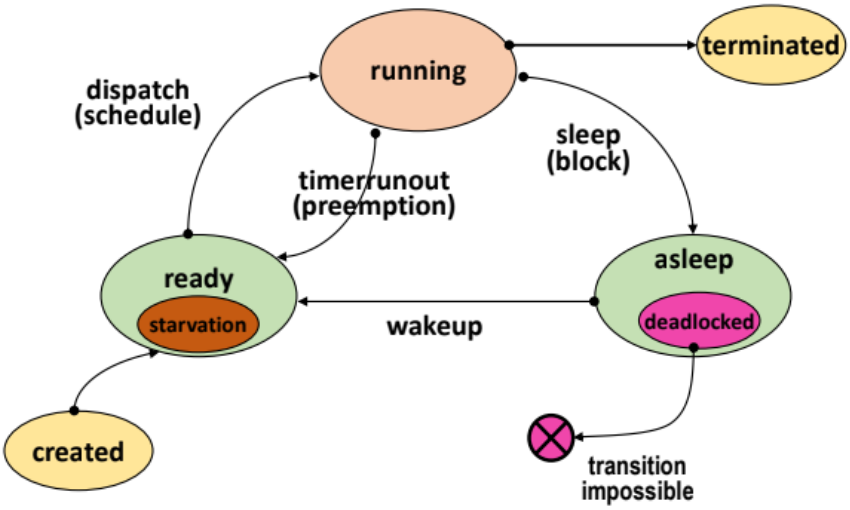
4주차 Deadlock

Deadlock 개념

- Blocked/Asleep state
  - 프로세스가 특정 이벤트 혹은 필요한 자원을 기다리는 상태
- Deadlock state
  - 프로세스가 발생 가능성이 없는 이벤트를 기다리는 경우 ⇒ 프로세스가 deadlock 상태에 있음
  - 시스템 내에 deadlock에 빠진 프로세스가 있는 경우 ⇒ 시스템이 deadlock 상태에 있음
- Deadlock vs Starvation



  - deadlock ⇒ asleep 상태에 존재, 일어날 가능성이 없는 이벤트나 자원을 기다림
  - starvation ⇒ ready 상태에 존재, 프로세스를 기다림, 발생 가능성이 있으나 운이 없어서 계속 기다리는 상태)

자원의 분류

- 일반적 분류

    - Hardware resources vs Software resources
- 선점 가능 여부에 따른 분류
    - Preemptible resources
        - 선점 당한 후, 돌아와도 문제가 발생하지 않는 자원
          ex) Processor(context switching), memory(swap-device) 등
    - **Non-preemptible resources**
        - 선점 당하면, 이후 진행에 문제가 발생하는 자원 ⇒ Rollback, restart등 특별한 동작이 필요
          ex) disk drive 등
- 할당 단위에 따른 분류
    - Total allocation resources
        - 자원 전체를 프로세스에게 할당
          ex) Processor, disk drive 등
    - Partitioned allocation resources
        - 하나의 자원을 여러 조각으로 나누어, 여러 프로세스들에게 할당
          ex) Memory 등
- 동시 사용 가능 여부에 따른 분류
    - **Exclusive allocation resources**
        - 한 순간에 한 프로세스만 사용 가능한 자원
          ex) Processor, memory (쪼개서 할당했을 때 할당된 영역은 혼자만 쓸 수 있다.), disk drive 등
    - Shared allocation resource
        - 여러 프로세스가 동시에 사용 가능한 자원
          ex) Program(sw), shared data 등
- 재사용 가능 여부에 따른 분류
    - **SR (Serially-reusable Resources)**
        - 시스템 내에 항상 존재 하는 자원, 사용이 끝나면 다른 프로세스가 사용 가능
          ex) Processor, memory, disk drive, program 등

- CR (Consumable Resources)
  - 한 프로세스가 사용한 후에 사라지는 자원
    ex) signal, message 등

Deadlock과 자원의 종류

- Deadlock을 발생시킬 수 있는 자원의 형태
  - Non-preemptible resources
  - Exclusive allocation resources
  - Serially reusable resources
  - 할당 단위는 영향을 미치지 않음
- CR을 대상으로 하는 Deadlock model
  - deadlock이 발생할 수 있으나 너무 복잡함

