Bài 6
Deadlock

Nguyên nhân deadlock

- Các tiến trình cùng hoạt động có thể cùng sử dụng các loại tài nguyên:
  - Tài nguyên có nhiều loại: chia sẻ được, không chia sẻ được.
  - Mỗi loại tài nguyên có một hoặc nhiều thể hiện.
- Mỗi P hoạt động gồm một chuỗi thao tác:
  - Đòi hỏi các tài nguyên, nếu tài nguyên không có sẵn → P phải đợi.
  - Sử dụng các tài nguyên được cấp.
  - Giải phóng các tài nguyên sau khi sử dụng.
- Nếu các P sử dụng chung ít nhất 2 tài nguyên có thể dẫn đến hệ thống gặp "nguy hiểm", ví dụ?

Nguyên nhân deadlock

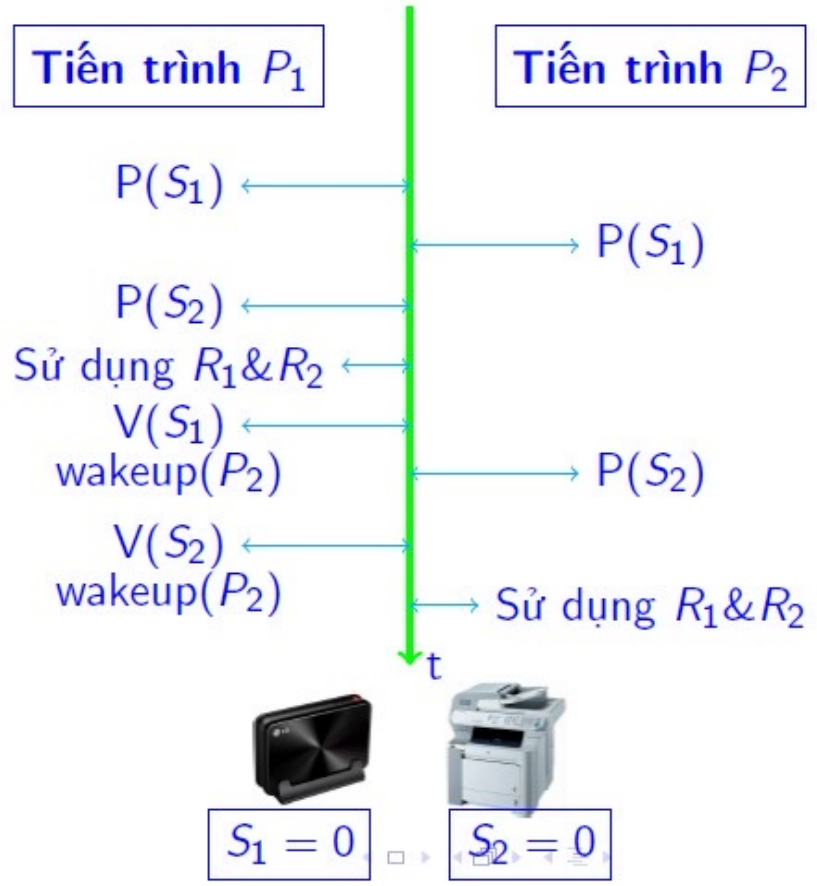
- Ví dụ:
  - Hệ thống có P1 và P2
  - P1 & P2 sử dụng chung tài nguyên R1 & R2
  - R1&R2 đều chỉ có 1 thể hiện.
  - R1 điều độ bởi semaphore S1
  - R2 điều độ bởi semaphore S2

$P(S_1)$
$P(S_2)$
{Sử dụng R_1&R_2}
$V(S_1)$
$V(S_2)$

Tiến trình P_1

$P(S_1)$
$P(S_2)$
{Sử dụng R_1&R_2}
$V(S_1)$
$V(S_2)$

Tiến trình P_2



Nguyên nhân deadlock

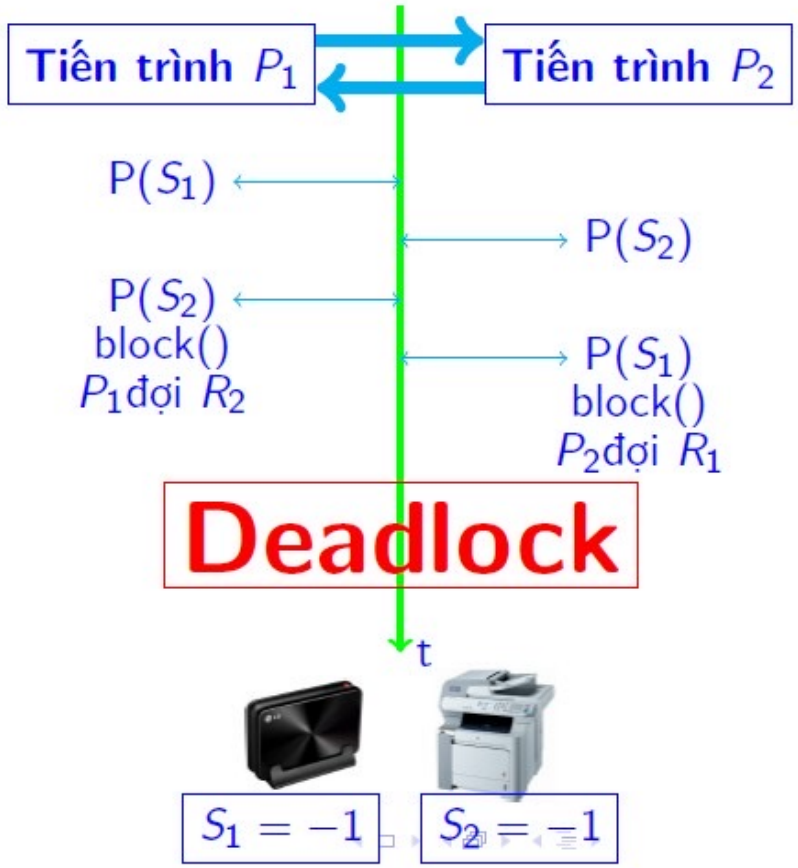
- Ví dụ:
  - Hệ thống có P1 và P2
  - P1 & P2 sử dụng chung tài nguyên R1 & R2
  - R1&R2 đều chỉ có 1 thể hiện.
  - R1 điều độ bởi semaphore S1
  - R2 điều độ bởi semaphore S2

