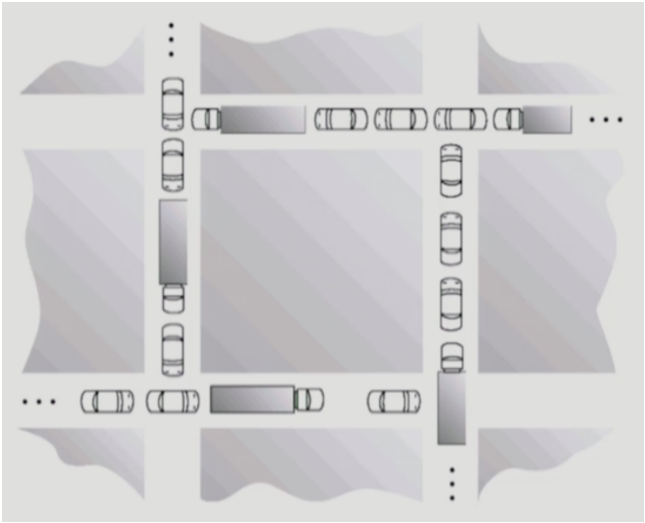
7. Deadlocks

▼ 목차

1. Deadlock (교착상태)

일련의 프로세스들이 서로가 가진 자원을 기다리며 block된 상태
일부 자원을 가지고 있으면서 서로의 자원을 요구하는 상태



세마포어 변수 A, B가 있고 프로세스 P1이 P(A), P(B)을 순서대로 호출하고 P2가 P(B), P(A)를 호출한다고 가정하면, P1과 P2가 서로 각

프로세스가 자원을 사용하는 절차에는 Request, Allocate, Use, Release가 있다.
Request : 자원을 요청하고, 만약 다른 프로세스가 자원을 사용하고 있어 받을 수 없다면 대기한다.
Allocate : 자원을 받는다.
Use : 프로세스가 받은 자원을 사용한다.
Release : 프로세스가 자원을 놓아준다.

Deadlock은 모든 프로세스가 Request 상태가 되어있는 상황이다.

2. Deadlock 발생 조건 4가지

1. Mutual exclusion (상호 배제)

하나의 프로세스만 자원을 독점
- 매 순간 하나의 프로세스만이 자원을 사용할 수 있다.

2. No preemption (비선점)

자원을 빼앗기지 않음
- 프로세스는 OS에 의해 강제로 자원을 빼앗기지 않는다.

3. Hold and wait (보유 대기)

가진 자원을 내놓지 않고 추가 자원 요청
- 자원을 가진 프로세스가 다른 자원을 기다릴 때, 보유하고 있는 자원을 놓지 않고 계속 가지고 있다.

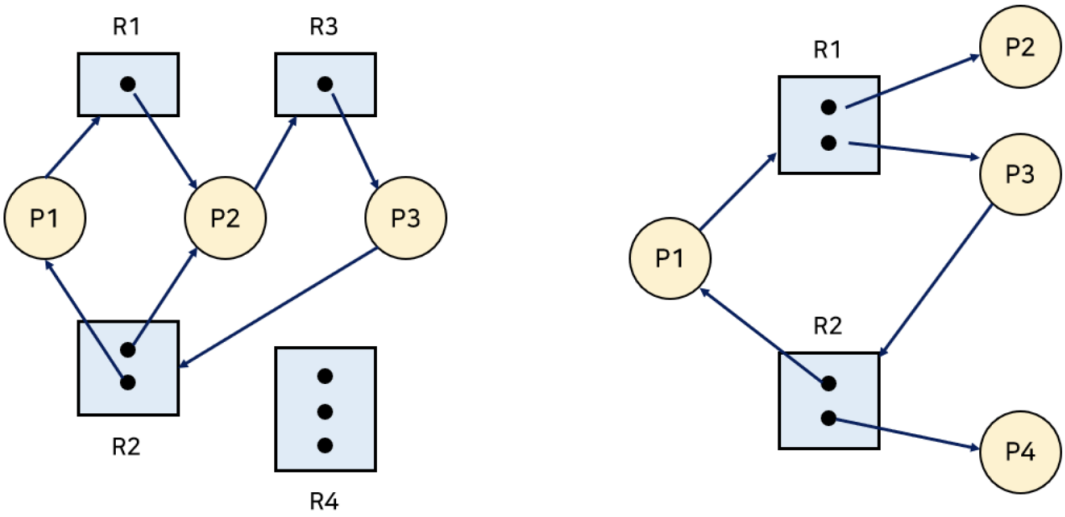
4. Circular wait (순환 대기)
- 자원을 기다리는 프로세스 간에 사이클이 형성되어야 한다.

