn tại deadlock

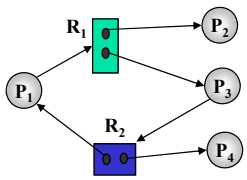


3.1. Biểu đồ phân phối tài nguyên – RAG(3): Minh hoạ

- Đồ thị phân phối tài nguyên có chu trình



H1: DeadLock



H2: Không DeadLock
*P*4 hoặc *P*2 có thể kết thúc
-> *P*1 và *P*3 kết thúc được.

3.1. Biểu đồ phân phối tài nguyên(4): Kết luận về đồ thị

- Nếu đồ thị không chu trình
  - Không xảy ra deadlock.
- Nếu đồ thị có chu trình
  - Nếu mỗi loại tài nguyên chỉ một cá thể thì ***chắc chắn*** xảy ra deadlock.
  - Nếu mỗi loại tài nguyên có một vài cá thể thì deadlock ***có thể*** xảy ra hoặc không.

4. Các phương pháp xử lý deadlock

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

5. Ngăn ngừa deadlock(1)

- 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
    - Ví dụ: read-only files(vì file read only thì nhiều tiến trình truy xuất nó cũng không ảnh hưởng gì-nội dung file không thay đổi)
  - Một số tài nguyên là không thể chia sẻ
    - Ví dụ: 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 không có tài nguyên nào cả.

5. Ngăn ngừa deadlock(2)

- **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**
  - Muốn có tài nguyên j, tiến trình phải giải phóng tất cả các tài nguyên có trọng số $i > j$ (nếu có)

6. Thoát khỏi deadlock

- 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 deadlock **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 **số tài nguyên tối đa** tiến trình yêu cầu.

6.1. Safe State(1)

- Một **trạng thái an toàn** nếu *hệ thống **có thể** phân phối các tài nguyên cho mỗi tiến trình theo một vài thứ tự nào đó mà vẫn tránh được deadlock.*
- 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**.
  - Chuỗi tiến trình là một dãy các tiến trình thực hiện theo thứ tự từ đầu đến cuối.

6.1. Safe State(2)

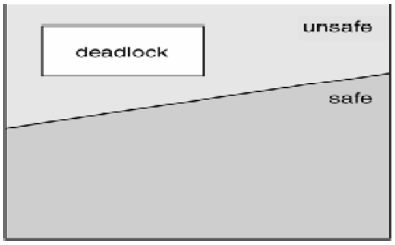
- Chuỗi tiến trình <P1, P2, …, Pn> là an toàn nếu với mỗi Pi, tài nguyên yêu cầu có thể được cung cấp bởi **tài nguyên khả dụng***(chưa phân phối cho tiến trình nào)* hiện tại và các tài nguyên đang được giữ bởi Pj, với j<i.
  - Nếu tài nguyên Pi cần đang bị Pj giữ thì nó có thể đợi cho đến khi tất cả 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, ...

6.1. Safe State(3)

- Nếu hệ thống ở trạng thái an toàn không có deadlock.
- Nếu hệ thống ở trạng thái không an toàn có thể có deadlock.
- Sự tránh khỏi deadlock 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**.

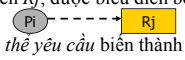
6.1. Safe State(4)

- So sánh: Safe, Unsafe, Deadlock State



6.2. Giải thuật đồ thị phân phối tài nguyên(1)

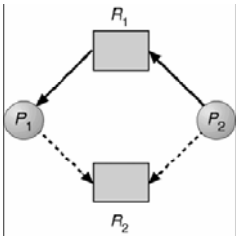
- **Claim edge** - *Cạnh có thể yêu cầu* (may request resource) biểu diễn bằng nét liền, *Pi → Rj* : tiến trình *Pi* đã yêu cầu tài nguyên *Rj*; được biểu diễn bởi một đường đứt nét.



- Cạnh *có thể 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 muốn yêu cầu trở lại thành cạnh co the yêu cầu.
- Hệ thống ở trong trạng thái an toàn khi đồ thị không chứa chu trình nào.
  - Khi xét **safe state,** coi các cạnh *có thể yêu cầu* như là các *cạnh yêu cầu*(vì giống nhau về phương, hướng)

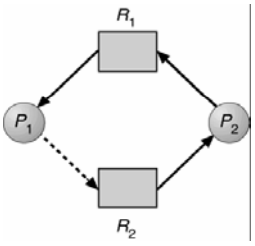
6.2. Giải thuật đồ thị phân phối tài nguyên(2): tránh DeadLock

- Giả sử P2 yêu cầu R2.
- Dù R2 vẫn đang tự do, vẫn không thể phân phối nó cho P2 vì sẽ tạo ra một chu trình → hệ thống trong trạng thái không an toàn.
- Nếu P1 yêu cầu R2 và P2 yêu cầu R1 thì deadlock sẽ xuất hiện.