$P(S_1)$
$P(S_2)$
{Sử dụng R_1&R_2}
$V(S_1)$
$V(S_2)$

Tiến trình P_1

$P(S_2)$
$P(S_1)$
{Sử dụng R_1&R_2}
$V(S_1)$
$V(S_2)$

Tiến trình P_2

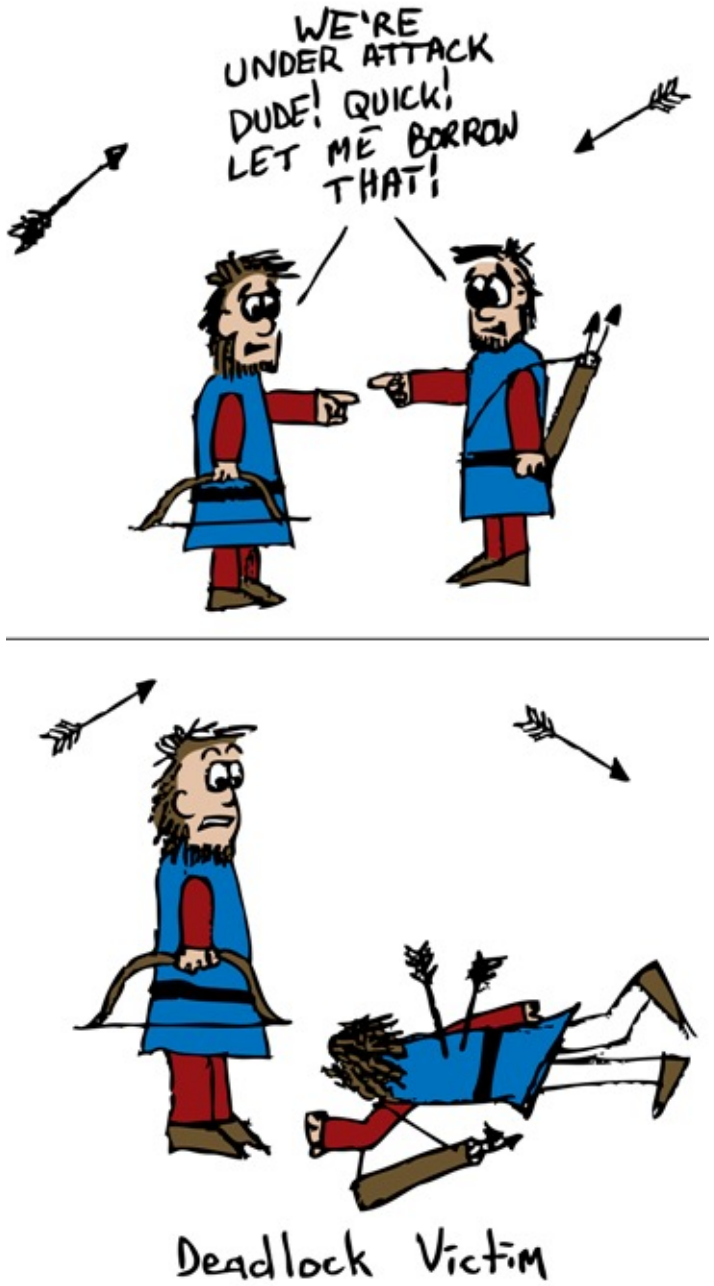


Khái niệm deadlock

- Deadlock là là hiện tượng trong hệ thống xuất hiện một tập các P mà mỗi P trong tập hợp đều chờ đợi được cấp phát tài nguyên, nhưng tài nguyên đó đang bị một P khác trong tập này chiếm giữ.
- Nói cách khác, deadlock là trạng thái trong hệ thống có ít nhất hai P đang dừng chờ lẫn nhau và chúng không thể hoạt động tiếp được. Sự chờ đợi đó có thể kéo dài vô hạn nếu không có sự tác động từ bên ngoài.
- Nguyên nhân:
  - Các P tranh chấp nhau tài nguyên không chia sẻ được → giải pháp khóa tài nguyên (mutex, semaphore, monitor).
  - Các P không được cấp phát đủ tài nguyên → block → không trả lại tài nguyên đang giữ.
  - Càng nhiều P bị block → càng nhiều tài nguyên bị giữ → deadlock càng nghiêm trọng.

