7️⃣ 데드락

Deadlock (교착상태)

일련의 프로세스들이 *서로가 가진 자원을 기다리며 block*된 상태

- Resource

  하드웨어, 소프트웨어 등을 포함하는 개념

  프로세스가 자원을 사용하는 절차 → Request, allocate, use, release

Deadlock 발생 4가지 조건

- *mutual exclusion*

  매 순간 하나의 프로세스만이 자원을 사용할 수 있음

- *no preemption*

  프로세스는 자원을 스스로 내어놓을 뿐 강제로 빼앗기지 않음

- *hold and wait*

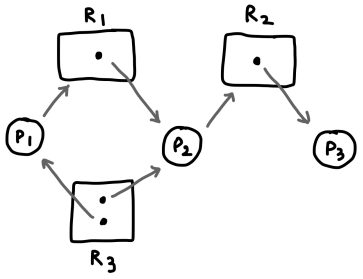
  자원을 가진 프로세스가 다른 자원을 기다릴 때 보유 자원을 놓지 않고 계속 가지고 있음

- *circular wait*

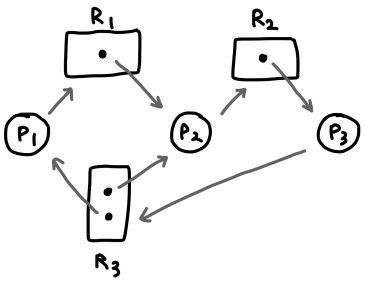
  자원을 가지는 프로세스 간에 사이클이 형성되어야 함

Resource-allovation graph (자원 할당 그래프)

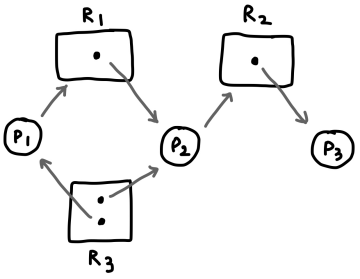
자원과 프로세스의 관계를 그림으로 그린 것



- Case 1
- Case 2



⇒ *deadlock*



⇒ *deadlock의 가능성*이 있다

P3가 자원을 반납하면 P2가지고 R1 반납하면 P4가 자원을 할당받음

그래프에 **cycle**이 존재하면

- 자원에 *instance 하나*면 deadlock
- 자원에 *instance 여러 개*면 cycle = deadlock은 꼭 아님

Deadlock 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

⇒ 데드락 방지

- **deadlock detection and recovery**

  deadlock 발생은 허용하되 그에 대한 *detection 루틴*을 두어 deadlock 발견 시 *recover*

- **deadlock ignorance**

  deadlock을 *시스템이 책임지지 않음*

  UNIX를 포함한 대부분의 OS가 채택

1. Deadlock prevention

- *mutual exclusion*

  공유해서는 안되는 자원의 경우 반드시 성립해야 함

  → 자원 자체의 성격에 따라야 하기 때문에 이 조건을 아예 방지할 수 없음

- *hold and wait*

  프로세스가 자원을 요청할 때 다른 어떤 자원도 가지고 있지 않아야 한다

  - 프로세스 시작 시 모든 필요한 자원을 할당받게 하는 방법
  - 자원이 필요한 경우 보유 자원을 모두 놓고 다시 요청

- *No preemption*

process가 어떤 자원을 기다려야 하는 경우 이미 보유한 자원이 선점됨

- 모든 필요한 자원을 얻을 수 있을 때 그 프로세스는 다시 시작된다
- state를 쉽게 save하고 restore할 수 있는 자원에서 주로 사용 (CPU, memory)

  → 빼앗아도 괜찮은 자원에 한해서

- *circular wait*

  모든 자원 유형에 할당 순서를 정하여 정해진 순서대로만 자원 할당

⇒ utilization 저하, throughput 감소, starvation 문제

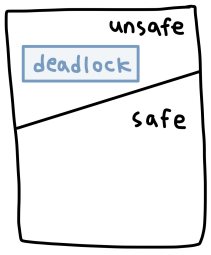
2. Deadlock avoidance

자원 요청에 대한 **부가정보**를 이용해 자원 할당이 deadlock으로 부터 안전한 지를 동적으로 조사해서 안전한 경우에만 할당

→ 부가 정보 : 자원을 최대로 쓰면 얼마나 쓸지 (**자원별 최대치**를 알고 있음) ~ 위험할 것 같으면 자원을 할당하지 않음

- 가장 단순하고 일반적인 모델은 프로세스들이 필요로 하는 각 자원별 최대 사용량을 미리 선언하도록 하는 방법
- **safe state** : 시스템 내의 프로세스들에 대한 *safe sequence가 존재하는 상태*
- **safe sequence**
  - 프로세스의 sequence가 safe하려면 Pi의 자원 요청이 "가용 자원 + 모든 Pj (j<i)의 보유 자원"에 의해 충족되어야 함