6.2. Giải thuật đồ thị phân phối tài nguyên(3)

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



6.3. Giải thuật chủ nhà băng(1)

- 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 để luôn thỏa mãn mọi yêu cầu từ các khách hàng*.
- Khi một tiến trình mới đi vào hệ thống:
  - 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 tiến trình 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.

6.3. Giải thuật chủ nhà băng(2): Cấu trúc dữ liệu

- *Giả thiết:n* = số tiến trình, *m* = số loại tài nguyên.
- ***Available***: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available.
- ***Max***: *n* x *m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.
- ***Allocation***: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j.
- ***Need***: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.

⇒ Need[i, j] = Max[i, j] – Allocation[i, j]

6.3. Giải thuật chủ nhà băng(3): Kiểm tra an toàn

- *Tư tưởng*: chúng ta tìm một chuỗi an toàn. Nếu tìm được, trạng thái hệ thống 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*] := false 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.
3. $Work := Work + Allocation_i$
   - *Finish*[*i*] := *true*
   - nhảy đến bước 2.
4. Nếu *Finish*[*i*] = true với tất cả *i* thì hệ thống ở trạng thái an toàn.

6.3. Giải thuật chủ nhà băng(3): Resource-Request Algorithm P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants *k* instances of resource type R_j.

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;

   - *If safe* ⇒ *the resources are allocated to* P_i.
   - *If unsafe* ⇒ P_i *must wait, and the old resource-allocation state is restored*

6.3. Giải thuật chủ nhà băng(4): Phân tích ví dụ

- 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:

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

6.3. Giải thuật chủ nhà băng(5): Phân tích ví dụ

- Need = Max – Allocation.

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

- 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.

6.3. Giải thuật chủ nhà băng(6): Phân tích ví dụ: run test

- Khởi tạo: work[3]=(3,3,2); finish[5]=(F,F,F,F,F)

$Need_{P1}$[i]<=work[i] với i=1->3

⇒Gán: $finish_{P1}$=T,

Work=(3,3,2)+(2,0,0)

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

- Kết quả vòng 1: $finish_{P1}$=T, work=(5,3,2)

6.3. Giải thuật chủ nhà băng(7): Phân tích ví dụ: run test

- Vòng 2: work[3]=(5,3,2); finish[5]=(T,F,F,F,F)

Need$_{P3}$[i]<=work[i] với i=1->3
⇒Gán: finish$_{P3}$=T,
Work=(5,3,2)+(2,1,1)

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

- Kết quả vòng 2: finish$_{P3}$=T, work=(7,4,3)

6.3. Giải thuật chủ nhà băng(8): Phân tích ví dụ: run test

- Vòng 3: work[3]=(7,4,3); finish[5]=(T,T,F,F,F)

Need$_{P4}$[i]<=work[i] với i=1->3
⇒Gán: finish$_{P4}$=T,
Work=(7,4,3)+(0,0,2)

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

- Kết quả vòng 3: finish$_{P4}$=T, work=(7,4,5)

6.3. Giải thuật chủ nhà băng(9): Phân tích ví dụ: run test

- Vòng 4: work[3]=(7,4,5); finish[5]=(T,T,T,F,F)

Need$_{P2}$[i]<=work[i] với i=1->3
⇒Gán: finish$_{P2}$=T,
Work=(7,4,5)+(3,0,0)

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

- Kết quả vòng 4: finish$_{P2}$=T, work=(10,4,5)

6.3. Giải thuật chủ nhà băng(10): Phân tích ví dụ: run test

- Vòng 5: work[3]=(10,5,5); finish[5]=(T,T,T,T,F)

Need$_{P0}$[i]<=work[i] với i=1->3
⇒Gán: finish$_{P0}$=T,
Work=(10,4,5)+(0,1,0)

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

- Kết quả vòng 5: finish[5]=(T,T,T,T,T) => **safe state**

6.4. Giải thuật yêu cầu tài nguyên cho tiến trình *Pi* (1)

- *Request* = vectơ yêu cầu cho tiến trình *Pi*. Nếu *Requesti* [*j*] = *k* thì tiến trình *Pi* muốn *k* cá thể của loại tài nguyên *Rj*.
- 1. Nếu *Requesti* ≤ *Needi* , 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 *Requesti* ≤ *Available*, chuyển sang bước 3. Trái lại *Pi* phải đợi vì tài nguyên chưa sẵn sàng.
- 3. Giả định phân phối các tài nguyên cho *Pi* bằng cách sửa trạng thái như sau:
  - *Available = Available – Requesti ;*
  - *Allocationi= Allocationi + Requesti ;*
  - *Needi = Needi – Requesti ;*
    - *Nếu an toàn -> phân phối tài nguyên cho Pi.*
    - *Nếu không an toàn -> Pi phải đợi, và trạng thái phân phối tài nguyên cũ được phục hồi.*

6.4. Giải thuật yêu cầu tài nguyên cho tiến trình *Pi* (

Need(ban dau)
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

- Ví dụ *P*1 yêu cầu (1,0,2)
  - Kiểm tra Request ≤ Need ↔ (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	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Tình huống hiện tại