ex) 프로세스 p0, p1, ... , pn이 있을 때 p0은 p1을 기다리고 p1은 p2를 기다리고, ... , pn은 p0을 기다린다.

---

3. Resource-Allocation Graph(자원 할당 그래프)

데드락이 발생했는지 알아보기 위해 사용



R은 자원이고 P는 프로세스를 의미한다. 자원 내의 동그라미는 자원(인스턴스)의 개수이다.
자원 → **프로세스**로 향하는 간선은 해당 자원을 프로세스가 보유 중(Allocate)이라는 의미이고, **프로세스** → **자원**으로 향하는 간선은 프로세

만약 **그래프에 사이클(Cycle)이 없다면 Deadlock이 아니다.**
반면, **사이클이 있다면 Deadlock이 발생할 수 있다.**

정확히 말하면, **자원당 하나의 인스턴스만 있는 경우엔 Deadlock이고**
여러 인스턴스가 존재하는 경우엔 Deadlock일 수도 있고 아닐 수도 있다.

---

4. Deadlock 처리 방법

위로 갈수록 더 강력한 방법

- **Deadlock Prevention**
  - 자원 할당 시 Deadlock의 4가지 필요 조건 중 어느 하나가 만족되지 않도록 하는 것
- **Deadlock Avoidance**
  - 자원 요청에 대한 부가적인 정보를 이용하여 deadlock의 가능성이 없는 경우 (즉, 시스템 state가 원래 state 돌아올 수 있는 경우)
- **Deadlock Detection and Recovery**
  - deadlock 발생은 허용하되 그에 대한 detection 루틴을 두어 deadlock 발견 시 recovery
- **Deadlock Ignorance**
  - Deadlock을 시스템이 책임지지 않는다. UNIX를 포함한 대부분의 OS가 채택하고 있는 방식이다.

  → deadlock은 자주 발생하는 이벤트가 아니기 때문에 데드락을 방지하기 위해 많은 오버헤드를 두는 것이 비효율적이다

4-1. Deadlock Prevention

데드락을 발생시키는 4가지 조건 중 어느 하나를 원천 차단하는 것

---

1) Mutual exclusion

- 배제 불가하다. 공유해서는 안되는 자원의 반드시 성립하게 되어 있다.
- 만약 한 자원을 여러 프로세스가 공유할 수 있다면 데드락이라는 문제도 생기지 않았을 것이다

2) Hold and Wait

- 프로세스가 자원을 요청할 때 다른 어떤 자원도 가지고 있지 않아야 한다
- hold를 하지 않도록 해서 deadlock이 발생하지 않도록 한다.

<방법 두 가지>

- **방법 1.** 프로세스 시작 시 모든 필요한 자원을 할당받게 하는 방법
  - 단점: 프로세스가 매 시점 필요한 자원이 다르기에 자원의 비효율성이 높아지므로 잘 사용하지 않는다
- **방법 2.** (자원을 hold한 상태인데 기다려야 한다면) 자원이 필요한 경우 보유자원을 모두 반환 뒤 다시 요청 → hold & wait
  - 자원 비효율성이 높지 않다.

