Deadlock

Môn học: Hệ điều hành

fit@hcmus
KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Nội dung

- Vấn đề deadlock
- Định nghĩa Deadlock
- Điều kiện phát sinh Deadlock
- Xử lý Deadlock

Vấn đề deadlock (1/2)

- Các tài nguyên của hệ thống (R_1, R_2, . . ., R_m) được chia sẻ cho các tiến trình có nhu cầu:
  - CPU cycles, memory space, I/O devices.
- Mỗi loại tài nguyên có thể có nhiều thể hiện (instances).
- Mỗi tiến trình sử dụng tài nguyên theo trình tự:
  - Request : yêu cầu tài nguyên, nếu yêu cầu không được thoã mãn nay, tiến trình phải đợi.
  - Use : sử dụng tài nguyên được cấp phát.
  - Release : giải phóng tài nguyên.
- Một tập các tiến trình bị chặn, mỗi tiến trình nắm giữ một tài nguyên và chờ đợi để truy xuất một tài nguyên khác được nắm giữ bởi một tiến trình trình khác trong tập đó. Ví dụ:
  - Hệ thống có 2 ổ đĩa.
  - P1 và P2 mỗi tiến trình giữ một ổ đĩa và cần một ổ đĩa khác.

Vấn đề deadlock (2/2)

- (a) Deadlock-free code.
- (b) Code with a potential deadlock.

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

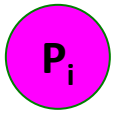
Định nghĩa Deadlock

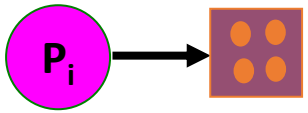
- Deadlock:
  - Chờ đợi một sự kiện không bao giờ xảy ra.
  - Các tiến trình trong tập hợp chờ đợi lẫn nhau.
- Starvation:
  - Chờ đợi không có giới hạn một sự kiện mà chưa thấy xảy ra.
- Deadlock kéo theo Starvation:
  - Điều ngược lại không chắc.

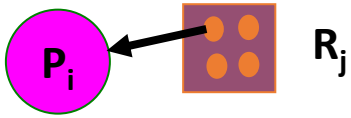
Các điều kiện xảy ra Deadlock

- Coffman, Elphick và Shoshani (1971) đã đưa ra 4 điều kiện cần có thể làm xuất hiện tắc nghẽn:
  - Mutual exclusion:
    - Hệ thống có sử dụng những loại tài nguyên mang bản chất không chia sẻ được.
  - Hold and wait:
    - 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.
  - No preemption:
    - Tài nguyên chỉ được thu hồi khi tiến trình đang chiếm giữ chúng tự nguyện trao trả.
  - Circular wait:
    - Tồn tại một chu kỳ trong đồ thị cấp phát tài nguyên .
- Hội đủ 4 điều kiện trên đây: Deadlock có thể xảy ra.

Mô hình hóa cấp phát tài nguyên bằng đồ thị

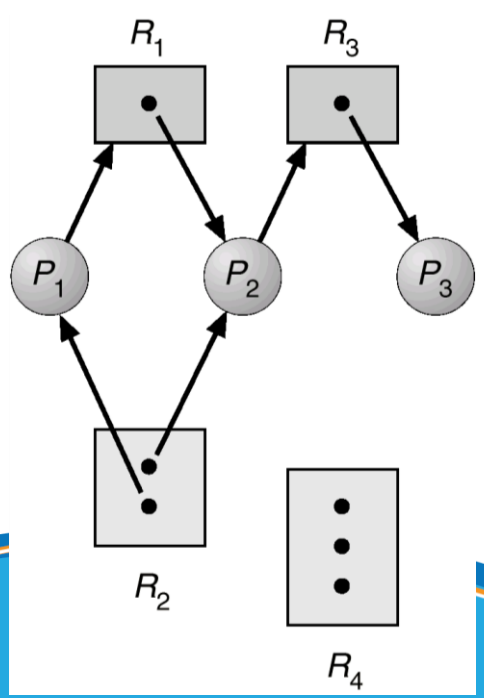
- 2 loại nodes:
  - P = {P1, P2, ..., Pn}, tập các tiến trình 

