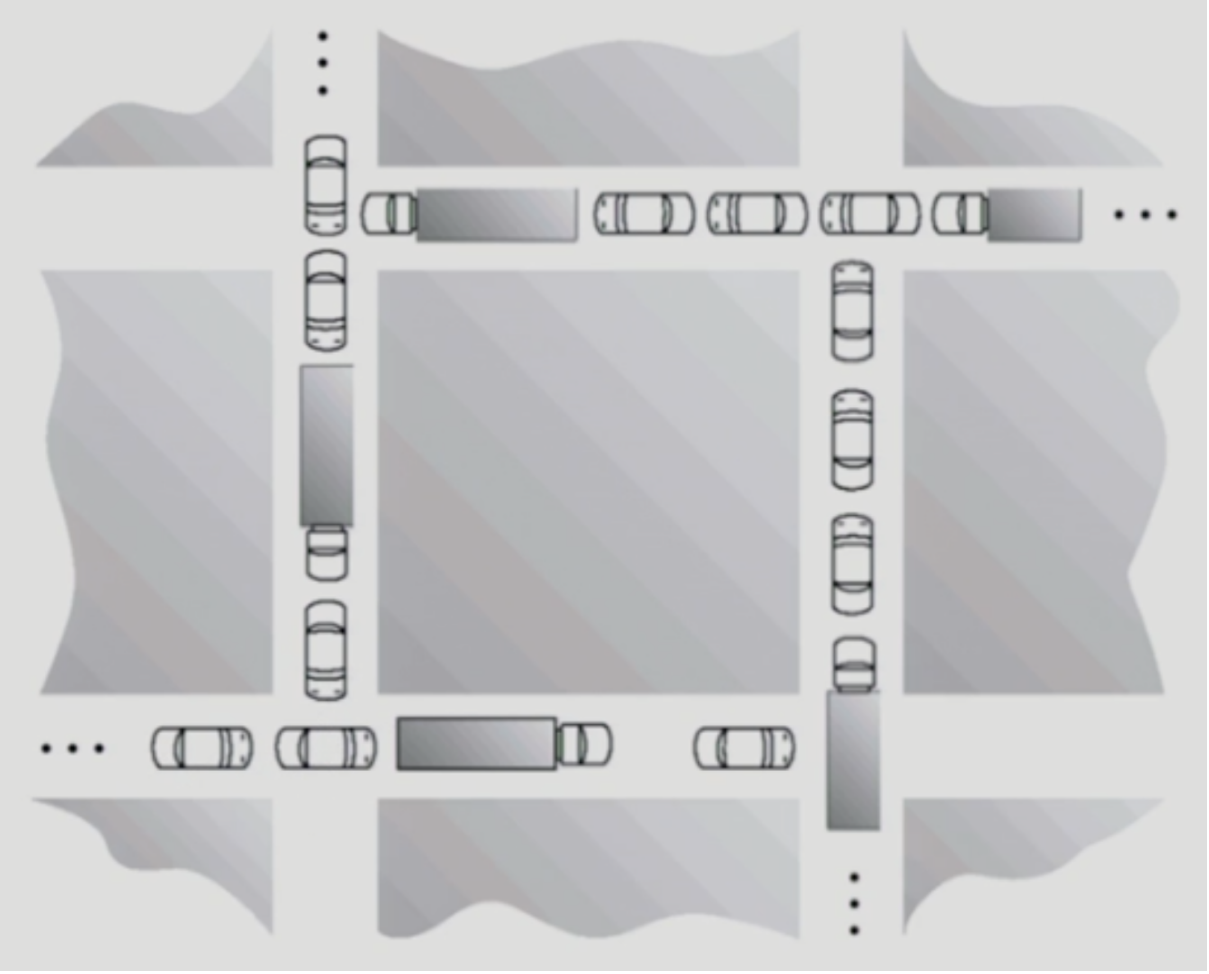
3주차_데드락

상태	진행 중
강의	CS 스터디
작성일	@2024년 2월 14일

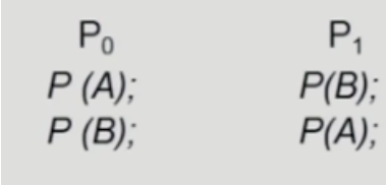
교착 상태 (deadlock)



