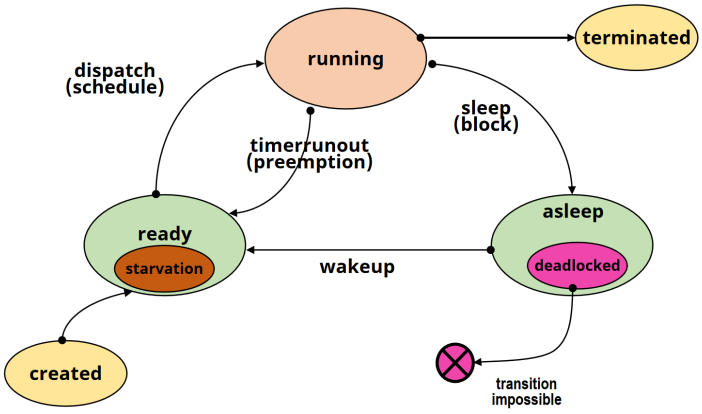
07. Deadlock

Deadlock 개요

Deadlock의 개념

- **Blocked/Asleep State**
  - 프로세스가 특정 이벤트를 기다리는 상태
  - 프로세스가 필요한 자원을 기다리는 상태
- **Deadlock State**
  - 프로세스가 Deadlock 상태에 있음: 프로세스가 발생 가능성이 없는 이벤트를 기다리는 경우
  - 시스템이 Deadlock 상태에 있음: 시스템 내 deadlock에 빠진 프로세스가 있는 경우

Deadlock vs Starvation



	Deadlock	Starvation
State	Blocked/Asleep State에 존재	Ready State에 존재
기다리는 자원	발생 가능성이 없는 이벤트	CPU 운이 없거나 우선 순위가 낮아서 계속 기다리게 됨 (발생 가능성 있음)

Deadlock 관점에서 자원의 분류

선점 가능 여부에 따른 분류

- **Preemptible Resources**
  - 선점 당한 후 돌아와도 작업 진행에 문제 없는 자원
  - ex) 프로세서(Context Switching), 메모리(Swap Device)
- **Non-preemptible Resources**
  - 선점 당한 후 돌아오면 작업 진행에 문제 발생하는 자원
    - Rollback, restart 등 특별한 동작이 필요
  - ex) 디스크 드라이브(disk drive, 파일 복사 중 끊겼다면 데이터 날라감)

할당 단위에 따른 분류

- **Total Allocation Resources**
  - 자원 전체를 프로세스에 할당
  - ex) 프로세서, 디스크 드라이브 등
- **Partitioned Allocation Resources**
  - 하나의 자원을 여러 조각으로 나눠 여러 프로세스에 할당
  - ex) 메모리

동시 사용 가능 여부에 따른 분류

- **Exclusive Allocation Resources**
  - 한 순간에 한 프로세스만 사용 가능한 자원
  - ex) 프로세서, 메모리(할당된 메모리 영역은 해당 프로세스만 사용 가능), 디스크 드라이브 등
- **Shared Allocation Resource**
  - 여러 프로세스가 동시 사용 가능한 자원
  - ex) SW 프로그램(source code, exe), shared data 등

재사용 가능 여부에 따른 분류

- **SR (Serially reusable Resources)**
  - 시스템 내 항상 존재하는 자원. 사용이 끝나면 다른 프로세스가 사용 가능
  - ex) 프로세서, 메모리, 디스크 드라이브, 프로그램 등

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

Deadlock을 발생시킬 수 있는 자원의 형태

- **Non-preemptible Resources**
  - 선점할 수 없는 자원에 대해 다른 프로세스가 요청한다면 deadlock 발생 가능
- **Exclusive Allocation Resources**
  - 한 프로세스만 사용 가능한 자원
- **Serially reusable Resources**
  - 시스템 내 항상 존재하는 자원

> 💡 **참고**
>
> - 할당 단위는 Deadlock 발생에 영향을 미치지 않음!
> - Consumable Resources 또한 Deadlock을 발생시킬 수 있지만, CR을 고려하면 Deadlock Model이 너무 복잡해져서 고려하지 않음

Example of Deadlock

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	...