Deadlock 발생의 예

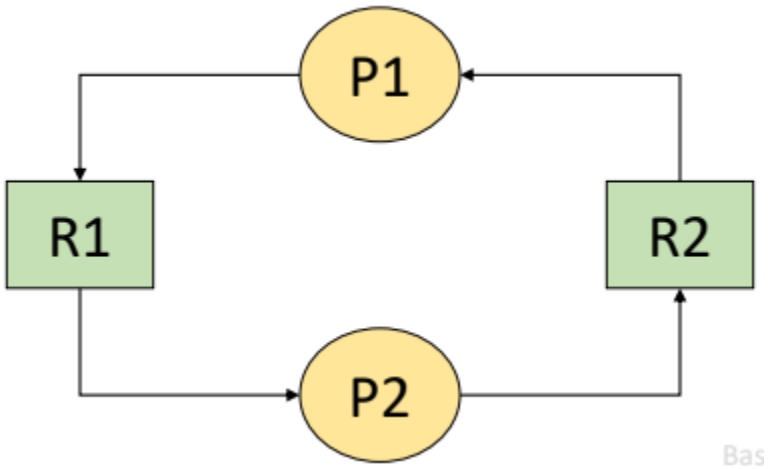
- 2개의 프로세스 (P1, P2)
- 2개의 자원 (R1, R2)

프로세스 P1	시간	프로세스 P2
...	t1	...
request R2 ← ①	t2	...
...	t3	request R1 ← ②
request R1 ← ③	t4	...
...	t5	request R2 ← ④
...	t6	...
release R1 ←	t7	...
...	t8	release R1 ←
release R2 ←	t9	...
...	t10	release R2 ←
...	t11	...

1. p1이 r2 요청
2. p2가 r1요청
3. p1이 r1 요청 ⇒ 아직 deadlock 아님
4. p2가 r2 요청 ⇒ 발생 가능성 없음, deadlock 발생

Deadlock Model(표현법)

- Graph Model
  - Node
    - 프로세스 노드(P1, P2), 자원 노드(R1, R2)
  - Edge
    - Rj → Pi : 자원 Rj 이 프로세스 Pi 에 할당 됨
    - Pi → Rj : 프로세스 Pi 가 자원 Rj을 요청 (대기 중)