The Deadlock Problem

- **Deadlock**
  - 일련의 프로세스들이 서로가 가진 자원을 기다리며 block된 ㅏㅅㅇ태
- Resource(자원)

    - 하드웨어, 소프트웨어 등을 포함하는 개면
    - (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 P2각각이 하나의 tape drive를 보유한 채 다른 하나를 기다리고 있다.
- Deadlock Example 2
    - Binary semaphores A and B



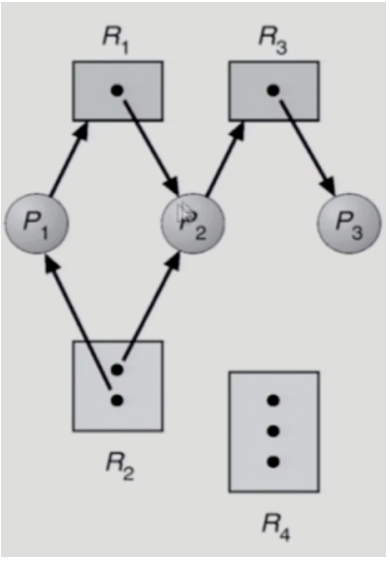
Deadlock 발생의 4가지 조건

- Mutual exclusion(상호 배제)
    - 매 순간 하나의 프로세스만이 자원을 사용할 수 있음
- No preemption(비선점)
    - 프로세스는 자원을 스스로 내어놓을 뿐 강제로 빼앗기지 않음
- Homd and wait(보유 대기)
    - 자원을 가진 프로세스가 다른 자원을 기다릴 때 보유 자원을 놓지 않고 계속 가지고 있음
- Circular wait(혼합 대기)
    - 자원을 기다리는 프로세스간에 사이클이 형성되어야 함
    - 프로세스P0, P1, ..., Pn이 있을 때
      P0은 P1이 가진 자원을 기다림
      P1은 P2가 가진 자원을 기다림