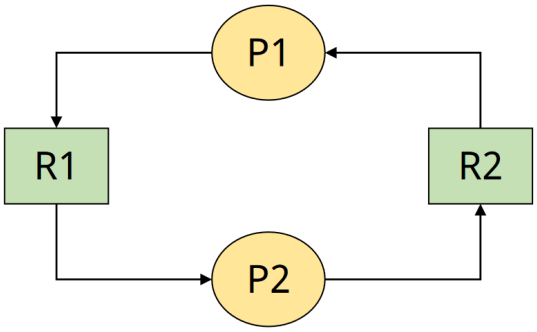
2개의 프로세스(P1, P2), 2개의 자원(R1, R2)

- 프로세스 `P1`과 `P2`가 각각 상대방이 사용하고 있는 자원 `R2`, `R1`에 대해 요청한 상태

- 발생 가능성 없는 이벤트

Deadlock Model (표현법)

Graph Model



`P2`가 `R2`를 요청하는 순간 Deadlock 발생
(Cycle 발생)

- **Node**
  - 프로세스 노드(`P1`, `P2`), 자원 노드(`R1`, `R2`)
- **Edge**
  - `R2` → `P1`: 자원 `R2`을 프로세스 `P1`에 할당
  - `P1` → `R1`: 프로세스 `P1`이 자원 `R1`을 요청 (대기 중)

State Transition Model

[예제]

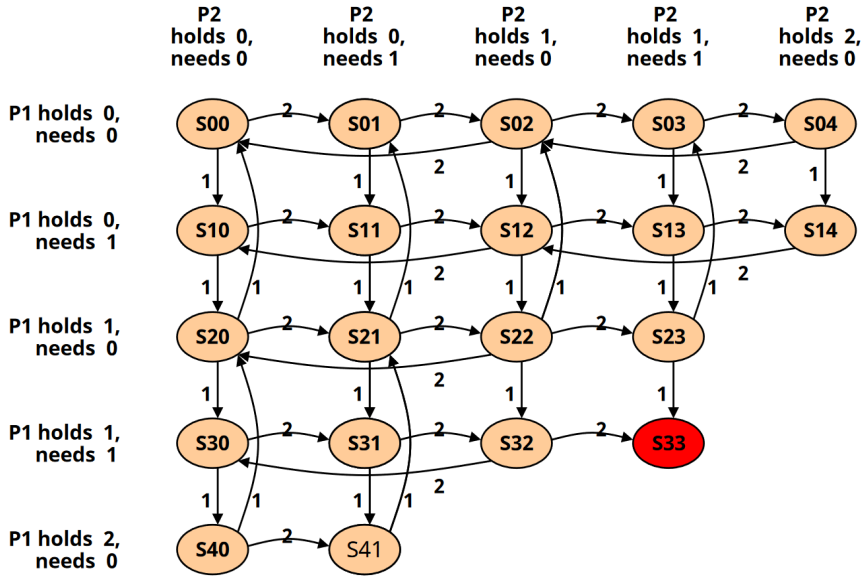
- 프로세스 2개, A 타입의 자원 2개 존재
- 프로세스는 한 번에 자원 하나만 요청/반납 가능

[가능한 State]

state	# of Res. units allocated	Request
0	0	X
1	0	O
2	1	X
3	1	O
4	2	X

state 인덱스/현재 자원 수/자원 요청 여부

[State Transition Model]



S[P1 상태][P2 상태]

- 각 프로세스에 의해 상태가 바뀌는 것을 상태 전이(state transition)으로 표현
- S33에서 Deadlock 발생: 각 프로세스가 자원을 갖고 있는 상태에서 요청할 때

Deadlock 발생 필요 조건 4가지

[자원의 특성]

1. Mutual Exclusion (상호 배제)
    - 한 번에 한 프로세스만 자원을 사용할 수 있다
2. No Preemption (비선점)
    - 프로세스가 어떤 자원의 사용을 끝낼 때까지 그 자원을 뺏을 수 없다

[프로세스의 특성]

3. Hold and Wait (점유 대기)
    - 프로세스가 할당된 자원을 가진 상태에서 다른 자원을 기다린다
4. Circular Wait (순환 대기)
    - 각 프로세스는 순환적으로 다음 프로세스가 요구하는 자원을 가지고 있다

Deadlock 해결 방법

Deadlock Prevention (교착상태 예방)

> Deadlock 발생 필요 조건 4가지 중 하나를 제거해 Deadlock이 발생하지 않도록 함
> ➡️ 하지만 심각한 자원 낭비 발생, 비현실적