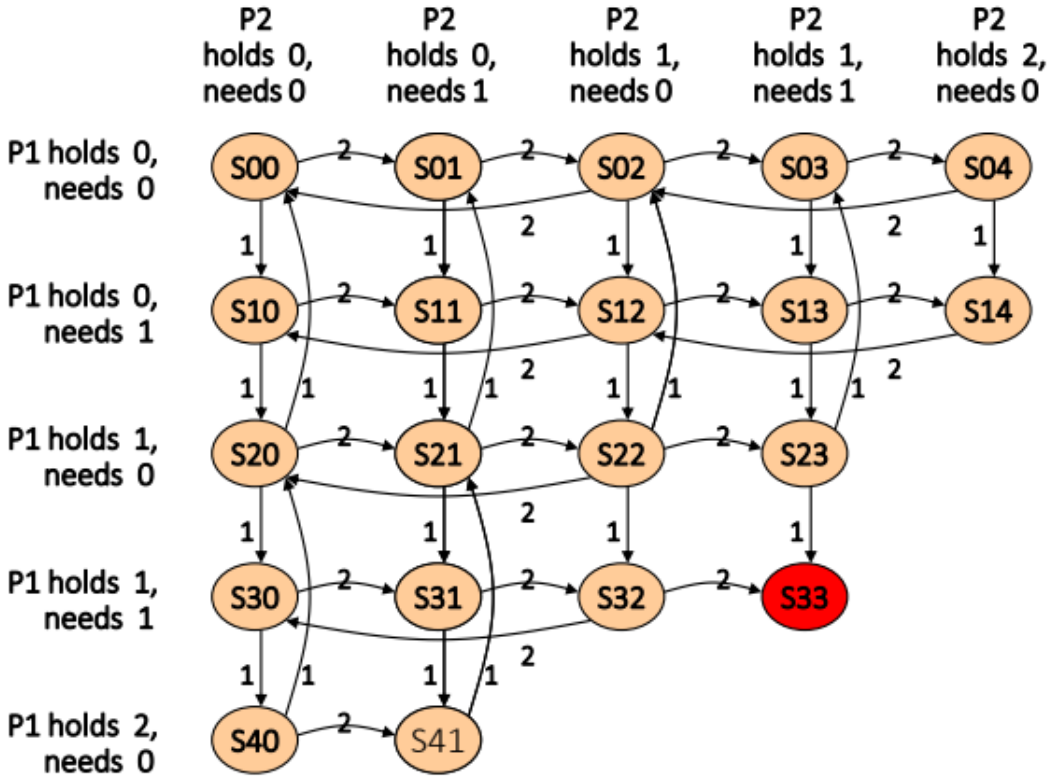
cycle 생성됨

- State Transition Model
  - 예제
    - 2개의 프로세스와 A type의 자원 2개(unit) 존재
    - 프로세스는 한번에 자원 하나만 요청/반납 가능
  - State

- 프로세스 1개

state	# of R units allocated	Request
0	0	x
1	0	o
2	1	x
3	1	o
4	2	x
	현재 자원 수	요청

- 프로세스 2개



Deadlock 발생 필요 조건

- 자원의 특성
  - Exclusive use of resources
  - Non-preemptible resources

- 프로세스의 특성
  - Hold and wait (Partial allocation)
    - 자원을 하나 hold하고 다른 자원 요청
  - Circular wait

Deadlock 해결 방법

- Deadlock prevention methods (교착상태 예방)
- Deadlock avoidance method (교착상태 회피)
- Deadlock detection and deadlock recovery methods (교착상태 탐지 및 복구)

Deadlock Prevention

- 4개의 deadlock 발생 필요 조건 중 하나를 제거 ⇒ Deadlock이 절대 발생하지 않음
  - Exclusive use of resources, Non-preemptible resources, Hold and wait (Partial allocation), Circular wait
- 모든 자원을 공유 허용 (Exclusive use of resources 조건 제거)
  - 현실적으로 불가능
- 모든 자원에 대해 선점 허용 (Non-preemptible resources 조건 제거)
  - 현실적으로 불가능
  - 유사한 방법
    - 프로세스가 할당 받을 수 없는 자원을 요청한 경우, 기존에 가지고 있던 자원을 모두 반납하고 작업 취소, 이후 처음 (또는 check-point)부터 다시 시작
    - 심각한 자원 낭비 발생 ⇒ 비현실적
- 필요 자원 한번에 모두 할당 (Hold and wait 조건 제거, Total allocation)
  - 자원 낭비 발생, 필요하지 않은 순간에도 가지고 있음
  - 무한 대기 현상 발생 가능
- Circular wait 조건 제거

    - Totally allocation을 일반화 한 방법
    - 자원들에게 순서를 부여
    - 프로세스는 순서의 증가 방향으로만 자원 요청 가능
    - 자원 낭비 발생

Deadlock Prevention 요약

- 4개의 deadlock 발생 필요 조건 중 하나를 제거
- Deadlock이 절대 발생하지 않음
- 심각한 자원 낭비 발생
    - Low device utilization
    - Reduced system throughput
- 비현실적

Deadlock Avoidance

- 시스템의 상태를 계속 감시
- 시스템이 deadlock 상태가 될 가능성이 있는 자원 할당 요청 보류
- 시스템을 항상 safe state로 유지
- Safe state
    - 모든 프로세스가 정상적 종료 가능한 상태 ⇒ **Safe sequence**가 존재
        - Deadlock상태가 되지 않을 수 있음을 보장
- Unsafe state
    - Deadlock 상태가 될 가능성이 있음, 반드시 발생한다는 의미는 아님
- 가정
    - 프로세스의 수가 고정됨
    - 자원의 종류와 수가 고정됨
    - 프로세스가 요구하는 자원 및 최대 수량을 알고 있음