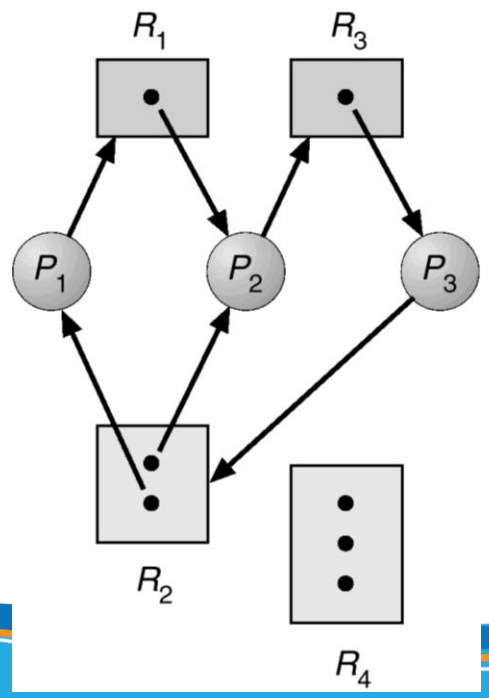
  - R = {R1, R2, ..., Rm}, tập các loại tài nguyên  R_j
- Tiến trình yêu cầu tài nguyên :
  - Pi → Rj
  - Pi yêu cầu 1 thể hiện của Rj 

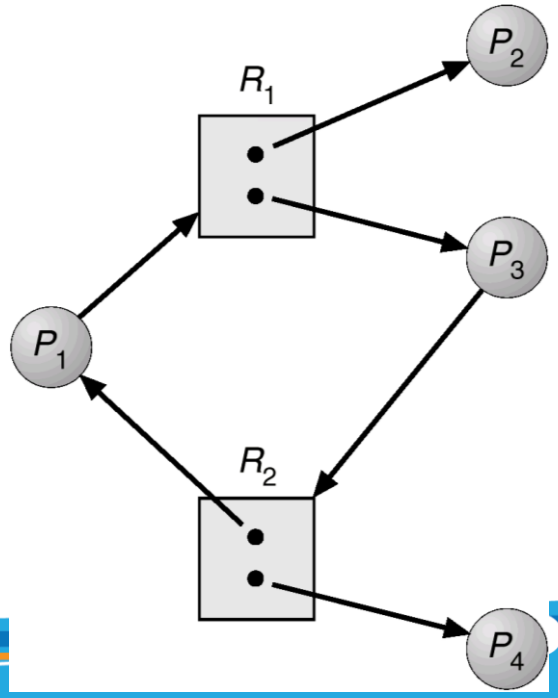
- Tài nguyên được cấp phát cho tiến trình :
  - Rj → Pi
  - Pi đang giữ 1 thể hiện của Rj 

Ví dụ đồ thị cấp phát tài nguyên

- Nếu đồ thị không có chu trình → không có deadlock.
- Nếu đồ thị có 1 chu trình ?
  - Nếu mỗi tài nguyên chỉ có 1 thể hiện ? → deadlock.
  - Nếu mỗi tài nguyên có nhiều thể hiện ? → có thể có deadlock.







Các phương pháp xử lý Deadlock

- Sử dụng một protocol để bảo đảm deadlock không bao giờ xảy ra trong hệ thống.
  - Deadlock prevention.
  - Deadlock avoidance.
- Cho phép hệ thống rơi vào tình trạng deadlock, sau đó tìm cách phát hiện và hiệu chỉnh.
  - Deadlock detection.
  - Deadlock recovery.
- Không quan tâm vấn đề deadlock !
  - Ostrich algorithm.

Deadlock Prevention – Mutual Exclusion

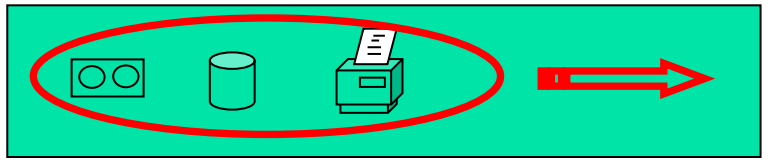
- Đảm bảo Deadlock không thể xảy ra.
  - Tìm cách loại bỏ ít nhất 1 trong 4 điều kiện cần để xảy ra Deadlock.
- Mutual Exclusion:
  - Không cần đảm bảo độc quyền truy xuất tài nguyên ?
    - Với các shareable resources: OK !
    - Với các non-shareable resource: Không thể ?
      - Giải pháp: virtualize, spooling, ... ?
      - Rất hạn chế, đặc biệt đối với tài nguyên phần mềm.