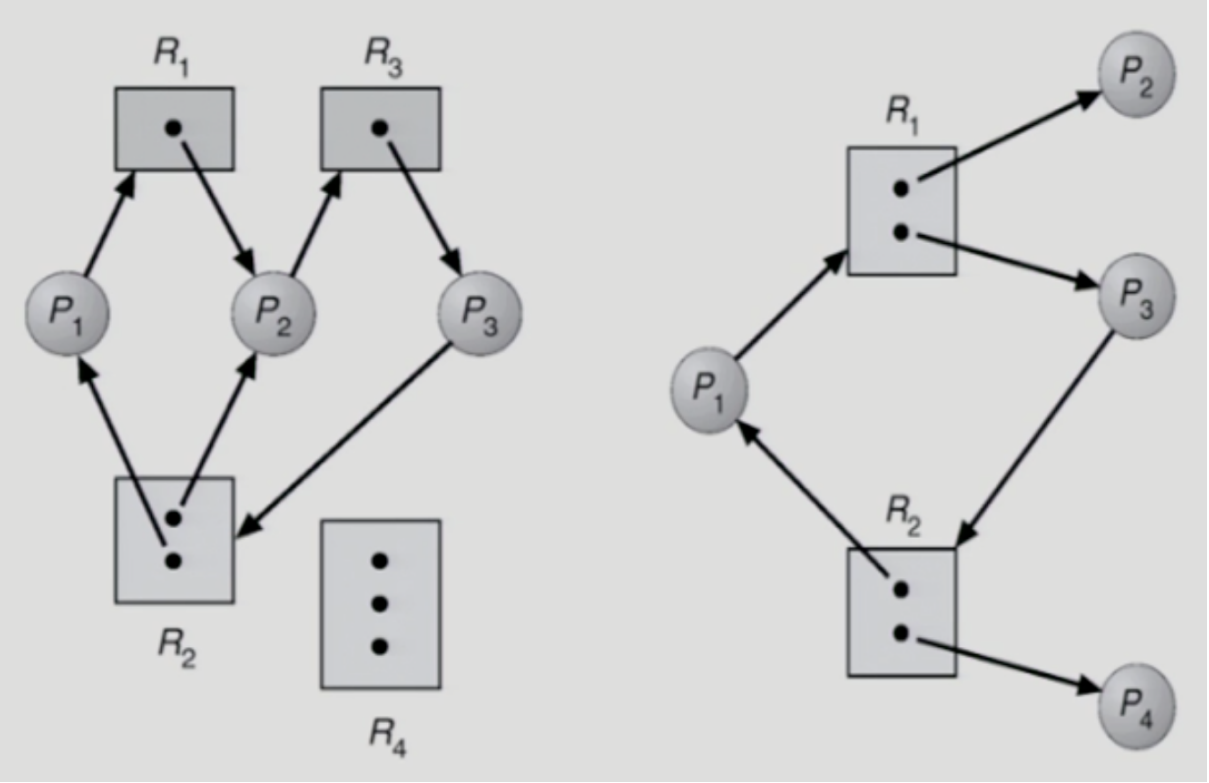
Pn-1은 Pn이 가진 자원을 기다림
Pn은 P0이 가진 자원을 기다림

Resource-Allocation Graph (자원할당 그래프)

- Vertex
  - Process P = {P1, P2, ..., Pn}
  - Resource R = {R1, R2, ..., Rm}
- Edge
  - request edge Pi → Rj
  - assignment edge Rj → Pi



Resource-Allocation Graph



- 왼쪽 그림은 deadlock, 오른쪽 그림은 deadlock 아님
- 그래프에 cycle이 없으면 deadlock이 아니다
- 그래프에 cycle이 있으면
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Deadlock의 처리 방법

- Deadlock Prevention
  - 자원 할당 시 Deadlock의 4가지 필요 조건 중 어느 하나가 만족되지 않도록 하는 것
- Deadlock Avoidance
  - 자원 요청에 대한 부가적인 정보를 이용해서 deadlock의 가능성이 없는 경우에만 자원을 할당
  - 시스템 state가 원래 state로 돌아올 수 있는 경우에만 자원 할당
- Deadlock Detection and recovery

    - Deadlock 발생은 허용하되 그에 대한 deletion루틴을 두어 deadlock 발견시 recover
- Deadlock Ignorance
    - Deadlock을 시스템이 책임지지 않음
    - UNIX등 포함한 대부분의 OS가 채택

Deadlock Prevention

- Mutual Exclusion
    - 공유해서는 안되는 자원의 경우 반드시 성립해야 함
- Hold and Wait
    - 프로세스가 자원을 요청할 때 다른 어떤 자원도 가지고 있지 않아야 한다.
    - 방법 1. 프로세스 시작 시 모든 필요한 자원을 할당받게 하는 방법
    - 방법 2. 자원이 필요할 경우 보유 자원을 모두 놓고 다시 요청
- No Preemption
    - process가 어떤 자원을 기다려야하는 경우 이미 보유한 자원이 선점됨
    - 모든 필요한 자원을 얻을 수 있을 때 그 프로세스는 다시 시작 된다.
    - State를 쉽게 save하고 restore할 수 있는 자원에서 주로 사용(CPU, memory)
- Circular Wait
    - 모든 자원 유형에 할당 순서를 정하여 정해진 순서대로만 자원 할당
    - 예를 들어 순서가 3인 자원 Ri를 보유 중인 프로세스 순서가 1인 자원 Rj을 할당받기 위해서는 우선 Ri를 release해야 한다.

