스마트시스템 운영체제
(LD01600)

김준철

정보시스템공학과

greensday@sungshin.ac.kr

7주차 강의

휴강(9.30)
(추석)

	주차	강의 목차
9.2	1	과목소개 / 운영체제 개요
9.9	2	컴퓨터 시스템 구조
9.16	3	process와 스레드1
9.22	4	process와 스레드2, CPU스케쥴링1
10.7	5	CPU스케쥴링2
10.14	6	process 동기화
10.21	7	교착 상태
10.28	8	중간고사
11.4	9	물리 메모리 관리
11.11	10	가상메모리 기초
11.18	11	가상메모리 관리
11.25	12	입출력시스템1
12.2	13	입출력시스템2, 파일시스템1
12.9	14	파일시스템2
12.16	15	기말고사

Operating Systems

ch.06 Deadlock 교착 상태

01 교착 상태의 개요

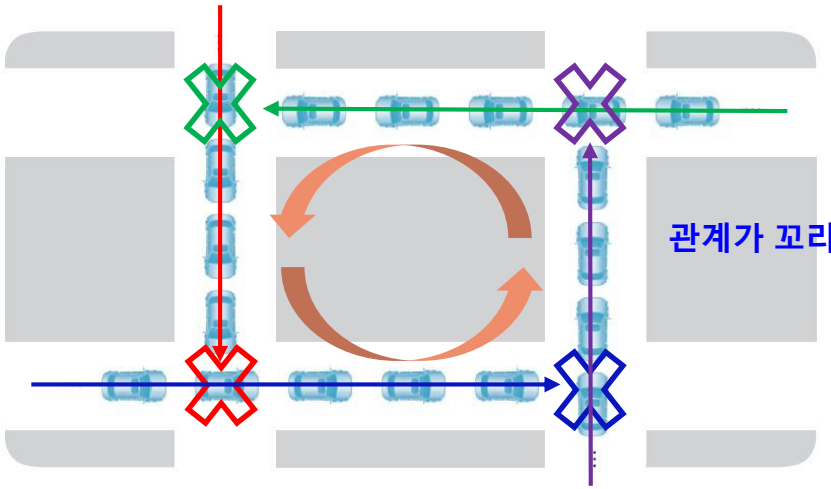
02 교착 상태 필요 조건

03 교착 상태 해결 방법

Deadlock(교착상태)의 정의

- **Deadlock**
  - 2개 이상의 process가 다른 process의 작업이 끝나기만 기다리며 작업을 더 이상 진행하지 못하는 상태

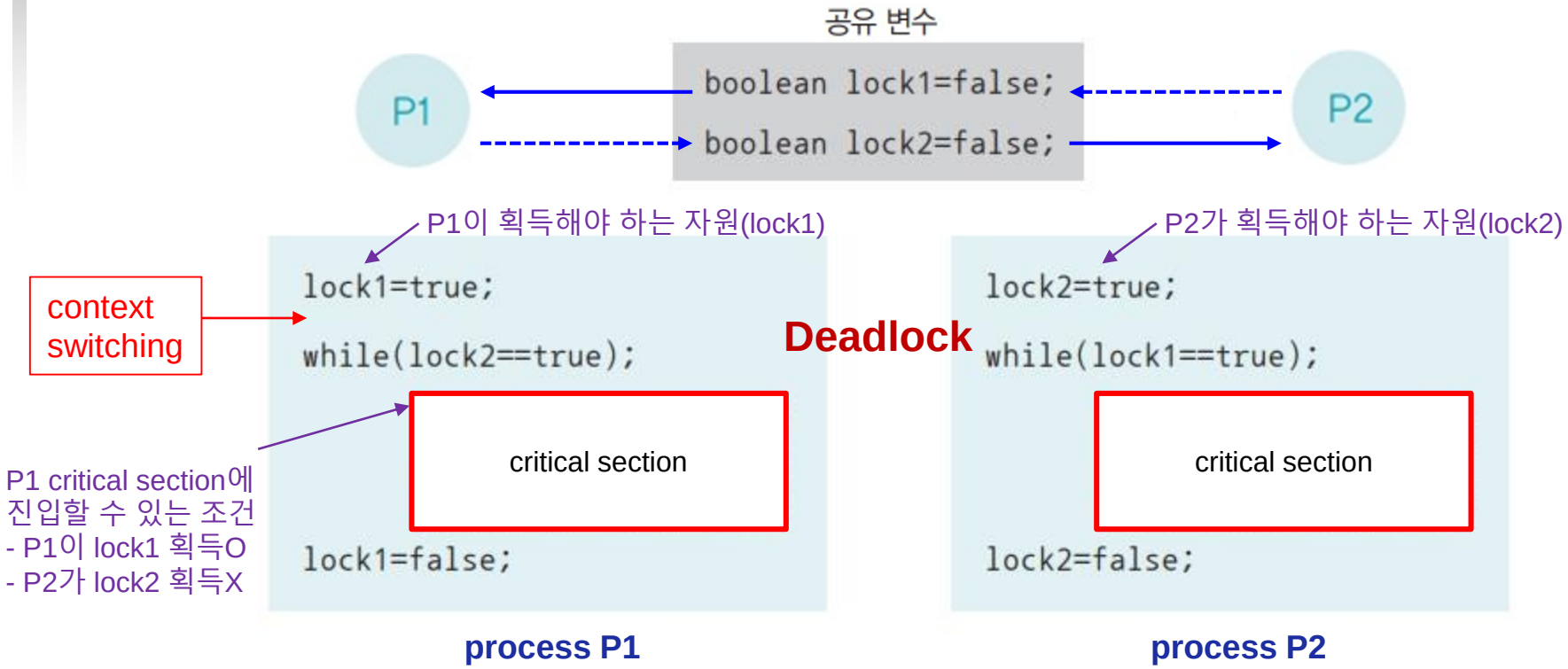
- **Starvation과 차이점**
  - Starvation(아사 현상) : 운영체제가 잘못된 정책을 사용하여 특정 process의 작업이 지연되는 문제
  - Deadlock(교착 상태) : 여러 process가 작업을 진행하다가 서로 간의 관계에 의해서 발생되는 진행불가 문제