Khái niệm deadlock

- Deadlock



Điều kiện xảy ra Deadlock

- Coffman, Elphick và Shoshani đã đưa ra 4 điều kiện cần có làm xuất hiện tắc nghẽn:
  - 1. Có sử dụng tài nguyên không thể chia sẻ (Mutual exclusion), gọi là tài nguyên găng (critical resource):
    - ✓ Mỗi thời điểm, một tài nguyên không thể chia sẻ được hệ thống cấp phát chỉ cho một tiến trình.
    - ✓ Khi tiến trình sử dụng xong tài nguyên này, hệ thống mới thu hồi và cấp phát tài nguyên cho tiến trình khác.
  - 2. Sự chiếm giữ và yêu cầu thêm tài nguyên (Wait for/Hold and wait):
    - ✓ Các tiến trình tiếp tục chiếm giữ các tài nguyên đã cấp phát cho nó trong khi chờ được cấp phát thêm một số tài nguyên mới.

Điều kiện xảy ra Deadlock

- Coffman, Elphick và Shoshani đã đưa ra 4 điều kiện cần có làm xuất hiện tắc nghẽn:
  - 3. Không thu hồi tài nguyên từ tiến trình đang giữ chúng (No preemption):
    - ✓ Tài nguyên không thể được thu hồi từ tiến trình đang chiếm giữ chúng trước khi tiến trình này sử dụng chúng xong.
  - 4. Tồn tại một chu trình trong đồ thị cấp phát tài nguyên (Circular wait):
    - ✓ Có ít nhất hai tiến trình chờ đợi lẫn nhau: tiến trình này chờ được cấp phát tài nguyên đang bị tiến trình kia chiếm giữ và ngược lại.

Điều kiện xảy ra Deadlock

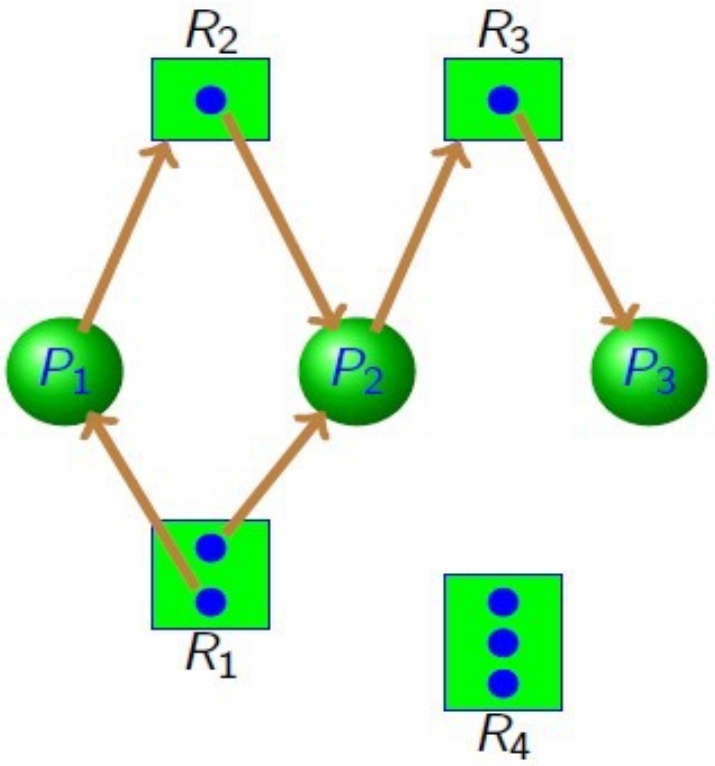
- Coffman, Elphick và Shoshani đã đưa ra 4 điều kiện cần có làm xuất hiện tắc nghẽn:
  1. Mutual exclusion
  2. Hold & wait
  3. No preemption
  4. Circular wait
- Phải đáp ứng đủ 4 điều kiện trên mới xảy ra ra deadlock.
- Có thể sử dụng đồ thị để mô hình hóa việc cấp phát tài nguyên.

Đồ thị cấp phát tài nguyên

- Dùng để mô hình hóa tình trạng deadlock trong hệ thống.
- Là đồ thị có hướng gồm tập đỉnh V và tập cạnh E.
- Tập đỉnh V gồm:
  - P = {P1; P2; … ; Pn} thể hiện các tiến trình trong hệ thống, biểu diễn bằng hình tròn.
  - R = {R1; R2; …; Rn} thể hiện các kiểu tài nguyên trong hệ thống, biểu diễn bằng hình vuông.
- Tập cạnh E:
  - Cung yêu cầu: đi từ tiến trình Pi đến tài nguyên Rj: Pi → Rj
  - Cung sử dụng(cấp phát): đi từ tài nguyên Rj đến tiến trình Pi: Rj → Pi

Đồ thị cung cấp tài nguyên

- Khi một tiến trình Pi yêu cầu tài nguyên Rj
  - Cung yêu cầu Pi→ Rj được thêm vào đồ thị.
  - Nếu yêu cầu được thỏa mãn, cung yêu cầu trở thành cung sử dụng Rj→Pi
  - Khi tiến trình Pi giải phóng tài nguyên Rj, cung sử dụng Rj→Pi bị xóa khỏi đồ thị.