3) No Preemption

- process가 어떤 자원을 기다려야 하는 경우 이미 보유한 자원이 선점됨(강제로 빼앗길 수 있음)
- 모든 필요한 자원을 얻을 수 있을 때 프로세스 다시 시작
- State를 쉽게 save저장, restore복구할 수 있는(=context switch) 자원에서 주로 사용 (CPU, memory)

4) Circular wait

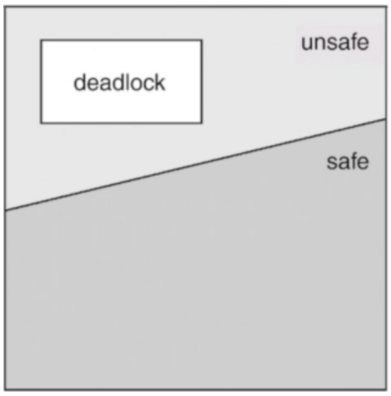
- 모든 자원 유형에 할당 순서를 정하여 정해진 순서대로만 자원을 할당한다.
- 이렇게 하면 쌍방향으로의 자원 요구가 이루어질 수 없으므로 Cycle이 생길 우려가 사라진다.
- 예를 들어 순서가 3인 자원 Ri 를 보유 중인 프로세스가 순서 1인 자원 Rj를 할당받기 위해서는 우선 Ri를 release해야 한다.

하지만 위의 방법들은 Utilization 자원 이용률 저하, throughput 처리량 감소, starvation 현상 등이 발생한다.

또한 발생 가능성이 낮은 deadlock 방지를 위한 제약 조건이 많기에 비효율적이다.

4-2. Deadlock Avoidance

- prevention: active하게 적극적으로 막는다.
- avoidance: 문제점으로부터 멀리 떨어지도록 해서 deadlock을 막는다.

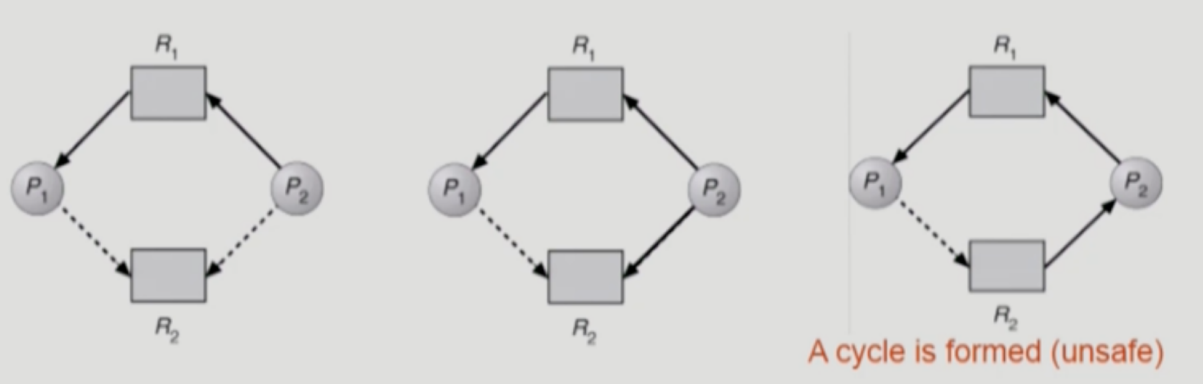


데드락의 가능성이 전혀 없는 경우에만 자원을 할당

프로세스가 시작되면 이 프로세스가 평생에 쓸 자원을 알고 있다고 가정해서 데드락을 피함
자원 요청시 데드락 발생 가능성이 있으면 자원을 주지 않음