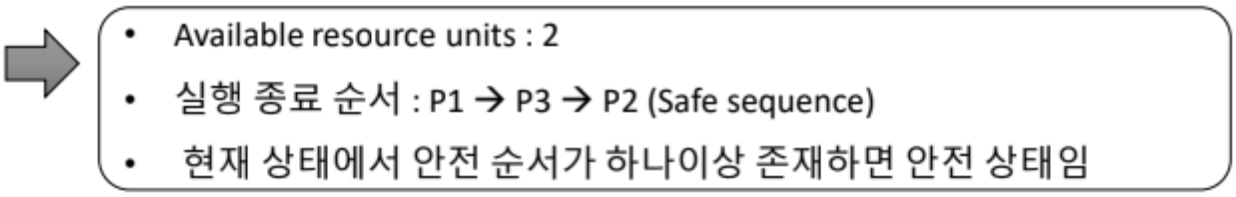
    - 프로세스는 자원을 사용 후 반드시 반납한다
    - 비현실적
- Dijkstra’s algorithm
    - Banker’s algorithm
- Habermann’s algorithm

Dijkstra’s banker’s algorithm

- Deadlock avoidance를 위한 간단한 이론적 기법
- 가정 - 한 종류(resource type) 의 자원이 여러 개(unit)
- 시스템을 항상 safe sate로 유지
- 1 resource type R, 10 resource units, 3 processes

State - 1

Process-ID	Max. Claim	Cur. Alloc.	Additional Need
P1	3	1	2
P2	9	5	4
P3	5	2	3
	최대 필요	현재 할당	추가적으로 필요



State - 2

Process-ID	Max. Claim	Cur. Alloc.	Additional Need
P1	5	1	4
P2	9	5	4
P3	7	2	5

- Available resource units : 3
- No safe sequence
- 임의의 순간에 세 프로세스들이 모두 세 개 이상의 단위 자원을 요청하는 경우 시스템은 교착상태 놓이게 됨

- 빌려줬을 때 safe sequence가 존재하면 빌려줌

State - 1

Process-ID	Max. Claim	Cur. Alloc.	Additional Need
P1	3	1	2
P2	9	5	4
P3	5	2	3

- When the process P2 requests a resource unit in State-1,
  - No safe sequence, after allocation
- Reject the request, because State-1-2 is unsafe

State – 1-2

Process-ID	Max. Claim	Cur. Alloc.	Additional Need
P1	3	1	2
P2	9	6	3
P3	5	2	3

- 빌려줬을 때 safe sequence가 존재하지 않으면 거절

State - 1

Process-ID	Max. Claim	Cur. Alloc.	Additional Need
P1	3	1	2
P2	9	5	4
P3	5	2	3

- When the process P2 requests a resource unit in State-1,
  - No safe sequence, after allocation
- Reject the request, because State-1-2 is unsafe

State – 1-2

Process-ID	Max. Claim	Cur. Alloc.	Additional Need
P1	3	1	2
P2	9	6	3
P3	5	2	3

Habermann's algorithm

- Dijkstra's algorithm의 확장
- 여러 종류의 자원 고려
  - Multiple resource types
  - Multiple resource units for each resource type
- 시스템을 항상 safe sate로 유지
- 3 types of resources: Ra, Rb, Rc
- Number of resource units for each type: (10, 5, 7)
- 5 processes

Max. claim of each process

Process-ID	Max. claim		
	Ra	Rb	Rc
P1	7	5	3
P2	3	2	2
P3	9	0	2
P4	2	2	2
P5	4	3	3

State-2

Process-ID	Max. Claim			Cur. Alloc.			Additional Need		
	Ra	Rb	Rc	Ra	Rb	Rc	Ra	Rb	Rc
P1	7	5	3	0	1	0	7	4	3
P2	3	2	2	2	0	0	1	2	2
P3	9	0	2	3	0	2	6	0	0
P4	2	2	2	2	1	1	0	1	1
P5	4	3	3	0	0	2	4	3	1