    → 가용 자원만 가지고 처리할 수 있는 프로세스가 있는지 찾음
  - 조건 만족하면 다음 방법으로 모든 프로세스의 수행 보장
    1. Pi의 자원 요청이 즉시 충족될 수 있으면 모든 Pj (j < i)가 종료할 때까지 기다린다
    2. P(i-1)이 종료되면 Pi의 자원요청을 만족시켜 수행한다



- 시스템이 *safe state*에 있으면 *no deadlock*
- 시스템이 *unsafe state*에 있으면 *possibility of deadlock*

Deadlock avoidance ⇒ 시스템이 unsafe state에 들어가지 않는 것을 보장

- single instance일 경우 : *resource allocation graph algorithm* 사용
- multiple instance일 경우 : *banker's algorithm* 사용

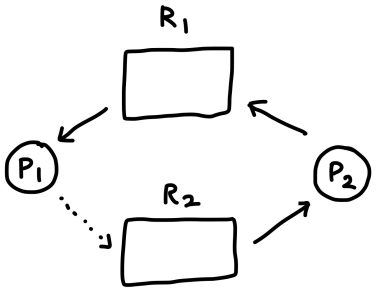
Resource-allocation graph algorithm

claim edge : Pi → Rj

- 프로세스 Pi가 자원 Rj를 *미래에 요청할 수 있음*을 뜻함 (**점선**)
- 프로세스가 *해당 자원 요청시* request edge로 바뀜 (**실선**)
- Rj가 release되면 assignment edge는 다시 claim edge로 바뀜

→ request edge의 assignment edge 변경 시 **점선 포함해서** cycle이 생기지 않는 경우에만 요청 자원을 할당

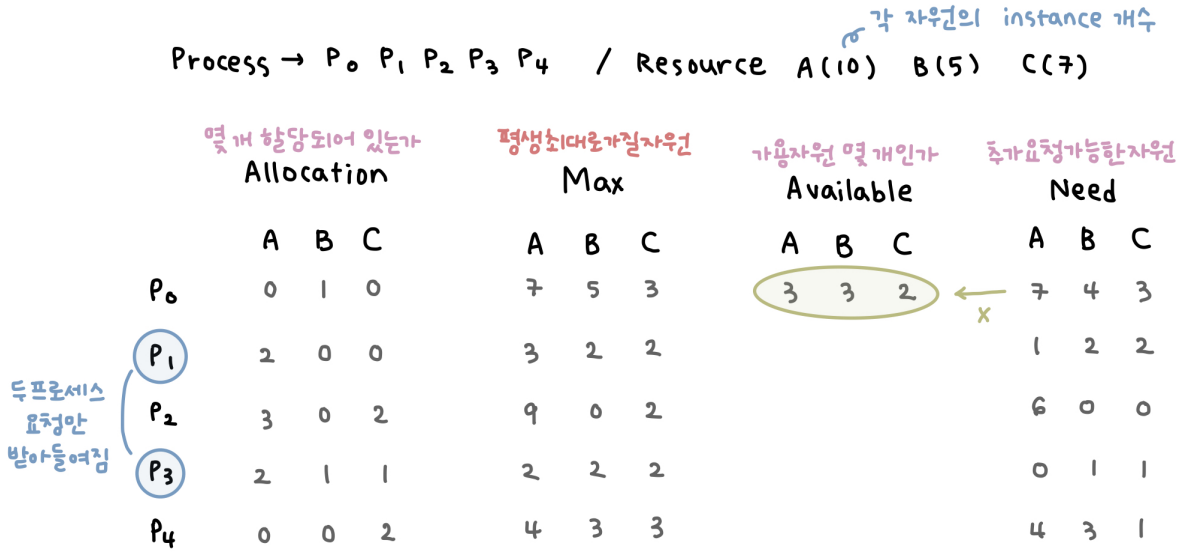
→ cycle 생성 여부 조사시 프로세스의 수가 n일 때 O(n^2)의 시간이 걸림