---

1. 자원당 인스턴스 1개인 경우 → 자원 할당 그래프 사용

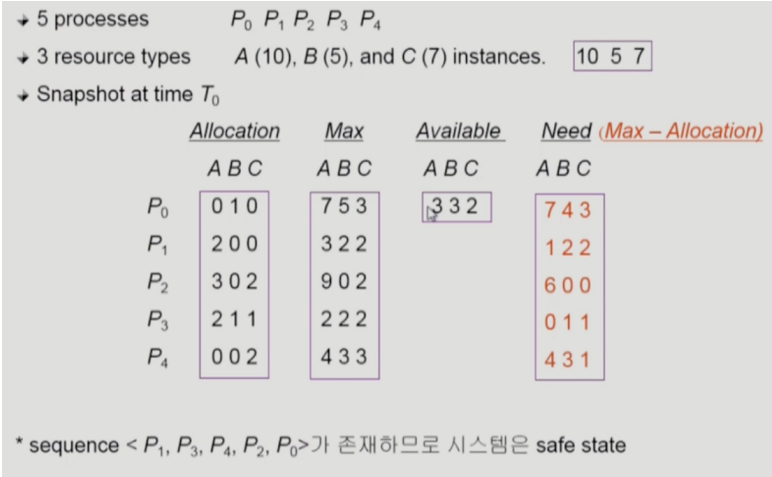


p2의 요청으로 r2자원을 넘겨주게 되면 사이클 (데드락)이 생길 가능성이 있기 때문에 자원을 주지 않는다

---

2. 자원당 인스턴스가 여러개 있는 경우

Banker's Algorithm 사용



Available ≥ Need : 수락

Available ≤ Need : 거절

→ p1 p3 p4 p2 p0 순으로 진행하게 되면 모두 처리 가능

최대 요청 자원을 모두 처리할 수 있다면 safe한 상태이자 데드락이 생기지 않는 것

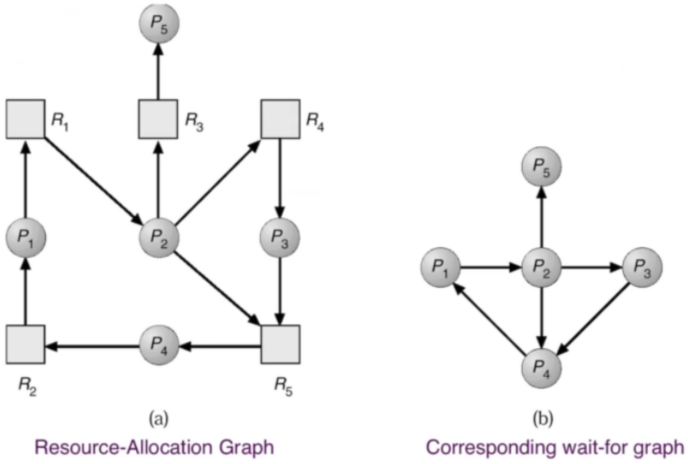
4-3. Deadlock Detection and Recovery

deadlock 발생은 그대로 놔두지만 그에 대한 detection routine을 두어 deadlock 발견 시 recovery

---

1. 자원당 인스턴스 1개

- Resource Allocation Graph에서의 cycle이 곧 deadlock을 의미
- Wait-for graph
  - 자원 할당 그래프에서 프로세스만으로 node 구성된 그래프를 말함
  - Wait-for graph에 사이클이 존재하는지 주기적으로 조사하는 Algorithm O(n^2) 동작



2. 자원당 인스턴스 여러개

- ✓ 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!

이 경우는 데드락이 발생하지 않음

그런데 만약 데드락이 발견되었다고 하면 바로 recovery 를 해야한다

- **Process termination - 프로세스 죽이기 (프로세스 종료)**
  1. deadlock에 연루된 프로세스들을 모두 죽임
  2. deadlock에 연루된 프로세스들을 하나씩 죽임 (deadlock이 없어질 때까지)
- **Resource preemption - 자원 뺏기 (프로세스 자원 뺏기)**
  - deadlock에 연루된 프로세스들로부터 자원을 뺏는 방법
  - 비용을 최소화할 victim의 선정
  - victim (희생양 프로세스)를 하나 선정해서 자원을 강제로 뺏음 → safe state로 rollback하여 process를 restart
  - 동일한 프로세스가 계속 victim으로 선정되면 starvation 문제 발생 → cost factor(비용적 측면)에서 rollback 횟수 같이 고려해야

4-4. Deadlock Ignorance

- 현재 대부분의 운영체제에서 채택한 방법으로, deadlock이 발생하지 않는다 생각하고 아무런 조치도 취하지 않는 것이다.
- deadlock은 매우 드물게 발생하기 때문에 조치를 취하는 것이 오히려 더 큰 오버헤드일 수 있기 때문이다.
- 만약 시스템에 deadlock이 발생한 경우 시스템이 느려지거나 프로세스가 멈춘 것을 사용자가 느낄 수 있으므로 일부 프로세스를 직접
- UNIX를 포함한 대부분의 OS가 채택하고 있다.