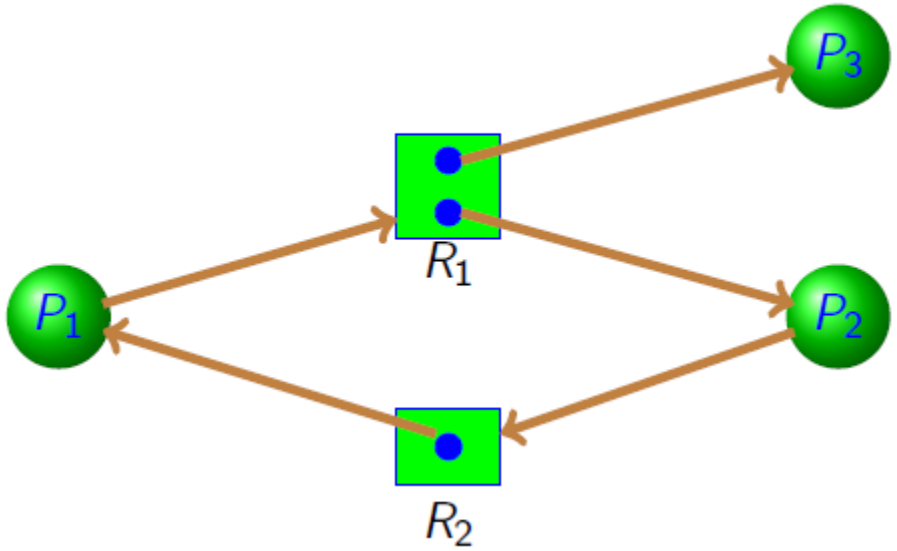


- Trạng thái hệ thống
  - 3 tiến trình P_1, P_2, P_3
  - 4 tài nguyên R_1, R_2, R_3, R_4

Đồ thị cung cấp tài nguyên

- Đồ thị không chứa chu trình → Không deadlock
- Nếu đồ thị chứa chu trình:
  - Nếu tài nguyên chỉ có 1 đơn vị → deadlock
  - Nếu tài nguyên có nhiều hơn một đơn vị → có khả năng xảy ra deadlock

Đồ thị có chu trình nhưng hệ thống không bế tắc



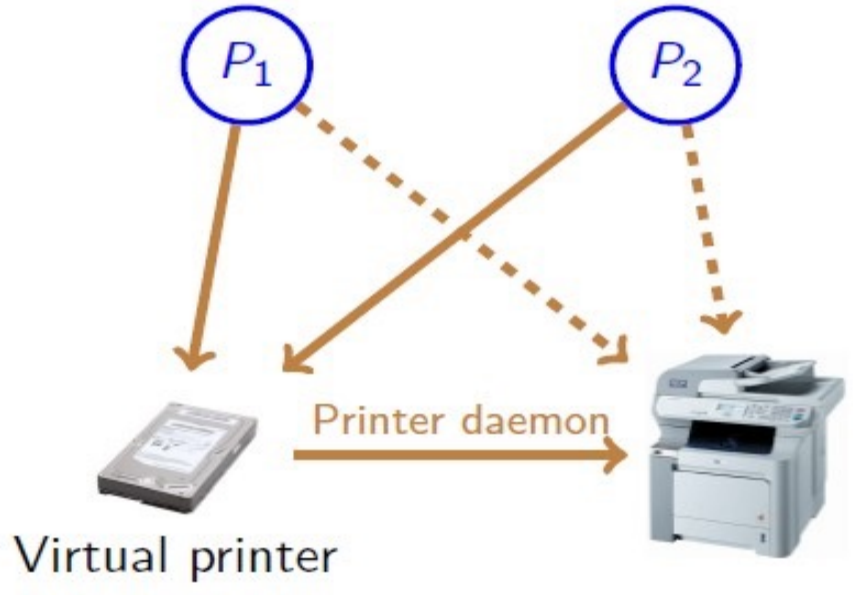
Phương pháp xử lý deadlock

- Ngăn chặn hoặc phòng tránh deadlock:
  - **Ngăn chặn**: Sử dụng các giải pháp ngăn chặn để đảm bảo hệ thống không bao giờ xảy ra deadlock. Tốn kém, áp dụng đối với hệ thống hay deadlock hoặc tổn thất do deadlock lớn.
  - **Phòng tránh**: kiểm tra từng yêu cầu tài nguyên của P và không chấp nhận yêu cầu nếu việc cấp tài nguyên có khả năng dẫn đến deadlock. Cần các thông tin phụ trợ, áp dụng với hệ thống ít xảy ra deadlock nhưng tổn hại lớn.
- **Cho phép xảy ra** deadlock và tìm cách sửa chữa (nhận biết và khắc phục)
  - Hệ thống hoạt động có thể dẫn đến deadlock → kiểm tra định kỳ, nếu có deadlock thì áp dụng các biện pháp loại bỏ deadlock, áp dụng với hệ thống ít xảy ra deadlock
- **Bỏ qua deadlock**, xem như hệ thống không bao giờ deadlock:
  - Xác suất xảy ra deadlock nhỏ hoặc chi phí giải quyết cao → giải pháp của hầu hết OS

Phương pháp xử lý deadlock → Ngăn chặn

- Nguyên tắc: tác dụng vào 1 trong 4 điều kiện cần của deadlock để nó không xảy ra:
  1. Mutual exclusion
  2. Hold & wait
  3. No preemption
  4. Circular wait

- Nguyên tắc: tác dụng vào 1 trong 4 điều kiện cần của deadlock để nó không xảy ra:
  1. Mutual exclusion (tài nguyên không chia sẻ): giảm mức độ critical resource của hệ thống
     - ✓ Gần như không thể tránh được điều kiện này vì bản chất tài nguyên gần như cố định.
     - ✓ Đối với một số tài nguyên, người ta có thể dùng các cơ chế SPOOL (Simultaneous peripheral operations online) để biến đổi thành tài nguyên có thể chia sẻ.