[그림] 교착상태의 예: 교통마비상태

Deadlock의 발생

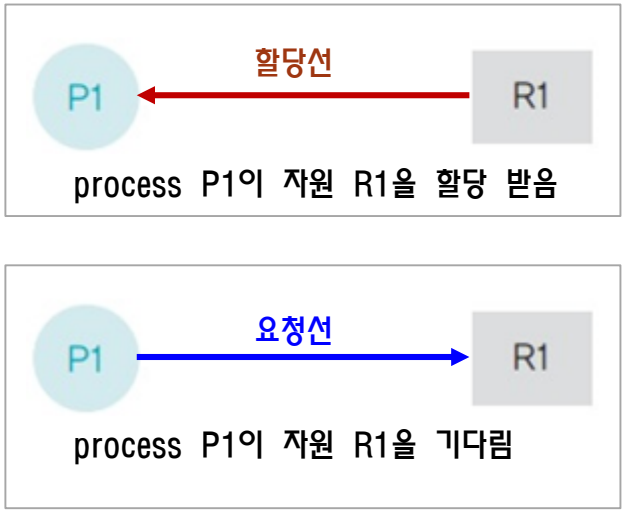
- **공유 변수**
  - Deadlock은 공유변수를 사용할 때도 발생 가능



[그림] deadlock를 발생시키는 critical section 진입 code

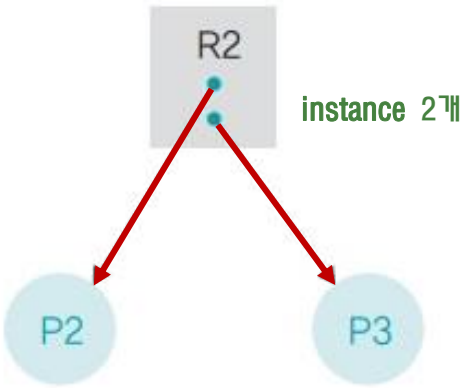
자원 할당 그래프 Deadlock 여부를 graph로 확인가능

- **자원 할당 그래프(Resource-Allocation Graph)**
  - 자원과 process의 관계를 그래프로 표현한 것
  - **process가 어떤 자원을 사용 중**이고 **어떤 자원을 기다리고 있는지**를 방향성 있게 나타냄
- **다중 자원(multiple resource)**
  - 여러 process가 하나의 자원을 동시에 사용하는 경우



