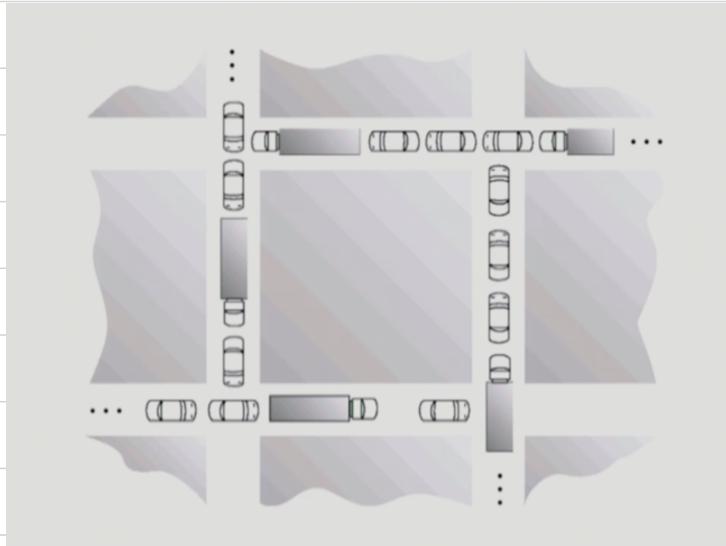


Deadlock (교착 상태)



The Deadlock Problem

- Deadlock

- 일련의 프로세스들이 서로가 가진 자원을 기다리며 block된 상태

- Resource

- 하드웨어, 소프트웨어 등을 포함하는 개념
- (예) I/O device, CPU cycle, memory space, Semaphore 등
- 프로세스가 자원을 사용하는 절차
 *Request, Allocate, Use, Release

- Deadlock Example 1

- 시스템에 2개의 tape drive가 있다.
- 프로세스 P₁과 P₂ 각각이 하나의 tape drive를 보유한 채 다른 하나를 기다리고 있다.

- Deadlock Example 2.

- Binary semaphores A and B

P ₁ :	P ₂ :
P(A);	P(B);
P(B);	P(A);

Dead lock 발생의 4가지 조건

• Mutual Exclusion (상호 배제)

V 매 순간 하나의 프로세스 만이 자원을 사용할 수 있음

• No preemption (비선점)

✓ 프로세스는 자원을 스스로 빼어놓을 뿐 강제로 빼앗기지 않음

• Hold and Wait (보유 대기)

▼ 자원을 가진 프로세스가 다른 자원을 기다릴 때 보유 자원을 놓지 않고 계속 가지고 있음.

• Circular wait (순환 대기)

✓ 자원을 기다리는 프로세스 간에 사이클이 형성되어야 한다.

∨ 프로세스 P_0, P_1, \dots, P_n 이 있을 때

P_0 은 P_1 이 가진 자율을 기다림

P_1 은 P_2 가 가진 자원을 기다림

P_{n+1} 은 P_n 이 가진 차원을 갖다짐

১০৮ ১০৯ শত শতের গুরু

Resource-Allocation Graph (자원 할당 그래프)

- Vertex

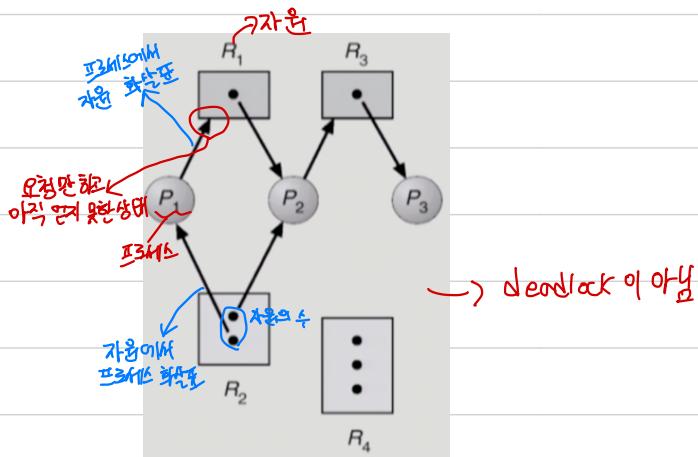
\vee Process $P = \{P_1, P_2, \dots, P_n\}$