1. **Mutual Exclusion (상호 배제) 조건 제거**
    - 해결책: 모든 자원 공유 허용 ➡️ 현실적으로 불가능
2. **No Preemption (비선점) 조건 제거**
    - 해결책: 모든 자원 선점 허용 ➡️ 현실적으로 불가능
    - 유사 해결책: 프로세스가 할당 받을 수 없는 자원을 요청한 겨우, 기존 갖고 있던 자원을 모두 반납하고 작업 취소, 처음(또는 check point)부터 다시 시작 ➡️ 심각한 자원 낭비 발생
3. **Hold and Wait (점유 대기) 조건 제거**
    - 해결책: 필요 자원을 한 번에 모두 할당(Total Allocation)
      ➡️ 필요하지 않은 순간에도 자원을 갖고 있어서 자원 낭비/무한 대기 현상 발생 가능
4. **Circular Wait (순환 대기) 조건 제거**
    - 해결책: Total Allocation 일반화. 자원에 순서를 부여, 프로세스가 순서 증가 방향으로만 자원 요청 가능하도록 해 사이클이 발생하지 않도록 함 ➡️ 자원 낭비 발생

Deadlock Avoidance (교착상태 회피)

> 시스템 상태를 계속 감시하면서 deadlock 상태가 될 가능성이 있는 자원에 대한 할당 요청 보류, 시스템이 항상 safe state로 유지될 수 있도록 함

Safe State vs Unsafe State

- **Safe State**
  - 시스템이 Deadlock이 발생하지 않고 모든 프로세스에 자원을 할당 가능한 상태
  - 시스템 내 프로세스들에 대한 Safe Sequence가 존재하는 상태
- **Safe Sequence**
  - 프로세스의 sequence <P1, P2, ..., Pn>에 대해, Pi(1≤i≤n)의 자원 요청이 “Pi에 할당된 자원(가용 자원) + 모든 Pj (j <i)의 보유 자원”에 의해 충족된다면 safe sequence임
  - Pi의 자원 요청이 즉시 충족될 수 없으면 모든 Pj(i<i)가 종료될 때까지 기다린 후, Pi의 자원 요청을 만족시켜 수행함
- **Unsafe State**
  - Safe Sequence가 존재하지 않는 상태
  - Deadlock 상태가 될 가능성이 있지만, 반드시 발생한다는 의미는 아님

[예시: Safe State Example]

- 가정: 동일한 단위 자원 R 10개, 프로세스 3개

State - 1

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

프로세스 ID/자원 최대 사용량/가용 자원 개수/필요 자원 개수

- 현재 시스템에서 할당 가능한 자원 2개
- Safe Sequence P1 → P3 → P2 존재 ⇒ Safe State

[예시: Unsafe State Example]

- 가정: 동일한 단위 자원 R 10개, 프로세스 3개

State - 2

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

- 현재 시스템에서 할당 가능한 자원 3개
- No safe sequence. 세 프로세스 모두 3개 이상의 단위 자원을 요청하는 경우 Deadlock 발생 가능
    - 필요 자원은 요구하는 자원이지, 없어도 작업 수행 완료할 수 있음 (Deadlock 발생 X)

Dijkstra's Algorithm (Banker's Algorithm)

[가정]

- Several Instances of a Resource Type (한 종류의 자원이 여러 개)
- 모든 프로세스는 자원의 최대 사용량을 미리 명시
- 프로세스가 요청 자원을 모두 할당 받은 경우 유한 시간 안에 자원을 다시 반납

[방법]

- 기본 개념: 자원 요청 시 safe state을 유지할 경우에만 자원 할당

1. 총 요청 자원 수가 가용 자원 수보다 적은 프로세스 선택 (그런 프로세스가 없다면 unsafe state)
    - unsafe state인 경우, 자원 요청 보류
2. safe state을 유지하는 프로세스가 있다면 그 프로세스에 자원 할당
3. 할당 받은 프로세스가 종료되면 모든 자원 반납
4. 모든 프로세스가 종료될 때까지 과정 반복

[예시: Safe State Example]

Haberman's Algorithm

- Dijkstra's Algorithm을 여러 종류의 자원을 고려할 수 있도록 확장
    - Multiple Instances of Multiple Resource Types
- 동일하게 시스템이 항상 safe state을 유지하도록 함

Deadlock Avoidance 장단점

[장점]

- Deadlock의 발생을 막을 수 있음

[단점]

