Chương 4. Deadlock

- Mô hình hệ thống
- Resource Allocation Graph(RAG)
- Phương pháp giải quyết deadlock
  - Deadlock prevention (ngăn chặn deadlock)
  - Deadlock avoidance (tránh deadlock)
  - Deadlock detection (phát hiện deadlock)
  - Deadlock recovery (phục hồi deadlock)

Vấn đề deadlock trong hệ thống

- *Tình huống*: một tập các process bị blocked,mỗi process giữ tài nguyên và đang chờ tài nguyên mà process khác trong tập đang có.

- Ví dụ 1
  - Giả sử hệ thống có 2 file trên đĩa.
  - P1 và P2 mỗi process đang mở một file và yêu cầu mở file kia.

- Ví dụ 2
  - Semaphore A và B, khởi tạo bằng 1

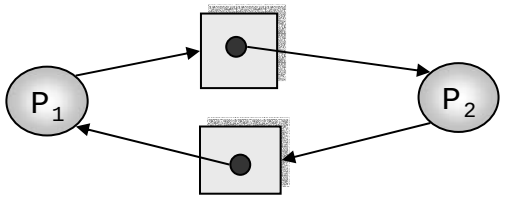
P0	P1
wait (A);	wait(B)
wait (B);	wait(A)

Mô hình hóa hệ thống

- Các loại tài nguyên kí hiệu R_1, R_2, . . ., R_m, bao gồm:
  - CPU cycle, không gian bộ nhớ, thiết bị I/O, file, semaphore, monitor,...
  - Mỗi loại tài nguyên Ri có Wi thực thể (instance).
- Quá trình sử dụng tài nguyên của mỗi process như sau
  - Yêu cầu (*request*): process phải chờ nếu yêu cầu không được đáp ứng ngay
  - Sử dụng (*use*)
  - Hoàn trả (*release*)
- Các tác vụ yêu cầu (*request*) và hoàn trả (*release*) đều là system call
  - Request/release device, open/close file, allocate/free memory
  - Wait/signal

Điều kiện tồn tại deadlock

- ***Mutual exclusion***: với mỗi tài nguyên, chỉ có một process sử dụng tại một thời điểm.
- ***Hold and wait***: một process vẫn sở hữu tài nguyên đã được cấp phát trong khi yêu cầu một tài nguyên khác.
- ***No preemption***: một tài nguyên không thể bị đoạt lại từ chính process đang sở hữu tài nguyên đó.
- ***Circular wait***: tồn tại một chu kỳ đóng các yêu cầu tài nguyên.



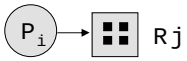
Deadlock có thể xảy ra nếu 4 điều kiện xuất hiện *đồng thời*.

Resource Allocation Graph(RAG)

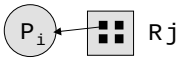
- RAG là đồ thị có hướng, tập đỉnh V và tập cạnh E.
  - Tập đỉnh V gồm 2 loại:
    - $P = \{P_1, P_2, \ldots, P_n\}$ (Tất cả process trong hệ thống)
    - $R = \{R_1, R_2, \ldots, R_m\}$ (Tất cả tài nguyên trong hệ thống)
  - Tập cạnh E gồm 2 loại
    - Request edge: cạnh có hướng từ Pi → Rj
    - Assignment edge: cạnh có hướng từ Rj → Pi
- Process