- Available resource units : (3, 3, 2)
- Safe sequence: P2 → P4 → P1 → P3 → P5
  - **Safe state** because there exist a safe sequence

- Dijkstra's algorithm와 같은 방식
- 빌려줬을 때 safe sequence가 존재하면 빌려줌

- When the process P2 requests (1, 0, 2) in State-2,
  - There exists a safe sequence (P2 → P4 → P1 → P3 → P5)
- **Accept the request, because State-2-1 is safe**

State-2-1

Process-ID	Max. Claim			Cur. Alloc.			Additional Need		
	Ra	Rb	Rc	Ra	Rb	Rc	Ra	Rb	Rc
P1	7	5	3	0	1	0	7	4	3
P2	3	2	2	3	0	2	0	2	0
P3	9	0	2	3	0	2	6	0	0
P4	2	2	2	2	1	1	0	1	1
P5	4	3	3	0	0	2	4	3	1

- 빌려줬을 때 safe sequence가 존재하지 않으면 거절

- When the process P1 requests (0, 3, 0) in State-2,
  - No safe sequence, after allocation
- **Reject the request, because State-2-2 is unsafe state**

State-2-2

Process-ID	Max. Claim			Cur. Alloc.			Additional Need		
	Ra	Rb	Rc	Ra	Rb	Rc	Ra	Rb	Rc
P1	7	5	3	0	4	0	7	1	3
P2	3	2	2	2	0	0	1	2	2
P3	9	0	2	3	0	2	6	0	0
P4	2	2	2	2	1	1	0	1	1
P5	4	3	3	0	0	2	4	3	1

Deadlock Avoidance 요약

- Daedlock의 발생을 막을 수 있음
- High overhead

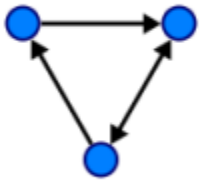
    - 항상 시스템을 감시하고 있어야 한다
- Low resource utilization
    - Safe state 유지를 위해, 사용되지 않는 자원이 존재
- Not practical
    - 프로세스 수, 자원 수가 고정
    - 필요한 최대 자원 수를 알고 있음

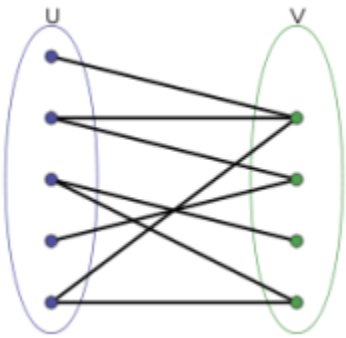
Deadlock Detection

- Deadlock 방지를 위한 사전 작업을 하지 않음
    - Deadlock이 발생 가능
- 주기적으로 deadlock 발생 확인
    - 시스템이 deadlock 상태인가?
    - 어떤 프로세스가 deadlock 상태인가?
- Resource Allocation Graph (RAG) 사용

Resource Allocation Graph (RAG)

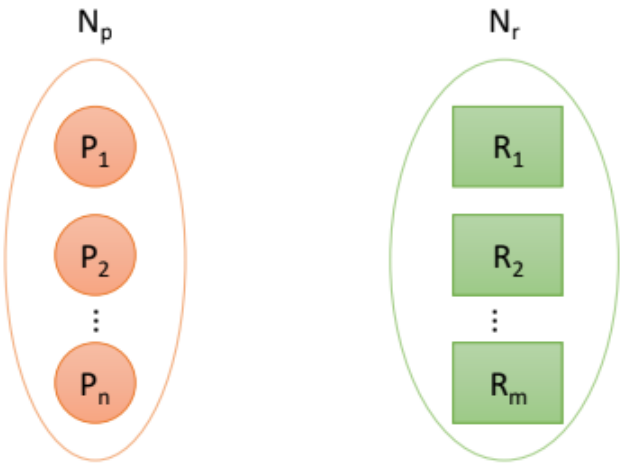
- Deadlock 검출을 위해 사용
- Directed, bipartite Graph