- Chỉ *printer daemon* mới làm việc với máy in ⇒ Bế tắc cho tài nguyên máy in bị hủy bỏ
- Không phải tài nguyên nào cũng dùng kỹ thuật SPOOL được

Phương pháp xử lý deadlock → Ngăn ngừa

- Nguyên tắc: tác dụng vào 1 trong 4 điều kiện cần của deadlock để nó không xảy ra:
  2. Hold & wait: phải bảo đảm rằng mỗi khi tiến trình yêu cầu thêm một tài nguyên thì nó không chiếm giữ các tài nguyên khác, dùng 1 trong 2 cơ chế sau:
     - ✓ Tiến trình phải *yêu cầu tất cả các tài nguyên cần thiết trước khi bắt đầu xử lý*
       - Khó có thể ước lượng chính xác tài nguyên cần sử dụng.
       - Nếu chỉ sử dụng tài nguyên ở giai đoạn cuối?
       - Tổng tài nguyên cần dùng vượt khả năng hệ thống?
     - ✓ Khi tiến trình yêu cầu một tài nguyên mới và bị từ chối, nó phải *giải phóng các tài nguyên đang chiếm giữ, sau đó lại được cấp phát trở lại* cùng lần với tài nguyên mới.
       - Tốc độ thực thi chậm.
       - Dữ liệu được lưu trong các tài nguyên tạm giải phóng xử lý thế nào?

Phương pháp xử lý deadlock → Ngăn ngừa

- Nguyên tắc: tác dụng vào 1 trong 4 điều kiện cần của deadlock để nó không xảy ra:
  3. No preemption: cho phép hệ thống được thu hồi tài nguyên từ các tiến trình bị khoá và cấp phát trở lại cho tiến trình khi nó thoát khỏi tình trạng bị khóa.
     - ✓ Với một số loại tài nguyên, việc thu hồi sẽ rất khó khăn vì vi phạm sự toàn vẹn dữ liệu .
  4. Circular wait: sắp xếp thứ tự cho tất cả các loại tài nguyên và đòi hỏi mỗi P phải yêu cầu tài nguyên theo thứ tự đó.
     - ✓ Các tiến trình khi yêu cầu tài nguyên phải tuân thủ quy định : khi tiến trình đang chiếm giữ tài nguyên Ri thì chỉ có thể yêu cầu các tài nguyên Rj nếu F(Rj) > F(Ri).

Phương pháp xử lý deadlock → Phòng tránh

- Phòng tránh deadlock (Deadlock Avoidance) đòi hỏi OS có trước thông tin bổ sung liên quan đến R mà một P sẽ yêu cầu và sử dụng trong suốt quá trình thực thi.
- Yêu cầu mỗi P khai báo số lượng tối đa mỗi loại tài nguyên mà nó cần. Quyết định cấp phát R cho P được thực hiện hay không phụ thuộc vào việc cấp phát đó có nguy cơ dẫn đến tắc nghẽn không.
- Giải thuật tránh tắc nghẽn sẽ kiểm tra trạng thái cấp phát tài nguyên (resource – allocation state) để bảo đảm hệ thống không rơi vào deadlock. Tuân thủ 2 nguyên tắc:
  - Không cho P bắt đầu hoạt động nếu yêu cầu R của nó có nguy cơ dẫn đến deadlock.
  - Không được cung cấp thêm R cho P nếu sự cung cấp này có nguy cơ dẫn đến deadlock.

Phương pháp xử lý deadlock → Phòng tránh

- Một số khái niệm:
  - Trạng thái của hệ thống là sự cấp phát R hiện tại cho các P.
  - Trạng thái an toàn: trạng thái A là an toàn nếu hệ thống có thể thỏa mãn các nhu cầu tài nguyên (cho đến tối đa) của mỗi tiến trình theo một thứ tự nào đó mà vẫn không xảy ra tắc nghẽn.
  - Một chuỗi cấp phát an toàn: một thứ tự của các P sao cho hệ thống có thể thỏa mãn tất cả các nhu cầu R của mỗi P theo một thứ tự nào đó mà vẫn không xảy ra deadlock.
  - Một trạng thái của hệ thống được gọi là không an toàn (unsafe) nếu không tồn tại một chuỗi cấp phát an toàn.
- Nếu:
  - Hệ thống ở trạng thái an toàn → không có deadlock
  - Hệ thống ở trạng thái không an toàn → có thể có deadlock

Phương pháp xử lý deadlock → Phòng tránh

- Phòng tránh deadlock: đảm bảo hệ thống sẽ không bao giờ bước vào trạng thái không an toàn:
  - Nếu mỗi loại tài nguyên có 1 thể hiện (instance): sử dụng giải thuật đồ thị phân phối tài nguyên.
  - Nếu mỗi loại tài nguyên có nhiều thể hiện (instance): sử dụng giải thuật nhà băng.
- Chiến lược cấp phát: chỉ thỏa mãn yêu cầu tài nguyên của tiến trình khi trạng thái kết quả là an toàn.