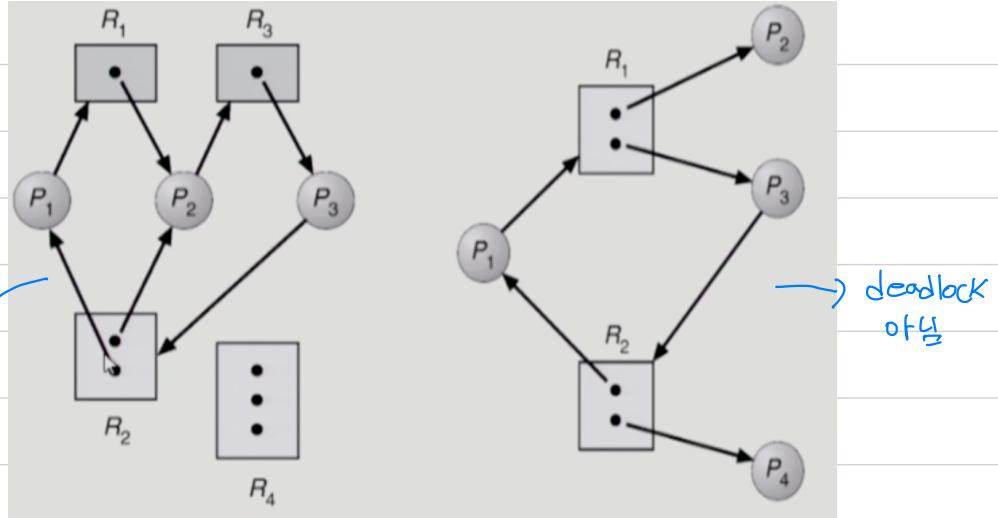
✓ Resource R = {R₁, R₂, ..., R_n}

- Edge

v requests edge $P_i \rightarrow R_j$

✓ assignment edge $R_i \rightarrow P_i$





- 그래프에 Cycle이 없으면 deadlock이 아니다.

- 그래프에 Cycle이 있으면

 v if only one instance per resource type, then deadlock

 v if several instances per resource type, possibility of deadlock

Deadlock의 처리 방법

- Deadlock Prevention

 v 자원 할당 시 Deadlock의 4가지 필요 조건 중 어느 하나가 만족되지 않도록 하는 것

- Deadlock Avoidance

 v 자원 요청에 대한 부가적인 정보를 이용해서 deadlock의 가능성이 없는 경우에만 자원을 할당

 v 시스템 state가 원래 state로 돌아올 수 있는 경우에만 자원 할당

- Deadlock Detection and recovery

 v Deadlock 발생은 허용하되 그에 대한 detection 무단을 두어 deadlock 발견시 recover

- Deadlock Ignorance

 v Dead lock을 시스템이 책임지지 않음

 v UNIX를 포함한 대부분의 OS가 선택

 ↳ 사람이 알아서 해결

Deadlock Prevention

- Mutual Exclusion

- 公共资源는 안되는 자원의 경우 반드시 성립해야 함.

- Hold and Wait

- 프로세스가 자원을 요청할 때 다른 어떤 자원도 가지고 있지 않아야 한다.
- 방법1. 프로세스 시작시 모든 필요한 자원을 할당 블록 하는 방법
- 방법2. 자원이 필요할 경우 보유 자원을 모두 놓고 다시 요청

- No Preemption

- 프로세스가 어떤 자원을 기다려야 하는 경우 이미 보유한 자원이 선점됨
- 모든 필요한 자원을 얻을 수 있을 때 그 프로세스는 다시 시작된다.
- Starvation을 쉽게避免하고饥饿을 할 수 있는 자원에서 주로 사용 (CPU, memory)

- Circular Wait

- 모든 자원 유형에 할당 순서를 정하여 정해진 순서대로만 자원 할당
- 예를 들어 순서가 3인 자원 R_3 를 보유 중인 프로세스가 순서가 1인 자원 R_1 을 할당받기 위해서는 우선 R_3 을 Release해야 한다.

⇒ Utilization 저하, throughput 감소, starvation 문제

일어나지 않은 deadlock을
마지 예방하기 때문에 상당히 비효율적

Deadlock Avoidance

- 자원 요청에 대한 부가정보를 이용해서 자원 할당이 deadlock으로 봉쇄 안전(safe) 한지를 동적으로 조사해서 안전한 경우에만 할당
- 가장 단순하고 일반적인 모델은 프로세스들이 필요로 하는 각 자원별 최대 사용량을 미리 선언하도록 하는 방법론.