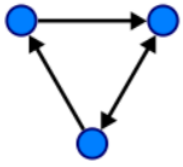
- High Overhead
  - 항상 시스템을 감시하고 있어어야 함
- Low Resource Utilization
  - Safe state 유지 위해 사용되지 않는 자원 존재
- Not Practical
  - 프로세스/자원 수가 고정되어 있고, 최대 자원 사용량을 미리 알고 있다는 가정은 비현실적

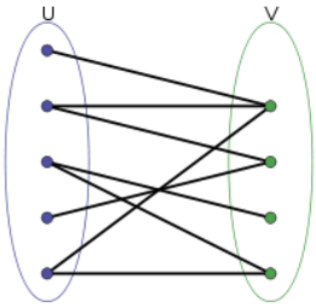
Deadlock Detection (교착 상태 탐지)

> Deadlock 방지를 위한 사전 작업을 하지 않고, 주기적으로 Deadlock이 발생했는지 확인 후 복구

Resource Allocation Graph (RAG)

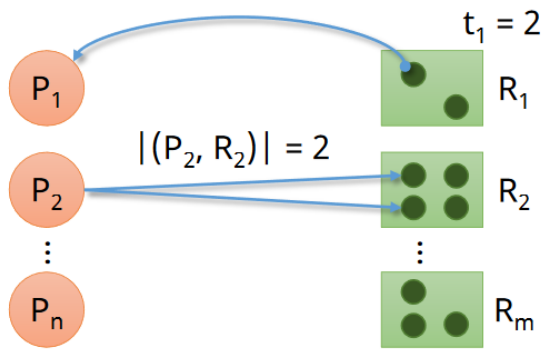
- Deadlock 검출을 위해 사용되는 그래프
- Directed, bipartite graph





[정의]

- Node $N = \{N_p, N_r\}$
  - 프로세스 집합 $N_p = \{P_1, P_2, \ldots, P_n\}$



- 자원 집합 $N_r = \{R_1, R_2, \ldots, R_n\}$
- Edge e: N_p와 N_r 사이에만 존재
  - $e = (P_i, R_j)$: 자원 요청
  - $e = (R_j, P_i)$: 자원 할당
- R_k: k 타입의 자원
- t_k: R_k의 단위 자원 수
- $|(a, b)|$: edge (a,b)의 수

Deadlock Detection Method: Graph Reduction

> 주어진 RAG에서 edge를 하나씩 지워나가면서 Deadlock을 탐지하는 방법

[방법]

1. Unblocked Process에 연결된 모든 edge 제거
   - Unblocked Process: 필요한 자원을 모두 할당 받을 수 있는 프로세스

     ▼ 수학적 정의

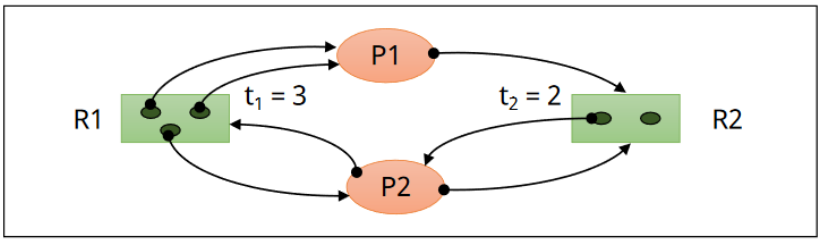
     Process P is unblocked if it satisfies $\forall j(|P_i, R_j)| \leq t_j - \sum_{allk} |(R_j, P_k)|)$

     P_i가 요청하는 모든 자원 수 ≤ 남은 자원 수
2. Unblocked Process가 존재하지 않을 때까지 1️⃣ 반복

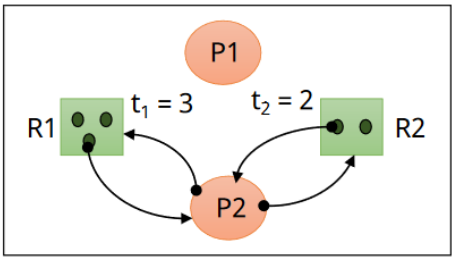
[Deadlock 판단 기준]

- **Completely Reduced**: 모든 edge가 제거됨
  - Deadlock에 빠진 프로세스가 없음
- **Irreducible**: 지울 수 없는 edge가 존재 (자원을 할당 받을 수 없는 프로세스 존재)
  - 하나 이상의 프로세스가 deadlock 상태