⇒ unsafe한 상황

Banker's algorithm

→ 테이블을 만들어서 *자원을 할당할지 안할지 결정*



- 자원에 여유가 있더라도 자원을 줬을 때 unsafe하면 할당하지 않음

  → 지금이 최대로 자원을 요청하는지 모르기 때문에
- sequence <P1, P3, P4, P2, P0> 가 존재하므로 safe state

  → *가용자원만으로 프로세스를 하나씩 종료시킬 수 있는 sequence 존재*

⇒ 이런 과정을 banker's algorithm이 하는 건 아니고 그냥 *비교, 할당 여부만 결정*

가정

- 모든 프로세스는 자원의 최대 사용량을 미리 명시
- 프로세스가 요청 자원을 모두 할당받은 경우 유한 시간 안에 이들 자원을 다시 반납한다

방법

- 자원 요청 시 safe 상태를 유지할 경우에만 할당
- *총 요청 자원의 수가 가용 자원의 수보다 적은 프로세스를 선택* (그런 프로세스 없으면 unsafe)
- 그런 프로세스가 있으면 그 프로세스에게 자원 할당
- 모든 프로세스가 종료될 때까지 이러한 과정 반복

3. Deadlock Detection and Recovery

Deadlock Detection

- Resource type당 single instance인 경우

  → 자원할당 그래프에서의 cycle이 곧 deadlock
- Resource type당 multiple instance인 경우

  → banker's algorithm과 유사한 방법 활용

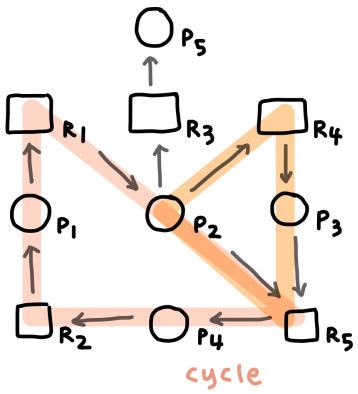
wait - for graph 알고리즘

- Resource type당 *single instance*인 경우
  - wait-for graph

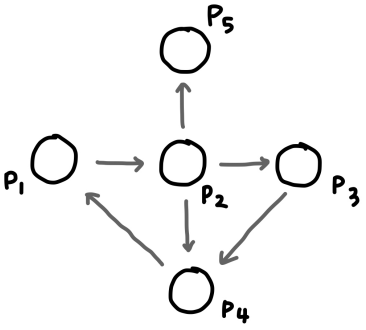
    자원 할당 그래프의 변형 / 프로세스만으로 node 구성 / Pi가 가지고 있는 자원을 Pk가 기다리는 경우 : Pk → Pi
  - algorithm

    wait - for 그래프에 사이클이 존재하는지를 주기적으로 조사 ⇒ O(n^2)

<Resource allocation graph>



<corresponding wait-for graph>



→ 자원 빼고 프로세스로만 만든 그래프

- Resource type당 *multiple instance*인 경우

  deadlock이 안 걸릴만한 프로세스 sequence를 찾음

  → 아무것도 요청하지 않은 프로세스가 가진 자원을 내놓는다고 가정

Recovery

- *process termination*

  → deadlock에 연루된 process를 죽임

  - 모든 프로세스를 죽임
  - 연루된 프로세스를 하나씩 죽여서 deadlock이 없어진지 확인
- *resource preemption*
  - 비용을 최소화할 victim의 선점
  - safe state로 rollback하여 process를 restart
  - starvation 문제
    - 동일한 프로세스가 계속해서 victim으로 선정되는 경우
    - cost factor에 rollback 횟수도 같이 고려 (비용이 적게 드는 프로세스 골라서 자원 뺏음)

Deadlock Ignorance

deadlock이 일어나지 않는다고 생각하고 아무런 조치도 취하지 않음

- deadlock이 매우 드물게 발생하므로 deadlock에 대한 조치 자체가 *더 큰 overhead*일 수 있음
- 만약 시스템에 deadlock이 발생한 경우 시스템이 비정상적으로 작동하는 것을 사람이 느낀 후 직접 process를 죽이는 방법 등으로 대처
- UNIX, Windows 등 대부분의 범용 OS가 채택