❑ Giải thuật xác định trạng thái an toàn: Safety Algorithm

Bước 1:

Tính bảng Need = Max – Allocation

Work = Available

Finish[i] = false với tất cả các tiến trình Pi

Bước 2:

Tìm tiến trình P_i thỏa mãn cả 2 điều kiện:

- Finish [i] = false
- $Need_i$ <= Work

(Nếu tìm không có thì chuyển sang bước 4)

Bước 3:

Với mỗi P_i đã được chọn:

$Work_{mới} = Work_{cũ} + Allocation_i$

Finish[i] = true

Tiếp tục lặp lại bước 2 với các P_i kế tiếp

Bước 4:

Nếu Finish[i] = true với tất cả các tiến trình Pi thì hệ thống ở trạng thái an toàn → cần chỉ ra chuỗi an toàn.

Ngược lại, hệ thống ở trạng thái không an toàn.

Giải thuật xác định trạng thái an toàn: Safety Algorithm

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	6	5	4	2	2	1	2	2	2
P2	5	4	6	3	1	3			
P3	4	3	5	2	1	3			
P4	3	5	2	1	3	1			

	Need		
	R1	R2	R3
P1	4	3	3
P2	2	3	3
P3	2	2	2
P4	2	2	1

B1. Work=(2,2,2)

Finish[i]=false với mọi i từ 1 đến 4.

B2: Tìm thấy tiến trình P3 thỏa mãn 2 điều kiện

Finish[3]=false

$Need_3$ <= Work

B3: Work= Work + $Alloc_3$ = (2,2,2)+(2,1,3) =(4,3,5)

Finish[3]=true

(xong P3 tiếp tục lặp lại từ bước 2, kế tiếp là P4)

B2: tìm thấy tiến trình P4 thỏa mãn 2 điều kiện:

Finish[4] = false

$Need_4$ <= Work

B3: Work= Work + $Alloc_4$ = (4,3,5)+(1,3,1) = (5,6,6)

Finish[4]=true

(do vẫn còn Finish[i]=false nên tiếp tục lặp lại từ P1)

B2: tìm thấy tiến trình P1 thỏa mãn 2 điều kiện (tương tự các bước trên)

B3: Work = Work + $Alloc_1$ = (5,6,6) + (2,2,1) = (7,8,7)

Finish[1]=true

(tiếp tục xét các tiến trình kế tiếp trong danh sách có Finish[i]=false)

B2: tìm thấy tiến trình P2 thỏa mãn 2 điều kiện (tương tự các bước trên)

B3: Work= Work + $Alloc_2$ = (7,8,7) + (3,1,3) =(10,9,10)

Finish[2]=true

B2: Không tìm thấy tiến trình nào thỏa mãn điều kiện

B4: Finish[i]=true với mọi i từ 1 đến 4 do vậy hệ thống ở trạng thái an toàn

- Giải thuật xác định trạng thái an toàn: Safety Algorithm

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	6	5	4	2	2	1	2	2	2
P2	5	4	6	3	1	3			
P3	4	3	5	2	1	3			
P4	3	5	2	1	3	1			

	Need		
	R1	R2	R3
P1	4	3	3
P2	2	3	3
P3	2	2	2
P4	2	2	1

Kết luận: sau khi thực hiện giải thuật Safety Algorithm, tìm được chuỗi an toàn: P3 → P4 → P1 → P2. Do đó, hệ thống đang ở trạng thái an toàn.

Giải thuật xác định trạng thái an toàn: Safety Algorithm

Cho trạng thái hệ thống tại một thời điểm như hình dưới đây, hãy cho biết hệ thống có ở trạng thái an toàn không, vì sao?

Process	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	1	2
P2	6	1	3	2	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

Giải thuật xác định trạng thái an toàn: Safety Algorithm

Process	Max			Allocation			Available			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	1	2	2	2	2
P2	6	1	3	2	1	1				4	0	2
P3	3	1	4	2	1	1				1	0	3
P4	4	2	2	0	0	2				4	2	0

- Bước chuẩn bị:

  + Tính bảng Need = Max – Allocation

  + Ghi nhận tất cả các tiến trình đều chưa xét (Finish [i] = false)

  + Work = Available = (4, 1, 2)

- Bắt đầu chạy giải thuật Safety:

  + P1: $Need_1$ > Work → bỏ qua

  + P2: $Need_2$ < Work → chọn P2:

    * Work = Work + $Alloc_2$ = (4,1,2) + (2,1,1) = (6,2,3)

    * Đánh dấu P2 đã xét.

  + P3: $Need_3$ < Work → chọn P3:

    * Work = Work + $Alloc_3$ = (6,2,3) + (2,1,1) = (8,3,4)

    * Đánh dấu P3 đã xét.