⇒ Utilization 저하, throughput 감소, starvation 문제

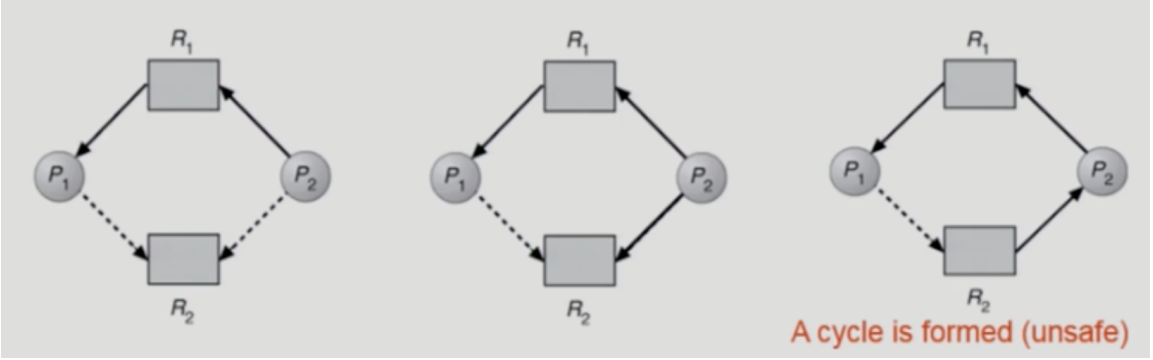
Deadlock Avoidance

- Deadlock Avoidance
    - 자원 요청에 대한 부가적인 정보를 이용해서 deadlock으로 부터 안전(safe)한지를 동적으로 조사해서 안전한 경우에만 자원을 할당

    - 가장 단순하고 일반적인 모델은 프로세스들이 필요로 하는 각 자원별 최대 사용량을 미리 선언하도록 하는 방법임
- safe state
    - 시스템 내의 프로세스들에 대한 safe sequence가 존재하는 상태
- safe sequence
    - 프로세스의 sequence <P1, P2, ..., Pn>이 safe하려면 Pi(1 ≤ i ≤ n)의 자원 요청이 "가용 자원 + 모든 Pi(j < i)의 보유 자원"에 의해 충족되어야 함
    - 조건을 만족하면 다음 방법으로 모든 프로세스의 수행을 보장
        - P_i의 자원 요청이 즉시 충족될 수 없으면 모든 P_i(j < i)가 종료될 때까지 기다린다.
        - P_(i-1)이 종료되면 Pi의 자원요청을 만족시켜 수행한다.

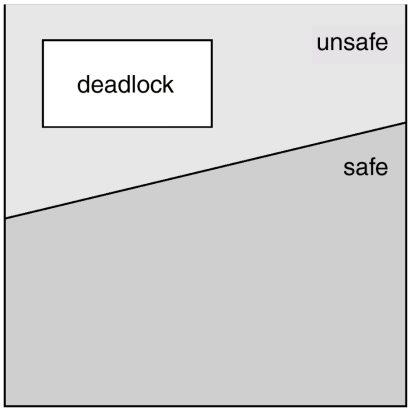
Resource Allocation Graph alogorithm

- Claim edge $P_i \rightarrow R_j$
    - 프로세스 Pi가 자원 Rj를 미래에 요청할 수 있음을 뜻함(점선으로 표시)
    - 프로세스가 해당 자원 요청시 request edge로 바뀜(실선)
    - Rj가 release되면 assignment edge 는 다시 calm edge로 바뀐다.
- request edge의 assignment edge 변경시 (점선을 포함하여) cycle이 생기지 않는 경우에만 요청 자원을 할당한다.
- Cycle 생성 여부 조사시 프로세스의 수가 n일때 $O(n^2)$ 시간이 걸린다.



Deadlock Avoidance

- 시스템이 safe state에 있으면
  ⇒ no deadlock
- 시스템이 unsafe state에 있으면
  ⇒
  possibility of deadlock
- Deadlock Avoidance
  - 시스템이 unsafe state에 들어가지 않는 것을 보장
  - 2가지 경우의 avoidance 알고리즘
    - Single instance per resource types
      - Resource Allocation Graph algorithm 사용
    - Multiple instances per resource types
      - Banker's Algorithm 사용



Example of Banker's Algorithm

- 5 processes $P_0 P_1 P_2 P_3 P_4$
- 3 resource types A(10), B(5), and C(7) instances.
- Snapshot at time T_0

	Allocation	*Max*	*Available*	*Need*
	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
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- 자원의 여유가 있더라도 자원을 최대로(Need) 요청했을 때 가용자원(Available)으로 처리가 안되면 요청 처리가 안된다.
- sequence < P_1, P_3, P_4, P_2, P_0 >가 존재하므로 시스템은 safe state

P_1 request (1, 0, 2)

- Check that Need ≤ Available, that is, (1, 2, 2) ≤ (3, 3, 2) ⇒ true.