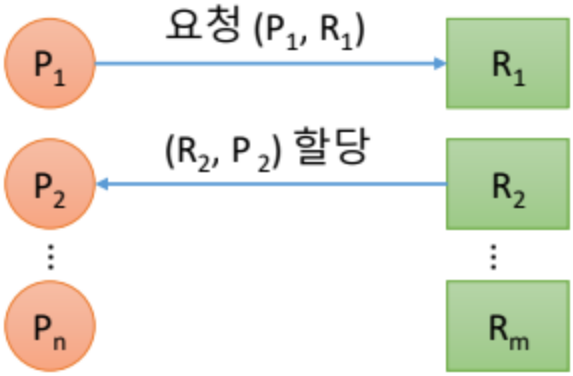


- 그래프 = 노드(N), 간선(E)
- N = 프로세스 노드와 리소스 노드의 집합

- N_p = 프로세스의 집합, N_r = 리소스의 집합,

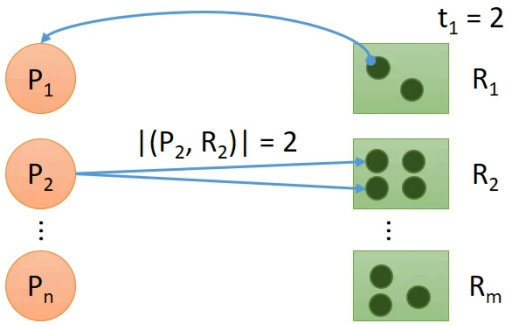


- Edge는 N_p와 N_r 사이에만 존재
  - e = (Pi, Rj) : 자원 요청
  - e = (Rj, Pi) : 자원 할당

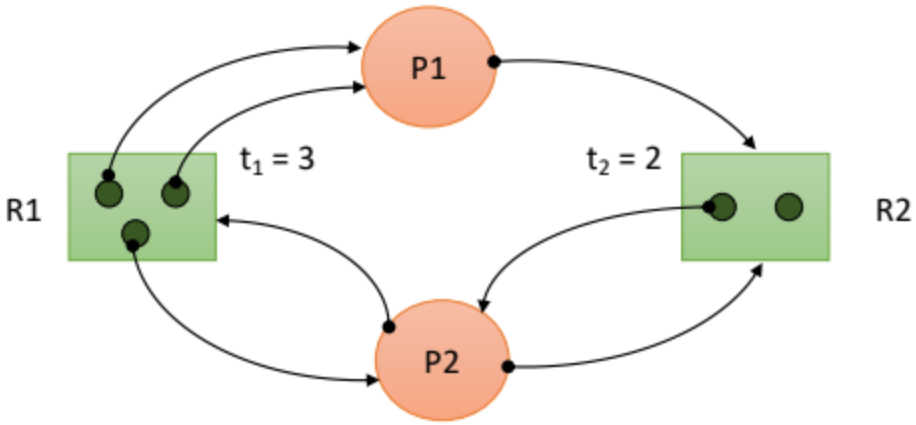


- R_k : k type의 자원
- t_k : R_k 의 단위 자원 수
  - For each $R_k \in N_r, \exists\ t_k$ which is the number of units of R_k