Giải thuật xác định trạng thái an toàn: Safety Algorithm

Process	Max			Allocation			Available			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	1	2	2	2	2
P2	6	1	3	2	1	1				4	0	2
P3	3	1	4	2	1	1				1	0	3
P4	4	2	2	0	0	2				4	2	0

+ P4: $Need_4$ < Work → chọn P_4:

  * Work = Work + $Alloc_4$ = (8,3,4) + (0,0,2) = (8,3,6)

  * Đánh dấu P_4 đã xét.

+ P1: $Need_1$ < Work → chọn P_1:

  * Work = Work + $Alloc_1$ = (8,3,6) + (1,0,0) = (9,3,6)

  * Đánh dấu P_1 đã xét.

Ghi chú: sau khi chạy giải thuật Safety, cần kiểm tra bảng Work cuối cùng có khớp với tổng tài nguyên của hệ thống (Allocation + Available).

- Tất cả các Pi đều đã xét, hệ thống ở trạng thái an toàn do tìm được 1 chuỗi an toàn là P2 → P3 → P4 → P1

Giải thuật yêu cầu tài nguyên: Resource-Request Algorithm

Giả sử tiến trình P_i yêu cầu thêm tài nguyên $Request_i$.

1. Nếu $Request_i$ <= Need[i], đến bước 2

   Ngược lại, xảy ra tình huống lỗi do P_i đã xin R quá yêu cầu tối đa của nó.

2. Nếu $Request_i$ <= Available, đến bước 3

   Ngược lại, P_i phải chờ do R của hệ thống chưa sẵn sàng cấp phát.

3. Giả sử hệ thống đã cấp phát cho P_i các R mà nó yêu cầu và cập nhật tình trạng hệ thống như sau:

   $Available = Available - Request_i;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

Nếu trạng thái kết quả là an toàn, lúc này các tài nguyên trên sẽ được cấp phát thật sự cho Pi. Ngược lại Pi phải chờ.

Giải thuật yêu cầu tài nguyên: Resource-Request Algorithm

Nếu tiến trình P2 yêu cầu 4 cho R1, 1 cho R3. Hãy cho biết yêu cầu này có thể đáp ứng mà bảo đảm không xảy ra tình trạng deadlock hay không?

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	2	2
P2	6	1	3	2	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

- Giải thuật yêu cầu tài nguyên: Resource-Request Algorithm

Request$_{P2}$ (4, 0, 1) → safe?

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	2	2
P2	6	1	3	2	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

Need			
	R1	R2	R3
P1	2	2	2
P2	4	0	2
P3	1	0	3
P4	4	2	0

	Need (updated)			Allocation (updated)			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	0	2	1
P2	0	0	1	6	1	2			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

- Chạy Safety Algorithm → chuỗi an toàn P2 -> P3 -> P4 -> P1 → có thể đáp ứng yêu cầu của P2.

Ví dụ:

Cho hệ thống có 5 tiến trình và 3 loại tài nguyên (A, B, C). A có 10 thể hiện, B có 5 thể hiện và C có 7 thể hiện. Trạng thái hệ thống tại thời điểm t_0 như hình bên.

- Hãy tính bảng Need.
- Hệ thống có ở trạng thái an toàn không? Nếu có, hãy tìm 1 chuỗi an toàn.
- Nếu P1 request (1,0,2) thì hệ thống có an toàn?

Process	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	

- Ví dụ: tính bảng Need

Process	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	

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

Phương pháp xử lý deadlock → Phòng tránh

- Ví dụ: Kiểm tra hệ thống an toàn (Safety Algorithm)

Process	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	

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ > Work (7,4,3 > 3,3,2)

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $Need_1$ < Work (1,2,2 < 3,3,2)

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0)

Work =	A	B	C
	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ > Work (6, 0, 0 > 5,3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work (0, 1, 1 < 5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)

Work =	A	B	C
	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ < Work (4, 3, 1 < 7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)

Work =	A	B	C
	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1 , 0)

Work =	A	B	C
	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ < Work (6, 0, 0 < 7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)

Work =	A	B	C
	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

- Ví dụ: Hệ thống có an toàn khi P xin R (Resource-Request Algorithm)