- Loại tài nguyên với 4 thực thể

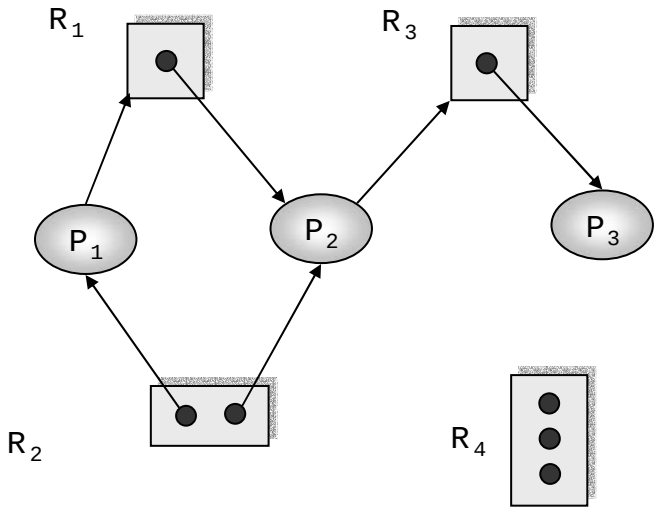
- Pi yêu cầu một thực thể của Rj



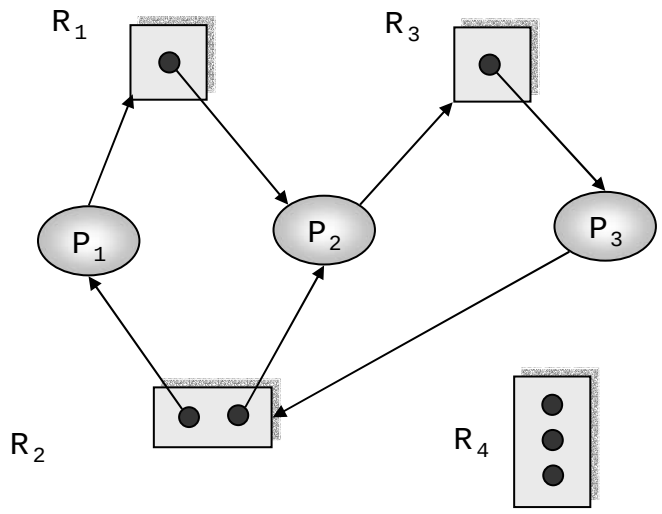
- Pi đang giữ một thực thể của Rj



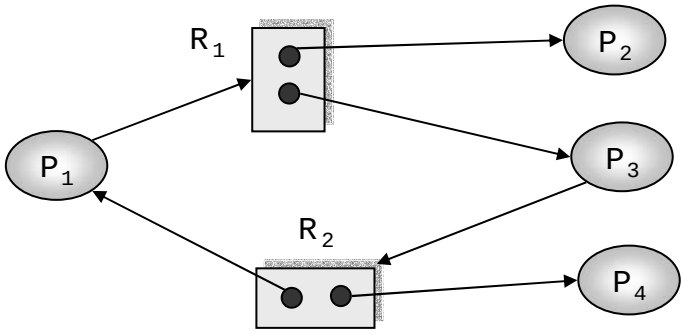
Ví dụ về RAG



RAG đang bị deadlock



Cycle RAG không deadlock



☺ RAG không tồn tại chu kỳ (cycle) ⇒ không có deadlock.

☹ RAG có ít nhất một chu kỳ (cycle)

–Nếu mỗi loại tài nguyên chỉ có một thực thể ⇒ deadlock.

–Nếu mỗi loại tài nguyên có nhiều thực thể ⇒ có thể xảy ra deadlock.

Các p.p giải quyết deadlock

- Dùng một giao thức (protocol) để ngăn chặn hoặc tránh deadlock, bảo đảm rằng hệ thống không bao giờ bị rơi vào tình trạnh deadlock.
  - Deadlock prevention
  - Deadlock avoidance
- Hệ thống có thể rơi vào trạng thái deadlock, sau đó phát hiện deadlock và phục hồi hệ thống.
- Bỏ qua mọi vấn đề, xem như không bao giờ có deadlock xảy ra trong hệ thống
  - ☺ Khá nhiều hệ điều hành sử dụng p.p này.
  - Deadlock không phát hiện, giải quyết ⇒ giảm hiệu suất của hệ thống. Cuối cùng, hệ thống có thể ngưng hoạt động và phải khởi động lại.

Deadlock Prevention

- Tìm cách ngăn chặn sao cho ít nhất một trong 4 điều kiện gây deadlock không xảy ra.

1) *Mutual Exclusion*: không cần thiết đối với sharable resource nhưng bắt buộc phải thỏa mãn đối với non-sharable resource ⇒ không hạn chế được.
2) *Hold and Wait*: sử dụng cơ chế "all-or-none"
   - Cách 1: bắt buộc mỗi process phải yêu cầu *toàn bộ tài nguyên cần thiết một lần*. Nếu có đủ tài nguyên thì hệ thống sẽ cấp phát, nếu không đủ tài nguyên thì process phải bị block
   - Cách 2: khi yêu cầu tài nguyên, process không được sở hữu bất kỳ tài nguyên nào. Nếu đang có thì phải trả lại trước khi yêu cầu
   - Khuyết điểm:
     - Hiệu suất sử dụng tài nguyên rất thấp
     - Có khả năng bị starvation