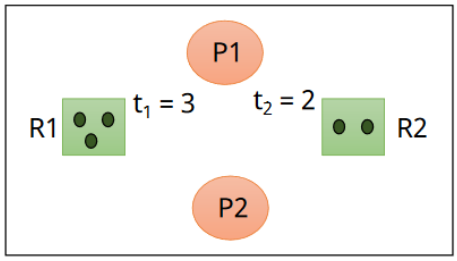
[Example 1]



(a) Initial state

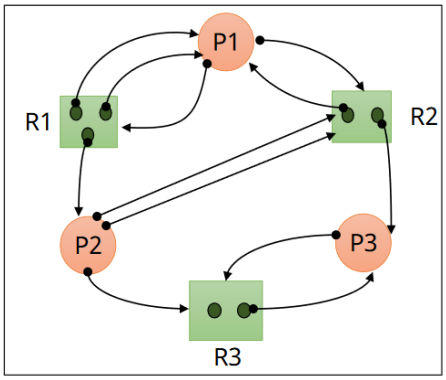


(b) Reduction by process P1

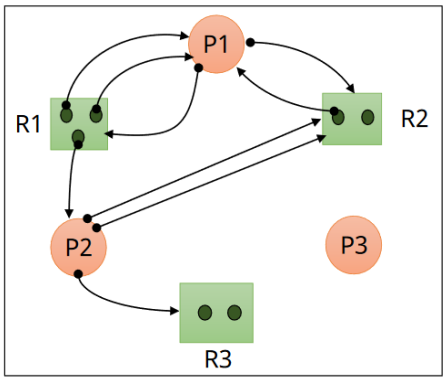


(c) Reduction by process P2

[Example 2]



(a) Initial state



(b) Reduction by process P3

P3 외 Unblocked Process 없음

[특징]

- High Overhead: 검사 횟수 또는 Node 수가 많은 경우

Deadlock Avoidance vs Detection

- **Deadlock Avoidance**
  - 최악의 경우를 생각
  - 앞으로 일어날 일을 고려해 예방하기 때문에 Deadlock 발생하지 않음
- Deadlock Detection
  - 최선의 경우를 생각

- 현재 상태만 고려하기 때문에 Deadlock 발생 시 Recovery 과정 필요

Deadlock Recovery (교착 상태 복구)

> Deadlock을 검출한 후 해결하는 과정

- Process Termination
  - Deadlock 상태에 있는 프로세스 강제 종료, 이후 재시작
- Resource Preemption
  - Deadlock 상태 해결 위해 선점할 자원 선택, 해당 자원을 갖고 있는 프로세스에서 자원을 뺏고 강제 종료

Process Termination

> Deadlock 상태인 프로세스 중 일부 종료

[Termination Cost Model]

- 종료 시킬 Deadlock 상태의 프로세스 선택
- Termination Cost 기준: 시스템에 맞게 선택, cost가 최소화되도록 선택
  - 프로세스 우선순위, 프로세스 종류, 총 수행 시간, 남은 수행 시간, 종료 비용 등

[종료 프로세스 선택 기준 예시]

- Lowest Termination Cost Process First
  - cost를 정한 후, 제일 적은 비용이 드는 프로세스를 먼저 종료
  - 장점: 단순하고 계산에 필요한 오버헤드 적음
  - 단점: 불필요한 프로세스가 종료될 가능성이 높음
- Minimum Cost Recovery
  - 최소 비용으로 deadlock 상태를 해소할 수 있는 프로세스를 먼저 종료
  - 장점: 최소 비용이 들도록 최고의 선택을 할 수 있음
  - 단점: 모든 경우의 수를 고려해야 하기 때문에 복잡하고 오버헤드가 높음 $O(2^k)$

Resource Preemption

> Deadlock 상태 해결 위해 선점할 자원 선택, 해당 자원을 갖고 있는 프로세스를 강제 종료

- Deadlock 상태가 아닌 프로세스가 종료될 수 있음
- 해당 프로세스는 이후 재시작 됨

[선점 자원 선택 기준]

- Preemption Cost Model 필요
- Minimum Cost Recovery Method 사용

Checkpoint Restart Method

> 프로세스 수행 중 특정 지점(checkpoint)마다 context 저장, 해당 프로세스가 강제 종료 되면 가장 최근의 checkpoint에서 재시작