- **|(a,b)|** : (a,b) edge의 수



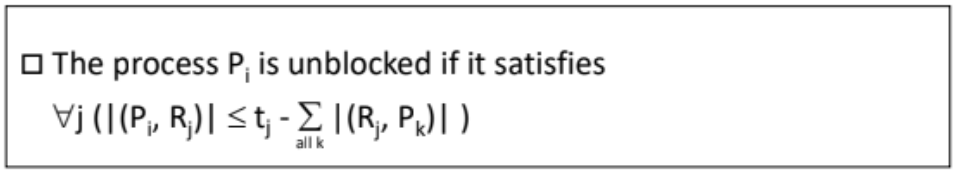
- RAG example



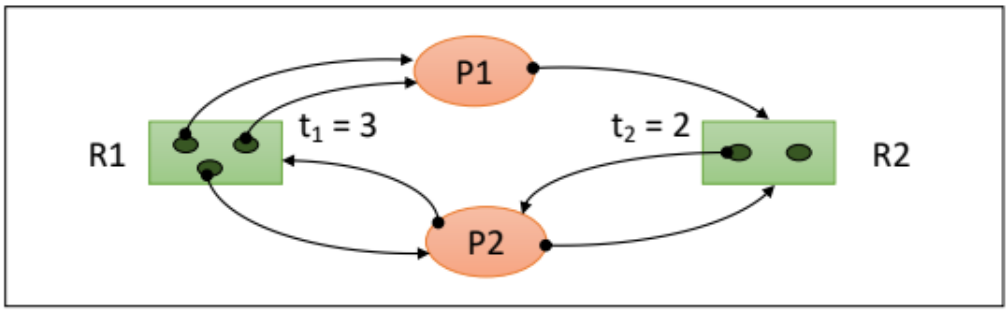
Graph reduction

- Deadlock인지 판단하는 방법
- 주어진 RAG에서 edge를 하나씩 지워가는 방법
- Completely reduced
  - 모든 edge가 제거 됨 ⇒ Deadlock에 빠진 프로세스가 없음
- Irreducible
  - 지울 수 없는 edge가 존재 ⇒ 하나 이상의 프로세스가 deadlock 상태

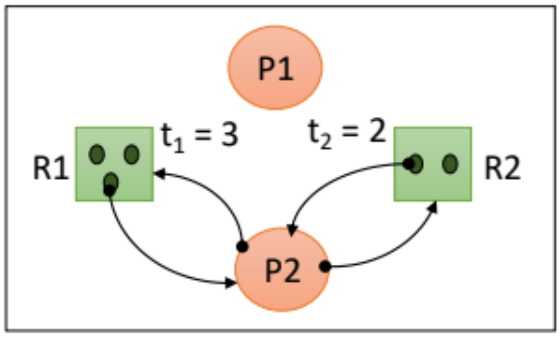
- Unblocked process에 연결된 edge 삭제
  - 필요한 자원을 모두 할당 받을 수 있는 프로세스



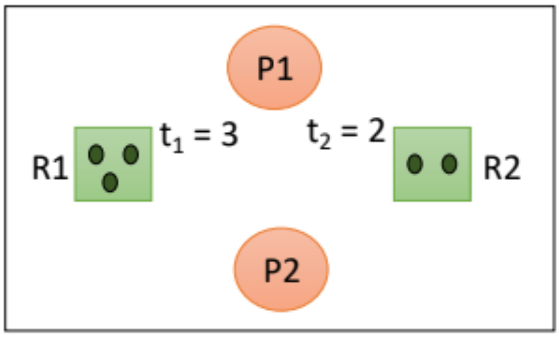
  - 모든 j에 대해 다음을 만족
  - P_i가 요청하는 R_j의 수 ≤ R_j의 전체 수 - 이미 사용 중인 R_j의 수
  - P_i가 요청하는 R_j의 수 ≤ 남은 R_j의 수
- 알고리즘
  1. Unblocked process에 연결 된 모든 edge를 제거
  2. 더 이상 unblocked process가 없을 때까지 1 반복
- 최종 Graph에서
  - 모든 edge가 제거 됨 (Completely reduced) ⇒ 현재 상태에서 deadlock이 없음
  - 일부 edge가 남음 (irreducible) ⇒ 자원을 할당 받을 수 없는 P 존재 ⇒ 현재 deadlock이 존재
- 예시 1 (deadlock 없음)