• Safe state

- v 시스템 내의 프로세스들에 대한 **Safe sequence**가 존재하는 상태

• Safe Sequence

- v 프로세스의 Sequence $\langle P_1, P_2, P_3, \dots, P_n \rangle$ 이 Safe 상태면 P_i ($1 \leq i \leq n$) 의 자원 요청이 "가용자원 + 모든 P_j ($j < i$)의 모유자원"에 의해 충족되어야 함
 - 조건을 만족한다면 다음 방법으로 모든 프로세스의 수행을 보장
 - P_i 의 자원 요청이 즉시 충족될 수 없으면 모든 P_j ($j < i$)가 풀려갈 때까지 기다린다.
 - P_{n+1} 이 풀려오면 P_1 의 자원을 양여시켜 수행한다.

• 시스템이 Safe state에 있으면

⇒ NO deadlock

• 시스템이 Unsafe state에 있으면

⇒ Possibility of deadlock

• Dead lock Avoidance

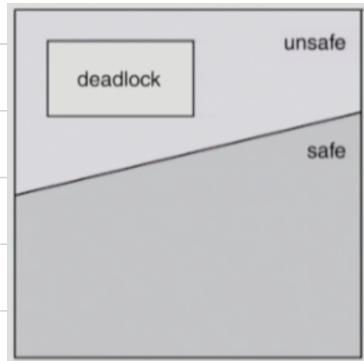
- v 시스템이 unsafe state에 들어가지 않는 것을 보장

- v 2가지 경우의 avoidance 알고리즘

- Single instance per resource types
 - Resource Allocation Graph Algorithm 사용

- Multiple instances per resource types

- Banker's Algorithm 사용



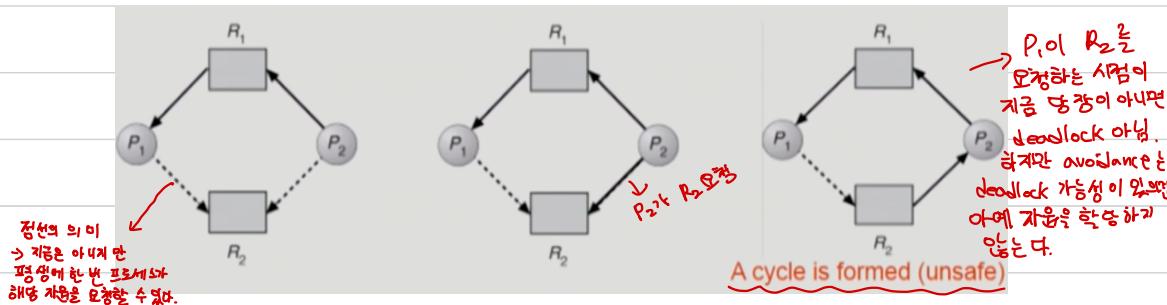
Resource Allocation Graph Algorithm

• Claim edge $P_i \rightarrow R_j$

- ✓ 프로세스 P_i 가 자원 R_j 를 미래에 요청할 수 있음을 표시(점선으로 표시)
- ✓ 프로세스가 해당 자원 요청시 request edge로 바뀜(실선)
- ✓ R_j 가 Release하면 assignment edge는 다시 claim edge로 바뀐다.

• Request edge의 assignment edge 변경시 (점선을 포함하여) Cycle이 생기지 않는 경우에만 요청 자원을 할당 한다.

• Cycle 생성 여부 조사시 프로세스의 수가 기일때 $O(n^3)$ 시간이 걸린다.



Banker's Algorithm

• 가정

- ✓ 모든 프로세스는 자원의 최대 사용량을 여러 명시
- ✓ 프로세스가 요청 자원을 모두 할당받은 경우 무한 시간 안에 이를 자원을 다시 반납한다.

• 방법

- ✓ 기본 개념: 자원 요청시 safe 상태를 유지할 경우에만 할당
- ✓ 총 요청 자원의 수가 가능한 자원의 수 보다 적은 프로세스를 선택 (그런 프로세스가 없다면 unsafe 상태)
- ✓ 그런 프로세스가 있으면 그 프로세스에게 자원을 할당
- ✓ 할당 받은 프로세스가 품출되면 모든 자원을 반납
- ✓ 모든 프로세스가 종료 될 때까지 이러한 과정 반복

Example of Banker's Algorithm

→ 5 processes

P₀ P₁ P₂ P₃ P₄

→ 3 resource types

A (10), B (5), and C (7) instances.

10 5 7

→ Snapshot at time T₀

	Allocation			Max			Available			Need (Max - Allocation)		
	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
P ₁	2	0	0	3	2	2	1	2	2	1	2	2
P ₂	3	0	2	9	0	2	6	0	0	6	0	0
P ₃	2	1	1	2	2	2	0	1	1	0	1	1
P ₄	0	0	2	4	3	3	4	3	1	4	3	1

* sequence < P₁, P₃, P₄, P₂, P₀>가 존재하므로 시스템은 safe state

↳ deadlock은 생기지 않지만 상당히 비효율적, 자원을 죄로 되는 상황인데도 혹시 모를 상황에 대비해 주지 않기 때문에.