	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	

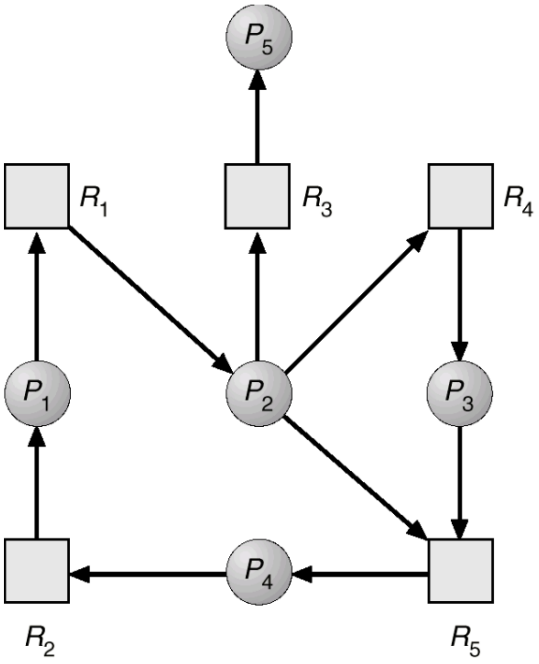
- Saftey algorithm에 의하면 sequence < P_1, P_3, P_4, P_2, P_0 > 는 safe!
- Can request for (3, 3, 0) by P_4 be granted?
- Can request for (0, 2, 0) by P_0 be granted?

Deadlock Detection and Recovery

- Deadlock Detection
  - Resource type 당 single instance인 경우
    - 자원할당 그래프에서늬 cycle이 곧 deadlock을 의미
  - Resource type당 multiple instance인 경우
    - Banker’s algorithm과 유사한 방법 활용
- Wait-for graph 알고리즘

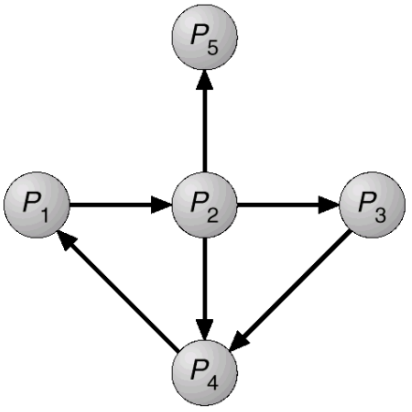
- Resource type 당 single instance인 경우
- Wait-for graph
  - 자원할당 그래프의 변형
  - 프로세스만으로 node 구성
  - P_i가 가지고있는 자원을 P_k가 기다리는 경우 P_k → P_i
- Algorithm
  - Wait-for graph에 사이클이 존재하는지를 주기적으로 조사
  - $O(n^2)$

Deadlock Detection and Recovery (자원당 인스턴스가 하나인 경우)



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

(a)는 데드락이 있음 (cycle이 무려 2개!)

자원의 최대 사용량을 미리 알릴 필요 없음 → 그래프에 점선이 없음

Deadlock Detection and Recovery (자원당 인스턴스가 여러개인 경우)

- Resource tyoe 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	

- No deadlock : sequence < P_0, P_2, P_3, P_1, P_4 > will work!

“Request”는 추가 요청 가능량이 아니라 현재 실제로 요청한 자원량을 나타냄

- P_2 requests an additional instance type of C.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State if System?
  - P_0 완료 후 자원 반납 가능, 그러나 충분하지 않음
  - Deadlock 존재 (P_1, P_2, P_3, P_4)

Deadlock Detection and Recovery

- Recovery
  - Process termination
    - Abort all deadlocked processes
    - Abort one process at a time until the deadlock cycle is eliminated
  - Resource Preemption
    - 비용을 최소화할 victim의 선정
    - safe state로 rollback하여 process를 restart
    - Starvation 문제
      - 동일한 프로세스가 계속해서 victim으로 선정되는 경우
      - cost factor에 rollback 횟수도 같이 고려

Deadlock Ignorance

- Deadlock이 일어나지 않는다고 생각하고 아무런 조치도 취하지 않음

- Deadlock이 매우 드물게 발생하므로 deadlcok에 대한 조치 자체가 더 큰 overhead일 수 있음.
- 만약, 시스템에 deadlock이 발생한 경우 시스템이 비정상적으로 작동하는 것을 사람이 느낀 후 직접 process를 죽이는 등의 방법으로 대처
- UNIX, Windows 등 대부분의 범용 OS가 채택 → 요즘은 대부눈 deadlock Ignorance를 사용