(a) Initial state

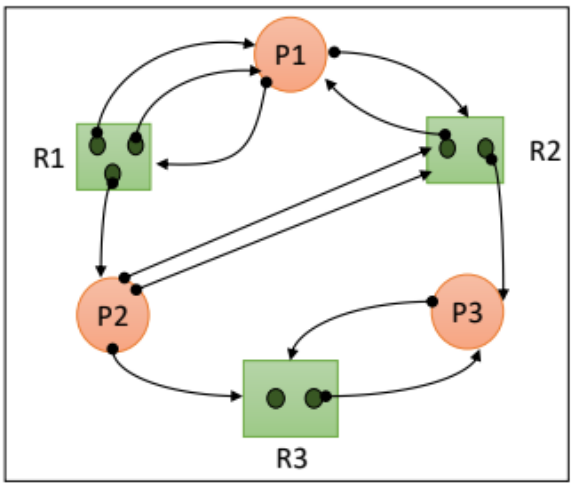


(b) Reduction by process P1

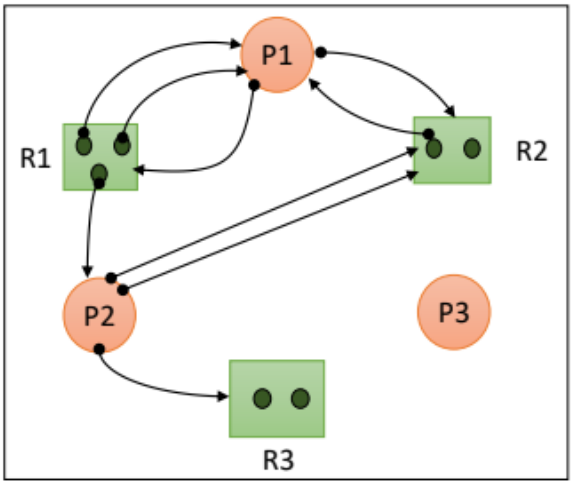


(c) Reduction by process P2

- 예시 2 (deadlock이 있음)



(a) Initial state



(b) Reduction by process P3

Graph Reduction 요약

- High overhead
  - 검사 주기에 영향을 받음

    - Node의 수가 많은 경우
- Low overhead deadlock detection methods (Special case)
    - Case-1) Single-unit resources
        - Cycle detection
    - Case-2) Single-unit request in expedient state
        - Knot detection

Deadlock Avoidance vs Detection

- Deadlock avoidance
    - 최악의 경우를 생각
    - 앞으로 일어날 일을 고려
    - Deadlock이 발생 하지 않음
- Deadlock detection
    - 최선의 경우를 생각
    - 현재 상태만 고려
    - Deadlock 발생 시 Recovery 과정이 필요

Deadlock Recovery

- Deadlock을 검출 한 후 해결 하는 과정
- Deadlock recovery methods
    - Process termination
    - Resource preemption

Process termination

- Deadlock 상태인 프로세스 중 일부 종료
- 강제 종료 된 프로세스는 이후 재시작 됨

- Termination cost model
  - 종료 시킬 deadlock 상태의 프로세스 선택
  - Termination cost
    - 우선순위, 종류, 총 수행 시간, 남은 수행 시간, 종료 비용 등등
- 종료 프로세스 선택
  - Lowest-termination cost process first
    - Simple
    - Low overhead
    - 불필요한 프로세스들이 종료 될 가능성이 높음
  - Minimum cost recovery
    - 최소 비용으로 deadlock 상태를 해소 할 수 있는 process 선택 ⇒ 모든 경우의 수를 고려 해야 함
    - Complex
    - High overhead (O(2^k))

Resource preemption

- Deadlock 상태 해결을 위해 **선점할 자원 선택**
- 해당 자원을 가지고 있는 프로세스를 종료 시킴
  - Deadlock 상태가 아닌 프로세스가 종료 될 수도 있음
  - 해당 프로세스는 이후 재시작 됨
- 선점할 자원 선택
  - Preemption cost model 이 필요
  - Minimum cost recovery method 사용 (O(r))

Checkpoint-restart method

- 프로세스의 수행 중 특정 지점(checkpoint) 마다 context를 저장
- Rollback을 위해 사용

- 프로세스 강제 종료 후, 가장 최근의 checkpoint에서 재시작