Process	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	

A B C
$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

1, 0, 2 < 1, 2, 2
$Request_1 < Need_1$ ✓

Step 2

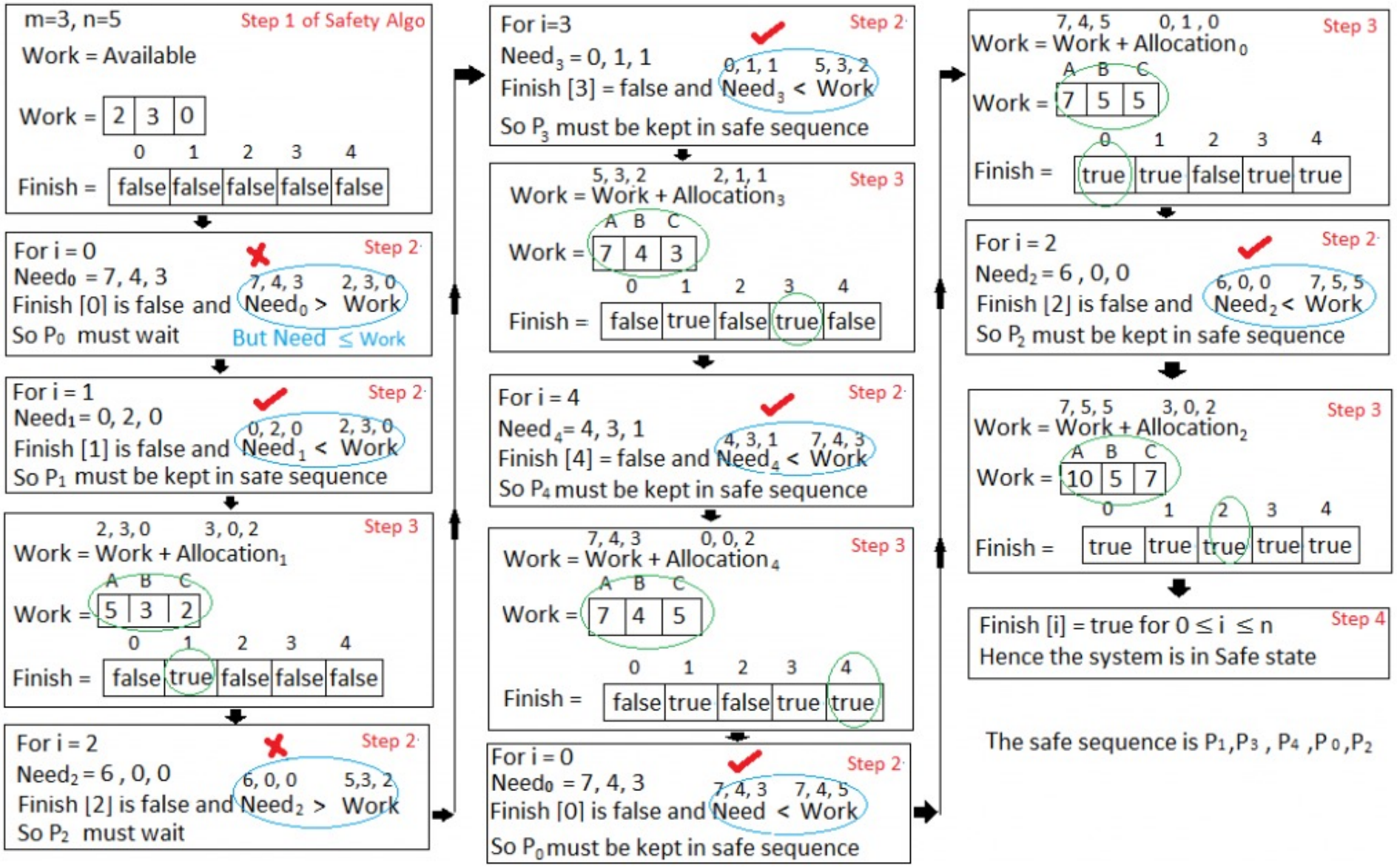
1, 0, 2 < 3, 3, 2
$Request_1 < Available$ ✓

Step 3

$Available = Available - Request_1$
$Allocation_1 = Allocation_1 + Request_1$
$Need_1 = Need_1 - Request_1$

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

- Ví dụ: Hệ thống có an toàn khi P xin R (Resource-Request Algorithm)



Phương pháp xử lý deadlock → Hiệu chỉnh/Phục hồi

- Khi đã phát hiện được tắc nghẽn, có hai lựa chọn chính để hiệu chỉnh tắc nghẽn :
  - Đình chỉ hoạt động của các tiến trình liên quan: Cách tiếp cận này dựa trên việc thu hồi lại các tài nguyên của những tiến trình bị kết thúc. Có thể sử dụng một trong hai phương pháp sau:
    - ✓ Đình chỉ tất cả các tiến trình trong tình trạng tắc nghẽn
    - ✓ Đình chỉ từng tiến trình liên quan cho đến khi không còn chu trình gây tắc nghẽn: để chọn được tiến trình thích hợp bị đình chỉ, phải dựa vào các yếu tố như độ ưu tiên, thời gian đã xử lý, số lượng tài nguyên đang chiếm giữ , số lượng tài nguyên yêu cầu...

Phương pháp xử lý deadlock → Hiệu chỉnh/Phục hồi

- Khi đã phát hiện được tắc nghẽn, có hai lựa chọn chính để hiệu chỉnh tắc nghẽn :
  - Thu hồi tài nguyên: thu hồi một số tài nguyên từ các tiến trình và cấp phát các tài nguyên này cho những tiến trình khác cho đến khi loại bỏ được chu trình tắc nghẽn. Cần giải quyết 3 vấn đề sau:
    - ✓ Chọn lựa một nạn nhân: tiến trình nào sẽ bị thu hồi tài nguyên? và thu hồi những tài nguyên nào ?
    - ✓ Trở lại trạng thái trước tắc nghẽn: khi thu hồi tài nguyên của một tiến trình, cần phải phục hồi trạng thái của tiến trình trở lại trạng thái gần nhất trước đó mà không xảy ra tắc nghẽn.
    - ✓ Tình trạng « đói tài nguyên »: làm sao bảo đảm rằng không có một tiến trình luôn luôn bị thu hồi tài nguyên ?

Tham khảo

- Slide Hệ điều hành – Lương Minh Huấn.
- http://dembinhyen.free.fr/UDS/Ebook/CD1/He%20Dieu%20Hanh/Htm/Bai052.htm
- https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/
- https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/