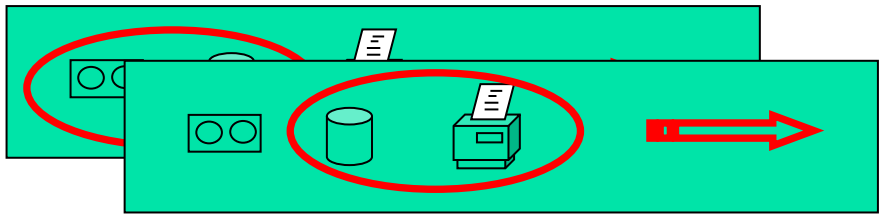
  → Kết luận : Không thể loại bỏ

Deadlock Prevention – Hold and Wait

- Hold and Wait:
  - Không cho vừa chiếm giữ, vừa yêu cầu thêm tài nguyên.
  - No Wait:
    - Cấp cho tiến trình tất cả các tài nguyên cần thiết trước khi bắt đầu xử lý.
      - Làm sao biết ?
  - No Hold:
    - Tiến trình không xin được tài nguyên mới: trả lại tài nguyên cũ, và chờ xin lại lần sau.
      - "Trả lại" tài nguyên ở trạng thái nào ?
  - Sử dụng tài nguyên kém hiệu quả, có thể starvation.

→ Kết luận : thật sự khó khả thi, không thể loại bỏ.





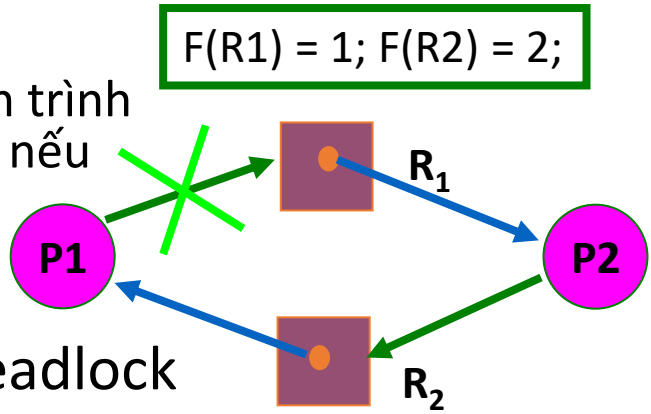
Deadlock Prevention – No Preemption

- No Preemption:
  - Hệ điều hành chủ động thu hồi các tài nguyên của tiến trình blocked.
    - Tài nguyên nào có thể thu hồi ?
      - CPU: OK.
      - Printer ?
  - Các tài nguyên bị thu hồi sẽ được bổ sung vào danh sách tài nguyên tiến trình cần xin lại. Tiến trình chỉ có thể tiếp tục xử lý khi xin lại đủ các tài nguyên này (cũ và mới).

→ Kết luận : thật sự khó loại bỏ hoàn toàn.

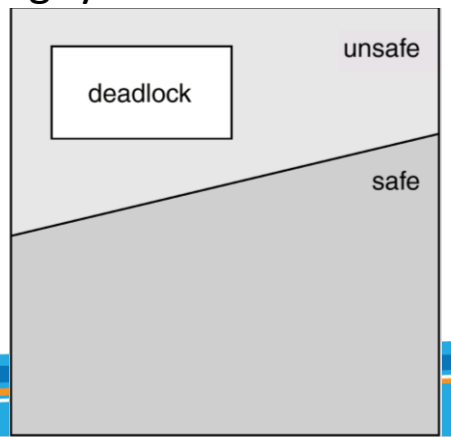
Deadlock Prevention – Circular Wait

- Circular Wait:
  - Không để xảy ra chu trình trong đồ thị cấp phát tài nguyên.
    - Cấp phát tài nguyên theo 1 trật tự nhất định.
      - Trật tự nào ?
  - Gọi R = {R_1, R_2, ..., R_m} là tập các loại tài nguyên.
    - Các loại tài nguyên được phân cấp theo độ ưu tiên F(R) thuộc [1-N].
    - Ví dụ : F(đĩa) = 2, F(máy in) = 12.
    - 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).
- Đảm bảo Deadlock không thể xảy ra:
  - Không thể loại bỏ ít nhất 1 trong 4 điều kiện cần để xảy ra Deadlock
  - Quá khắt khe, không khả thi ...