↳ banker's algorithm은 위의 sequence로는 구하는게 아니라 각각의 프로세스 요청에 대해 가능, 불가능을 그때 그때 판별 하는 것.

Deadlock Detection and Recovery

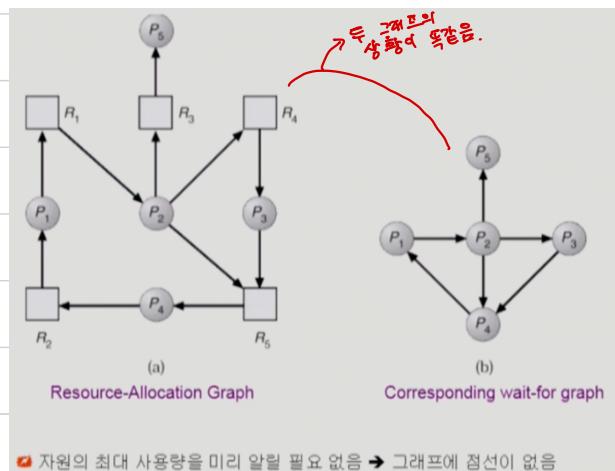
• Deadlock Detection

- Resource type당 single instance인 경우
⇒ 자원형당 그래프에서의 cycle이 둘 deadlock을 빚어
- Resource type당 multiple instance인 경우
⇒ Banker's algorithm과 유사한 방법 활용

• Wait-for graph 알고리즘

- Resource type당 single instance인 경우
- Wait-for graph
 - 자원형당 그래프의 변환
 - 프로세스 번호로 node 구성
 - P_j가 가지고 있는 자원을 P_i가 가지려는 경우 P_j → P_i

- Algorithm
 - Wait-for graph에 사용률이 존재하는지를 무기적으로 조사
 - O(n²)
 - Wait-for graph의 모든 노드를 조사하는 경우
 - 개별 프로세스에 차운을 최대 경우가 n²
 - 모든 경우를 확인한다면 k(n)=n²-n+1



▣ 자원의 최대 사용량을 미리 알릴 필요 없음 → 그래프에 점선이 없음

❖ Resource type 당 multiple instance인 경우

- ✓ 5 processes: $P_0 P_1 P_2 P_3 P_4$
- ✓ 3 resource types: A (7), B (2), and C (6) instances
- ✓ Snapshot at time 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			

deadlock을 확인하기 위해선
비단적 이 아님고 낙관적으로
접근!! ex) 지금 Available이
0,0,0이지만 P_0이 반납할 것 이므로
0,1,0이 될 것이다..

- ✓ No deadlock: sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will work!

☞ "Request"는 추가 요청 가능량이 아니라 현재 실제로 요청한 자원량을 나타냄

∴ 위 예시와 banker's algorithm 와는 그래프 보다 표를 통해 확인하는데 간편하여

→ 만약 P_2 가 C를 1개 요청한다면?

⇒ 자원 요청을 하지 않는 프로세스는 P_0 뿐이며 P_0 가 가진 자원을 반납한다고 가정하면 $1\ 1\ 0$ 이지만
이 걸로 어떤 프로세스의 요청도 만족 시킬 수 없기 때문에 deadlock occurs.

• Recovery

• Process termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated

deadlock에 연루된
프로세스 한 번에 종료.

deadlock에 연루된
프로세스 한 개씩 종료됨.

• Resource preemption

- 비용을 최소화 할 victim의 요청
- Safe state로 rollback하여 process를 restart
- Starvation 문제
 - 동일한 프로세스가 계속 victim으로 선점되는 경우
 - Cost factors로 rollback 횟수도 같이 고려

죽인 프로세스
선택

☞ 따라서 자원을 뺏은 피란을
조금씩 다르게 해야 함

Deadlock Ignorance

- Deadlock이 일어나지 않는다고 생각하고 아무런 조치도 취하지 않음

▽ Deadlock이 매우 드물게 발생 함으로 Deadlock이 다른 조치 자체가 더 큰 overhead 될 수 있음.

▽ 만약, 시스템이 deadlock이 발생한 경우 시스템이 비정상적으로 작동하는 것을 사람이 느낀 후 직접 process를 종이는 등의 방법으로 대처

▽ UNIX, Windows 등 대부분의 OS가 저작