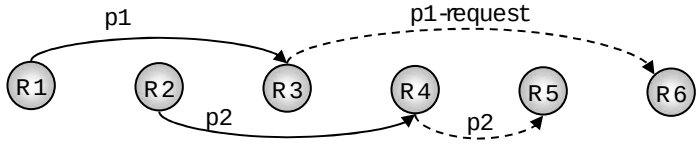
Deadlock Prevention (t.t)

3) *No Preemption*: nếu process A có sở hữu tài nguyên và đang yêu cầu tài nguyên khác nhưng tài nguyên này chưa đáp ứng được thì.
   - Cách 1: A phải *trả cho hệ thống mọi tài nguyên đang sở hữu*
     - A phải bắt đầu lại từ đầu, yêu cầu các tài nguyên đã bị đoạt lại và cả tài nguyên đang yêu cầu
   - Cách 2: Hệ thống sẽ kiểm tra tài nguyên mà A yêu cầu
     - Nếu tài nguyên là sở hữu của process đang yêu cầu và đợi thêm tài nguyên, tài nguyên này được hệ thống đoạt lại và cấp phát cho A.
     - Nếu tài nguyên là sở hữu của process không đợi tài nguyên, A phải đợi và tài nguyên của A bị đoạt lại. Tuy nhiên hệ thống chỉ đoạt lại các tài nguyên mà process khác yêu cầu

❑ Thường áp dụng cho tài nguyên có thể dễ dàng lưu và khôi phục trạng thái như CPU register, bộ nhớ, ...

Deadlock Prevention (t.t)

4) *Circular Wait*: các tài nguyên trong hệ thống được đánh số thứ tự tuyến tính
   - Ví dụ: F(tape drive) = 1, F(disk drive) = 5, F(Printer) = 12
     - F là hàm định nghĩa thứ tự dựa trên hiệu suất sử dụng của tài nguyên.
   - Cách 1: Bắt buộc mỗi process yêu cầu tài nguyên theo thứ tự tuyến tính tăng dần. Ví dụ
     - Chuỗi yêu cầu hợp lệ: tape drive → disk drive → Printer
     - Chuỗi yêu cầu không hợp lệ: disk drive → tape drive
   - Cách 2: Khi một process yêu cầu một tài nguyên Rj thì phải trả lại các tài nguyên Ri với F(Ri) > F(Rj)



Deadlock Avoidance

- Deadlock prevention sử dụng tài nguyên không hiệu quả.
- Deadlock avoidance chỉ dựa trên điều kiện thứ 4 để tránh deadlock mà vẫn đảm bảo hiệu suất sử dụng tài nguyên tối đa đến mức có thể.
- Yêu cầu mỗi process khai báo số lượng tài nguyên tối đa cần để thực hiện công việc
- Giải thuật deadlock-avoidance sẽ kiểm tra trạng thái cấp phát tài nguyên (*resource-allocation state*) để bảo đảm hệ thống không bao giờ rơi vào deadlock.
- Trạng thái cấp phát tài nguyên được định nghĩa dựa trên số tài nguyên còn lại, số tài nguyên đã được cấp phát và yêu cầu cực đại của các process

Trạng thái "safe" và "unsafe"

- Một trạng thái được gọi là "***safe***" nếu tồn tại ít nhất một cách mà trong một khoảng thời gian hữu hạn nào đó, hệ thống có thể cấp phát tài nguyên thỏa mãn cho tất cả process thực thi hoàn tất .
- Khi một process yêu cầu một tài nguyên đang sẵn có, hệ thống sẽ kiểm tra: nếu việc cấp phát này không dẫn đến tình trạng unsafe thì sẽ cấp phát ngay.

	holding	maximum need
A	1	4
B	4	6
C	5	8
Available = 2		

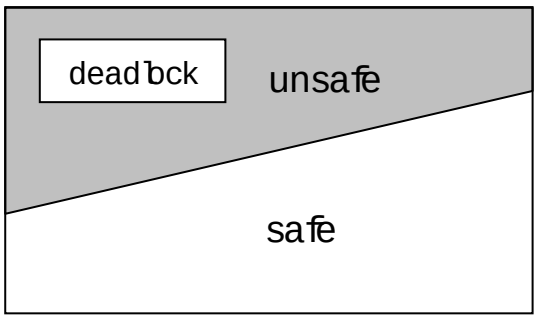
Trạng thái **safe**

	holding	maximum need
A	8	10
B	2	5
C	1	3
Available =1		

Trạng thái **unsafe**

Safe,unsafe và deadlock

- Nếu hệ thống đang ở trạng thái "safe" ⇒ không deadlock.
- Nếu hệ thống đang ở trạng thái "unsafe" ⇒ có khả năng dẫn đến deadlock.
- Deadlock avoidance ⇒ bảo đảm hệ thống không bao giờ đi đến trạng thái "unsafe".