Deadlock Avoidance

- Một số định nghĩa cơ bản:
  - Trạng thái an toàn (Safe): hệ thống có thể thỏa mãn các nhu cầu tài nguyên (cho đến tối đa) của tất cả các tiến trình theo một thứ tự nào đó mà không dẫn đến tắc nghẽn.
    - Việc cấp phát tài nguyên không hình thành chu trình nào trong đồ thị cấp phát.
  - Một chuỗi cấp phát an toàn: một thứ tự kết thúc các tiến trình $<P_1, P_2, ..., P_n>$ sao cho với mỗi tiến trình Pi nhu cầu tài nguyên của Pi có thể được thỏa mãn với các tài nguyên còn tự do của hệ thống, cộng với các tài nguyên đang bị chiếm giữ bởi các tiến trình Pj khác, với $j<i$.
- Thay đổi chiến lược cấp phát tài nguyên để đảm bảo không dẫn dắt hệ thống vào tình trạng deadlock.
  - Chỉ thỏa mãn yêu cầu tài nguyên của tiến trình khi trạng thái kết quả là an toàn.
  - Đòi hỏi phải biết trước một số thông tin về nhu cầu sử dụng tài nguyên của tiến trình.
- Nhận xét:
  - Hệ thống ở safe state không deadlocks.
  - Nếu hệ thống ở unsafe state ? có khả năng deadlock.
  - Avoidance ? bảo đảm hệ thống không bao giờ đi vào trạng thái unsafe.



Deadlock Avoidance – Giải pháp

- Giả sử hệ thống được mô tả với các thông tin sau:
  - int Available[NumResources];
    - Available[r]= số lượng các thể hiện còn tự do của tài nguyên r.
  - int Max[NumProcs, NumResources];
    - Max[p,r]= nhu cầu tối đa của tiến trình p về tài nguyên r.
  - int Allocation[NumProcs, NumResources];
    - Allocation[p,r] = số lượng tài nguyên r thực sự cấp phát cho p.
  - int Need[NumProcs, NumResources];
    - Need[p,r] = Max[p,r] – Allocation[p,r]

Deadlock Avoidance – Giải thuật đơn giản

- Pi xin k thể hiện của Rj

1. if (k <= Need[i,j])
   goto 2;
   else
   Error();
2. if (k <= Available[j])
   Allocate(i,j,k); //cấp cho Pi k thể hiện Rj
   else
   MakeWait(Pi);

Deadlock Avoidance – Banker's Algorithm

- Pi xin k thể hiện của Rj
  1. if (k <= Need[i,j]) goto 2;
     else Error();
  2. if (k <= Available[j]) goto 3;
     else MakeWait(Pi);
  3. /*Giả sử hệ thống đã cấp phát cho Pi các tài nguyên mà nó yêu cầu và cập nhật tình trạng hệ thống như sau:*/

     Available[j] = Available[j] - k;

     Allocation[i,j]= Allocation[i,j]+ k;

     Need[i,j] = Need[i,j] - k;

     Result = TestSafe();

     //cấp cho Pi k thể hiện Rj
     if (Result == Safe) Allocate(i,j,k);
     else MakeWait(Pi);

- TestSafe():
  1. Giả sử có các mảng:
     int Work[NumResources] = Available[NumResources];
     int Finish[NumProcs] = false;
  2. Tìm i sao cho:
     Finish[i] == false & Need[i,j] <= Work[i,j], j <=NumRes
     Nếu không có i như thế, đến bước 4.
  3. Work = Work + Allocation[i];
     Finish[i] = true;
     Đến bước 2
  4. Nếu Finish[i] == true với mọi i, thì hệ thống ở trạng thái an toàn.

Ví dụ

- Giả sử tình trạng hệ thống được mô tả như sau:

	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			

Ví dụ (tt)

- Resources cần cho mỗi tiến trình được tính như sau:

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

3 – 1

Ví dụ (tt)

- P2 yêu cầu 4 cho R1, 1 cho R3 → Chạy Banker's Algorithm.

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

Ví dụ (tt)

- P2 yêu cầu 4 cho R1, 1 cho R3 → Chạy TestSafe().

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	9	3	6
P2	0	0	0	0	0	0			
P3	0	0	0	0	0	0			
P4	0	0	0	0	0	0			

P2 yêu cầu 4 cho R1, 1 cho R3 → chấp nhận

Deadlock Detection

- Chấp nhận hệ thống rơi vào trạng thái deadlock.
- Hệ thống nên cung cấp:
  - Một giải thuật kiểm tra và phát hiện deadlock có xảy ra trong hệ thống hay không.
  - Một giải thuật để hiệu chỉnh, phục hồi hệ thống về trạng thái trước khi deadlock xảy ra.
- Cần tốn kém chi phí để:
  - Lưu trữ, cập nhật các thông tin cần thiết.
  - Xử lý giải thuật phát hiện deadlock.
  - Chấp nhận khả năng mất mát khi phục hồi.