원형: process, 사각형: 자원

[그림] 자원 할당 그래프

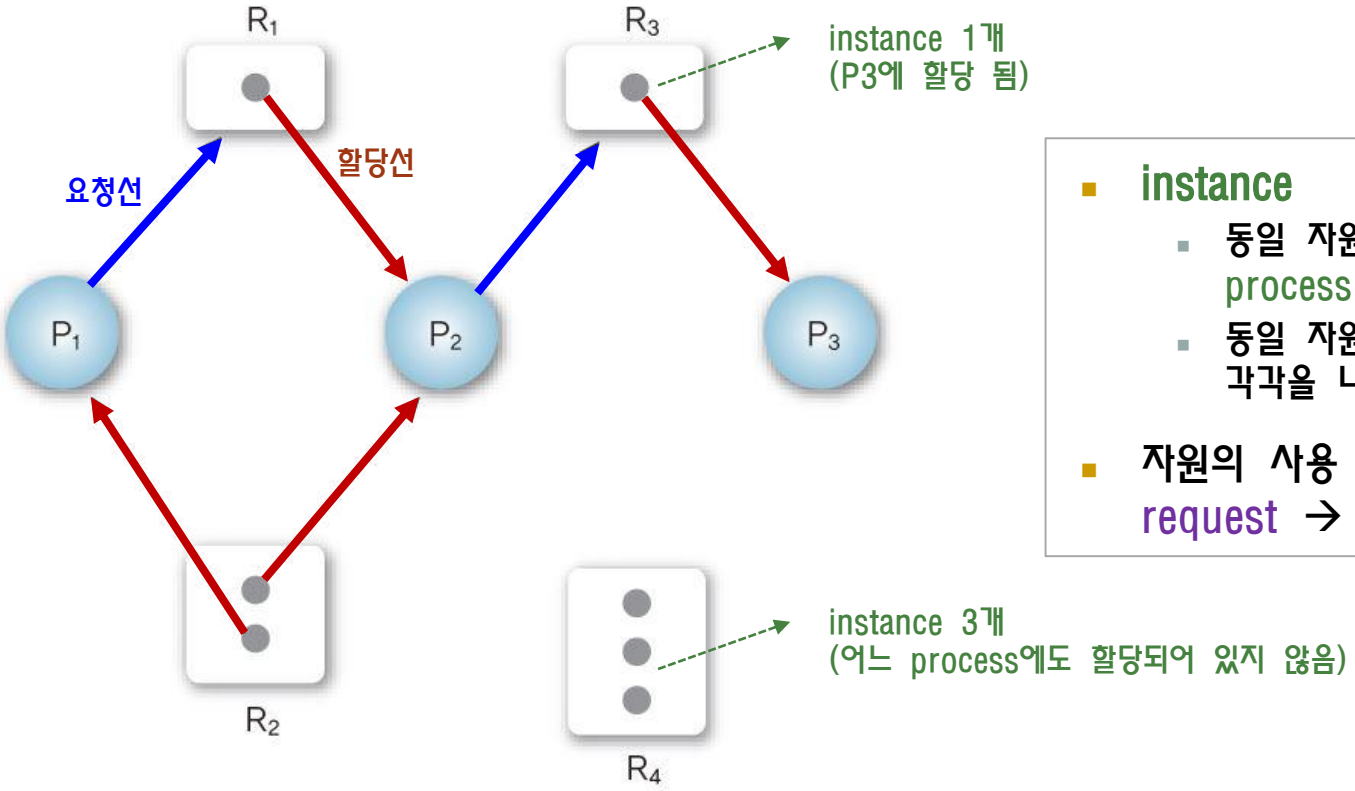


수용할 수 있는 process 수(instance)를 사각형 안에 작은 점으로 표시

[그림] 2개의 process를 수용할 수 있는 자원

자원 할당 그래프

- 자원 할당 그래프(Resource-Allocation Graph)의 예

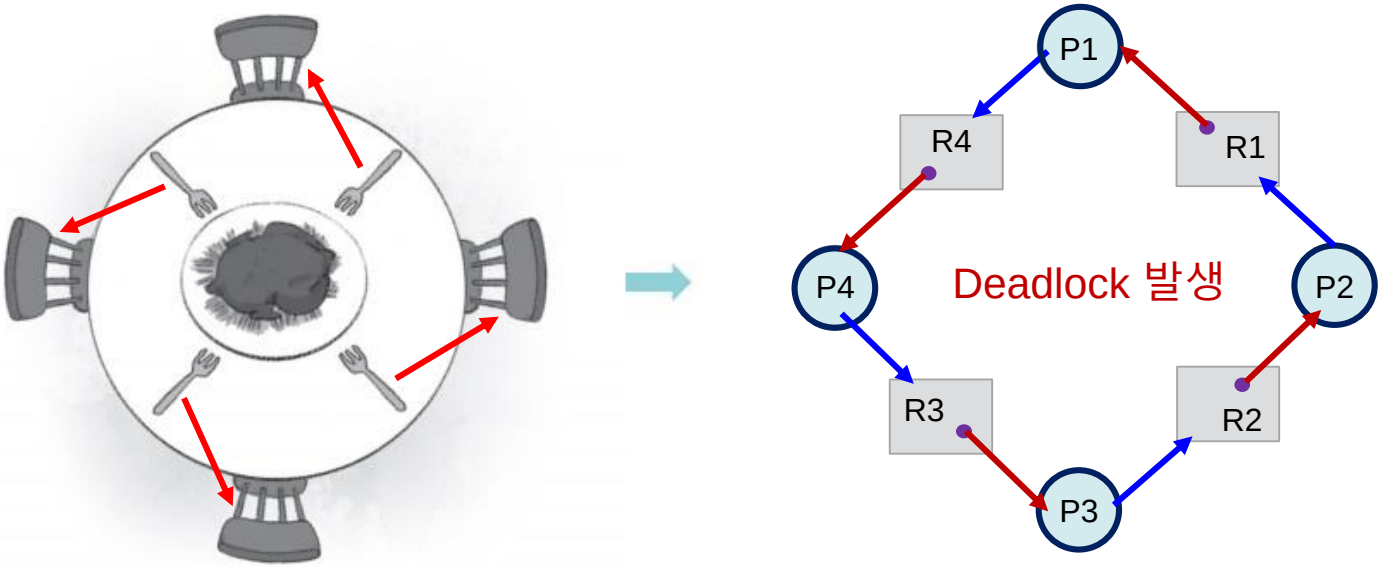


- instance
  - 동일 자원이 수용할 수 있는 process 수
  - 동일 자원이 여러 개 있을 때 각각을 나타냄
- 자원의 사용
  request → use → release

[그림] 자원 할당 그래프 예와 process 상태

자원 할당 그래프

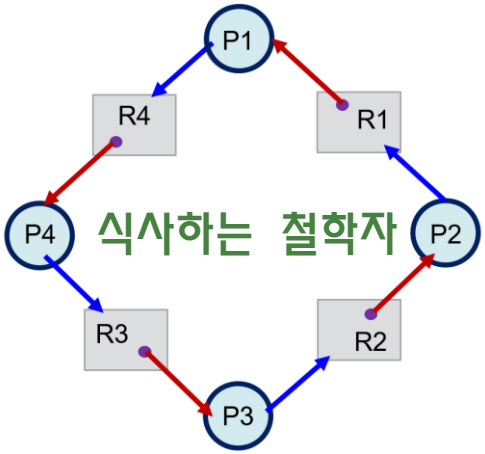
- **식사하는 철학자 문제**
  - 철학자는 오른손, 왼손에 모두 **포크**를 잡아야만 식사 가능
    - 조건: 왼쪽에 있는 **포크**를 잡은 뒤 오른쪽에 있는 **포크**를 잡을 수 있음



[그림] 식사하는 철학자 문제의 자원할당 그래프