Giải thuật Banker

- Áp dụng cho hệ thống cấp phát tài nguyên trong đó mỗi loại tài nguyên có thể có nhiều instance.
- Mô phỏng nghiệp vụ ngân hàng (banking)
- Một số giả thiết
  - Mỗi process phải khai báo số lượng tối đa tài nguyên mỗi loại mà process đó cần để hoàn tất công việc.
  - Khi process yêu cầu một tài nguyên thì có thể phải đợi mặc dù tài nguyên được yêu cầu đang có sẵn
  - Khi process đã có được đầy đủ tài nguyên thì phải hoàn trả trong một khoản thời gian hữu hạn nào đó.

Giải thuật Banker (t.t)

Một số giả thiết trong giải thuật Banker:

- **n** = số processes
- **m** = số loại tài nguyên

hạn chế của giải thuật Banker

- **Max**[n, m]
  - Max[i, j] = k ⇔ số instance cực đại mà Pi yêu cầu đối với Rj.
- **Available**[m]
  - Available[j] = k ⇔ loại tài nguyên Rj đang sẵn có k instance.
- **Allocation**[n, m]
  - Allocation[i,j] = k ⇔ Pi đã được cấp phát k instance của Rj.
- **Need**: ma trận n x m .
  - Need[i,j] = k ⇔ Pi cần thêm k instance của Rj.

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

Giải thuật kiểm tra trạng thái

1. Gọi Work và Finish là hai vector kích thước tương ứng là m, n. Khởi tạo ban đầu
   Work[i] := Available[i]
   Finish[j] = false, ∀j
2. Tìm i thỏa điều kiện sau
   (a) Finish [i] = false
   (b) $Need_i \leq Work$ (hàng thứ i của Need)
   Nếu không tồn tại i thoả điều kiện, đến bước 4.
3. Work := Work + $Allocation_i$
   Finish[i] := true
   quay về bước 2.
4. Nếu Finish[i] = true ∀i, hệ thống đang ở trạng thái safe

Độ phức tạp của giải thuật $O(m \cdot n^2)$

$Y \leq X \Leftrightarrow Y[i] \leq X[i]$, ví dụ $(0, 3, 2, 1) \leq (1, 7, 3, 2)$

Giải thuật cấp phát tài nguyên

Gọi $Request_i[m]$ là request vector của process P_i.

$Request_i[j] = k \Leftrightarrow P_i$ cần k instance của tài nguyên R_j.

1. Nếu $Request_i \le Need_i \Rightarrow$ đến bước 2. Ngược lại, báo lỗi vì process đã vượt quá giới hạn yêu cầu cực đại.
2. Nếu $Request_i \le Available \Rightarrow$ qua bước 3. Ngược lại, P_i phải chờ vì tài nguyên không còn đủ để cấp phát.
3. Giả định cấp phát tài nguyên đáp ứng yêu cầu của P_i thử cập nhật trạng thái hệ thống như sau:

   $Available := Available - Request_i$;

   $Allocation_i := Allocation_i + Request_i$;

   $Need_i := Need_i - Request_i$;

   Nếu trạng thái là safe $\Rightarrow$ tài nguyên được cấp thực sự cho P_i.