Deadlock Detection – Giải thuật

- int Available[NumResources];
  // Available[r]= số lượng các thể hiện còn tự do của tài nguyên r
- int Allocation[NumProcs, NumResources];
  // Allocation[p,r] = số lượng tài nguyên r thực sự cấp phát cho p
- int Request[NumProcs, NumResources];
  // Request[p,r] = số lượng tài nguyên r tiến trình p yêu cầu thêm

1. Giả sử có các mảng:
   int Work[NumResources] = Available;
   int Finish[NumProcs];
   for (i = 0; i < NumProcs; i++)
   Finish[i] = (Allocation[i] == 0);
2. Tìm i sao cho:
   Finish[i] == false & Request[i,j] <= Work[i,j], j <= NumRes
   Nếu không có i như thế, đến bước 4.
3. Work = Work + Allocation[i];
   Finish[i] = true;
   Đến bước 2.
4. Nếu Finish[i] == true với mọi i, thì hệ thống ở trạng thái không có deadlock.
   Ngược lại, các tiến trình Pi, Finish[i] == false sẽ ở trong tình trạng deadlock.

Deadlock Detection – Nhận xét

- Khi nào, và mức độ thường xuyên cần kích hoạt giải thuật phát hiện deadlock ? Phụ thuộc vào:
  - Tần suất xảy ra deadlock ?
  - Số lượng các tiến trình liên quan, cần "rolled back"?
    - 1 cho mỗi chu trình rời nhau.
- Một cách cẩn thận tối đa, kích hoạt giải thuật phát hiện mỗi khi có một yêu cầu cấp phát bị từ chối.

Chi phí cao

- Tiết kiệm chi phí: kích hoạt giải thuật phát hiện deadlock sau những chu kỳ định trước.
  - Khuyết điểm ?

Thiếu thông tin

Deadlock Recovery – Hủy bỏ tiến trình

- Hủy tất cả các tiến trình liên quan deadlock.
  - Thiệt hại đáng kể ...
- Hủy từng tiến trình liên quan cho đến khi giải toả được chu trình deadlock.
  - Tốn chi phí thực hiện giải thuật phát hiện deadlock.
  - Bắt đầu từ tiến trình nào ? Sau đó là ai ? ...
    - Priority of the process.
    - How long process has computed, and how much longer to completion.
    - Resources the process has used.
    - Resources process needs to complete.
    - How many processes will need to be terminated.
    - Is process interactive or batch?
  - Sao cho thiệt hại ít nhất.

Deadlock Recovery – Thu hồi tài nguyên

- Lần lượt thu hồi một số tài nguyên từ các tiến trình và cấp phát các tài nguyên này cho những tiến trình khác đến khi giải toả được chu trình deadlock.
- 3 vấn đề cần quan tâm:
  - Chọn lựa “nạn nhân”: – Thu hồi tài nguyên nào ? Của ai ?
    - Tiến trình nắm giữ nhiều tài nguyên.
    - Tiến trình có thời gian đã xử lý không cao.
    - ... ?
  - Rollback: quay lại trạng thái an toàn (nào?), khởi động lại tiến trình từ trạng thái an toàn đó.
    - Phải lưu vết hoạt động của hệ thống -> Chi phí cao.
  - Starvation: có thể một tiến trình nào đó luôn luôn bị chọn làm “nạn nhân”, không bao giờ có đủ tài nguyên để tiến triển xử lý.

Deadlock Ignorance – Ostrich's algorithm

- "Stick your head in the sand and pretend there is no problem."
- Có thể chấp nhận không ?
  - Cân nhắc giữa tần suất xảy ra deadlock và chi phí giải quyết deadlock.
    - Nếu deadlock xảy ra mỗi năm một lần, nhưng sự cố hệ thống do lỗi phần cứng và lỗi HĐH xảy ra mỗi tuần một lần, thì có đáng để giảm hiệu năng hệ thống để giải quyết deadlock ?
  - Thực tế, HĐH có nên block thực thi lời gọi hệ thống để mở một thiết bị vật lý như máy in không khi có một tác vụ in khác đang được thực hiện ?
    - Thông thường là tùy thuộc vào trình điều khiển thiết bị quyết định: hoặc là block, hoặc là trả về mã lỗi.
- Là giải pháp của hầu hết HĐH hiện nay.