	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 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

6.4. Phân tích ví dụ: Is Allocation (1 0 2) to P1 Safe?

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

Tài Nguyên Chưa phân phối

If P1 requests max resources, can complete

6.4. Phân tích ví dụ: Run Safety Test

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

If P1 requests max resources, can complete

6.4. Phân tích ví dụ: Allocate to P1, Then

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

6.4. Phân tích ví dụ: Release - P1 Finishes

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

Now P3 can acquire max resources and release

6.4. Phân tích ví dụ: Release - P3 Finishes

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

Now P4 can acquire max resources and release

6.4. Phân tích ví dụ: Release - P4 Finishes

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

Now P2 can acquire max resources and release

6.4. Phân tích ví dụ: Release - P2 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3			
P1	0	0	0	3	2	2	3	2	2	10	4	5
P2	0	0	0	9	0	2	9	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	0	4	3	3	4	3	3			

Now P0 can acquire max resources and release

6.4. Phân tích ví dụ: So P1 Allocation (1 0 2) Is Safe

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

6.4. Phân tích ví dụ: Is allocation (0 2 0) to P0 Safe?

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

Try to Allocate 2 B to P0

6.4. Phân tích ví dụ: Run Safety Test

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

No Processes may get max resources and release

6.4. Phân tích ví dụ: Avoid Unsafe State- Do Not give (0,2,0) to P0

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

Return to Safe State and do not allocate resource

6.4. Phân tích ví dụ: Suspend P0, Pending Request

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

When enough resources become available, P0 can be woken

7. Phát hiện deadlock

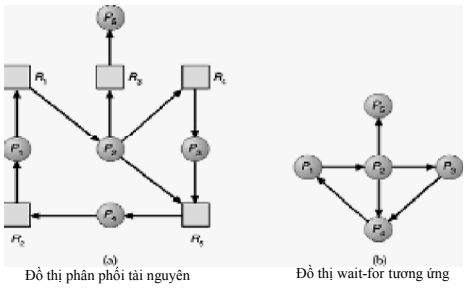
- Nếu một hệ thống không thể thực hiện được việc ngăn ngừa hay tránh xa deadlock thì deadlock có thể xuất hiện. Trong môi trường này, hệ thống phải cung cấp:
  - Giải thuật phát hiện deadlock
  - Giải thuật phục hồi từ deadlock

7.1. Mỗi loại tài nguyên có 1 cá thể

- Khi tất cả các loại tài nguyên chỉ có một cá thể, giải thuật xác định deadlock 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.
  - *Pi* → *Pj* nếu *Pi* đang đợi *Pj*.
- Định kỳ sử dụng giải thuật tìm kiếm chu trình trong đồ thị.
- Giải thuật đó đòi hỏi n^2 phép toán(sử dụng thuật toán sắp xếp Topo để kiểm tra xem đồ thị có chu trình?), với n là số đỉnh trong đồ thị: có chu trình → có thể có deadlock.

Có thể chọn làm tiểu luận

7.1.1. Resource-Allocation Graph and Wait-for Graph



(a) Đồ thị phân phối tài nguyên

(b) Đồ thị wait-for tương ứng

7.2. 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 *Pi* đang yêu cầu *k* cá thể nữa của loại tài nguyên *Rj*.

7.3. Giải thuật phát hiện deadlock(1)

- 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 *Allocationi* ≠ 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*] ≤ *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 mọi i thì hệ thống không có deadlock
  - Nếu *Finish*[*i*] = false, với một số các g/t *i*, $1 \le i \le n$, thì *Pi* bị deadlock, hệ thống ở trong trạng thái deadlock.

7.3. Giải thuật phát hiện deadlock(2): Ví dụ

- Giả thiết có:
  - 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 deadlock.

7.3. Giải thuật phát hiện deadlock(2): Ví dụ

- *P*2 yêu cầu thêm 1 cá thể loại *C*.

	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.
  - Deadlock xuất hiện, gồm các tiến trình *P*1, *P*2, *P*3, và *P*4.

7.3. Giải thuật phát hiện deadlock(3): Sử dụng

- Thời điểm và mức thường xuyên cần đến giải thuật phụ thuộc:
  - Deadlock có khả năng thường xuyên xảy ra như thế nào?
  - Có bao nhiêu tiến trình bị tác động khi deadlock xuất hiện
- Nếu giải thuật phát hiện deadlock 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" deadlock.
- Nếu phát hiện được deadlock, 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)

8. Phục hồi từ deadlock(1): Dừng tiến trình

- Hủy bỏ tất cả các tiến trình bị deadlock (có *Finish[i] = false*).
- Hủy bỏ một tiến trình tại một thời điểm đến khi chu trình deadlock 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?

8. Phục hồi từ deadlock(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 giá trị.

9. Phương pháp kết hợp xử lý deadlock

- 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 deadlock trongmỗi lớp.

Q & A

- List Câu hỏi