   Nếu unsafe $\Rightarrow P_i$ phải đợi. Phục hồi trạng thái

   $Available := Available + Request_i$;

   $Allocation_i := Allocation_i - Request_i$;

   $Need_i := Need_i + Request_i$;

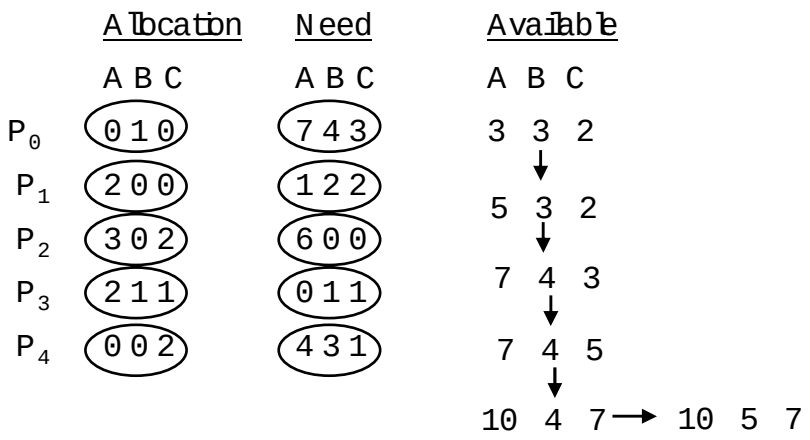
Giải thuật Banker – Ví dụ (t.t)

- Có 5 process $P_0 - P_4$
- Có 3 loại tài nguyên: *A* (có 10 instance), *B* (5 instance) và *C* (7 instance).
- Sơ đồ cấp phát trong hệ thống tại thời điểm T_0

	Allocation			**Max**			**Available**			**Need**			
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	
P_0	0	1	0	7	5	3	3	3	2	7	4	3	❺
P_1	2	0	0	3	2	2				1	2	2	❶
P_2	3	0	2	9	0	2				6	0	0	❹
P_3	2	1	1	2	2	2				0	1	1	❷
P_4	0	0	2	4	3	3				4	3	1	❸

Kiểm tra sự an toàn

Chuỗi cấp phát an toàn < P_1, P_3, P_4, P_2, P_0>



Ví dụ:

- Yêu cầu (1,0,2) của P_1 có thỏa được không? Kiểm tra điều kiện Request ≤ Available

  (1,0,2) ≤ (3,3,2) = *TRUE*

- Kiểm tra trạng thái hiện tại có phải là safe hay không?

	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	2	0	0	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

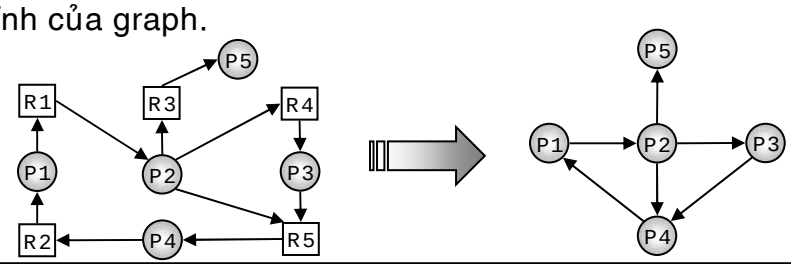
- BT: yêu cầu (3,3,0) của P_4, yêu cầu (0,2,0) của P_0 có thể thỏa mãn hay không?

Deadlock Detection

- Chấp nhận xảy ra deadlock trong hệ thống, kiểm tra trạng thái hệ thống bằng giải thuật phát hiện deadlock.
- Nếu có deadlock thì tiến hành phục hồi hệ thống
- Các giải thuật phát hiện deadlock thường sử dụng mô hình RAG.
- Có hai mô hình hệ thống cấp phát tài nguyên được khảo sát
  1. Single-Instance: mỗi loại tài nguyên chỉ có một instance
  2. Multiple-Instance: mỗi loại tài nguyên có thể có nhiều instance

Mô hình Single-Instance

- Sử dụng wait-for graph (một biến thể của RAG), trong đó, các node của graph là các process.
  - Wait-for graph được dẫn xuất từ RAG bằng cách bỏ các node biểu diễn tài nguyên và ghép các cạnh tương ứng.
  - Pi $\rightarrow$ Pj $\Leftrightarrow$ Pi đang chờ tài nguyên từ Pj
- Một giải thuật được gọi định kỳ (!) để kiểm tra có tồn tại chu kỳ (cycle) trong wait-for graph hay không? Giải thuật phát hiện chu kỳ có độ phức tạp là $O(n^2)$, với n là số đỉnh của graph.



Mô hình Multiple-Instance

- Mô hình wait-for graph không áp dụng được cho trường hợp mỗi loại tài nguyên có nhiều instance.
- Các cấu trúc dữ liệu dùng trong giải thuật phát hiện deadlock
  - **Available**[m]: # instance sẵn có của mỗi loại tài nguyên
  - **Allocation**[n,m]: # instance đã cấp phát cho mỗi process
  - **Request**[n,m]: yêu cầu hiện tại của process. Request [i,j] = k ⇔ Pi đang yêu cầu thêm k instance của Rj

Detection Algorithm

1. Gọi *Work* và *Finish* là vector kích thước *m* và *n*:
   (a) *Work* := *Available*
   (b) Với i = 1,2, …, n, nếu $Allocation_i \neq 0 \Rightarrow Finish[i] = false$
   ngược lại $\Rightarrow Finish[i] = true$
2. Tìm i thỏa mãn:
   Finish[i] = false và $Request_i \leq Work$,
   Nếu không tồn tại *i* , đến bước 4.
3. $Work = Work + Allocation_i$
   Finish[i] = true
   quay về bước 2.

Độ phức tạp của giải thuật $O(m.n^2)$

4. $\exists i, 1 \leq i \leq n$: *Finish[i]* = false ⇒ hệ thống đang ở trạng thái deadlock. Hơn thế nữa, *Finish[i] = false* ⇒ P_i bị deadlock.

Detection Algorithm – Ví dụ

- Có 5 processes P_0 – P_4
- 3 loại tài nguyên A (7 instance), *B* (2 instance), *C* (6 instance).

	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			

- Cấp phát theo thứ tự <P_0, P_2, P_3, P_1, P_4> sẽ có kết quả là $\forall i$ Finish[i] = true $\Rightarrow$ hệ thống không bị deadlock.

Detection Algorithm – Ví dụ (t.t)

- P_2 yêu cầu thêm một instance của *C*. Ma trận Request như sau:

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

- Trạng thái của hệ thống là gì?
  - Có thể thu hồi tài nguyên đang sở hữu bởi process P_0 nhưng vẫn không đủ đáp ứng yêu cầu của các process khác.

  $\Rightarrow$ Tồn tại deadlock, bao gồm các processe P_1, P_2, P_3, và P_4.

Deadlock Recovery

- Phục hồi hệ thống bị deadlock chủ yếu là bẻ gãy chu kỳ wait-for của các process bị deadlock.
  - Hủy bỏ tất cả process bị deadlock .
  - Hủy bỏ lần lượt từng process và thu hồi tài nguyên cho đến khi không còn deadlock.
- Dựa trên cơ sở nào để hủy process ?
  - Độ ưu tiên của process ?
  - Thời gian đã thực thi của process và thời gian còn lại ?
  - Loại tài nguyên mà process đã sử dụng ?
  - Tài nguyên mà process cần để hoàn tất công việc ?
  - Số lượng processes cần hủy bỏ ?
  - Process là interactive process hay batch process?
  - ...

Thu hồi tài nguyên

- Đoạt tài nguyên từ một process, cấp phát cho process khác cho đến khi không còn deadlock nữa.
- Các vấn đề trong chiến lược thu hồi tài nguyên:
  - Chọn "nạn nhân" – tối thiểu chi phí (có thể dựa trên số tài nguyên sở hữu, thời gian CPU đã tiêu tốn,...)
  - Rollback – quay trở về trạng thái safe, bắt đầu các process từ trạng thái đó. Gồm có total rollback và check-point rollback
    - Hệ thống cần lưu giữ một số thông tin về trạng thái các process đang thực thi
  - Starvation – phải bảo đảm không có process sẽ luôn luôn bị đoạt tài nguyên mỗi khi deadlock xảy ra

Phương pháp tổng hợp

- Phân hoạch tài nguyên thành các nhóm có phân cấp

  $\Rightarrow$ dùng phương pháp linear ordering để phòng chống deadlock đối với các nhóm. Trong mỗi nhóm, dùng giải thuật phù hợp nhất để giải quyết deadlock.
- Một số ví dụ
  - Swappable space: khối bộ nhớ hoặc thiết bị lưu trữ phụ dùng làm bộ nhớ tráo đổi tạm (swap memory)
    - Hold-and-Wait Prevention + Avoidance
  - Process resource: assignable device như tape drive, file,...
    - Avoidance hoặc Resource Ordering
  - Main memory: memory page/segment
    - Preemption
  - Internal resource: PCB, I/O channel,…
    - Resource Ordering