Deadlock 필요조건

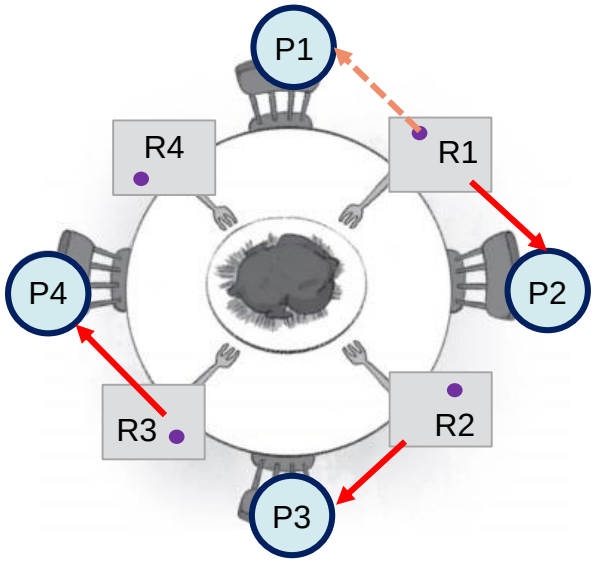
- Deadlock 발생의 4가지 필요 조건
  - 다음 4가지 조건이 **모두 만족**해야만 **deadlock 발생**(필요조건)
    - Deadlock을 회피하려면 아래 4개 조건 중 1개 조건만 만족하지 못하게 하면 됨



1. **mutual exclusion(상호 배제)**: 한 process가 사용하는 자원은 다른 process와 공유할 수 없는 **배타적인 자원**인 경우
2. **no-preemption(비선점)**: 한 process가 사용 중인 자원은 중간에 다른 **process가 빼앗을 수 없는 비선점 자원**인 경우
3. **hold and wait(점유와 대기)**: process가 어떤 **자원을 할당 받은 상태에서 다른 자원을 기다리는 상태**인 경우
4. **circular wait(원형 대기)**: 점유와 대기를 하는 process 간의 관계가 **원**을 이루는 경우 (**관계가 꼬리에 꼬리를 무는 형태**)

식사하는 철학자 문제와 Deadlock 필요조건

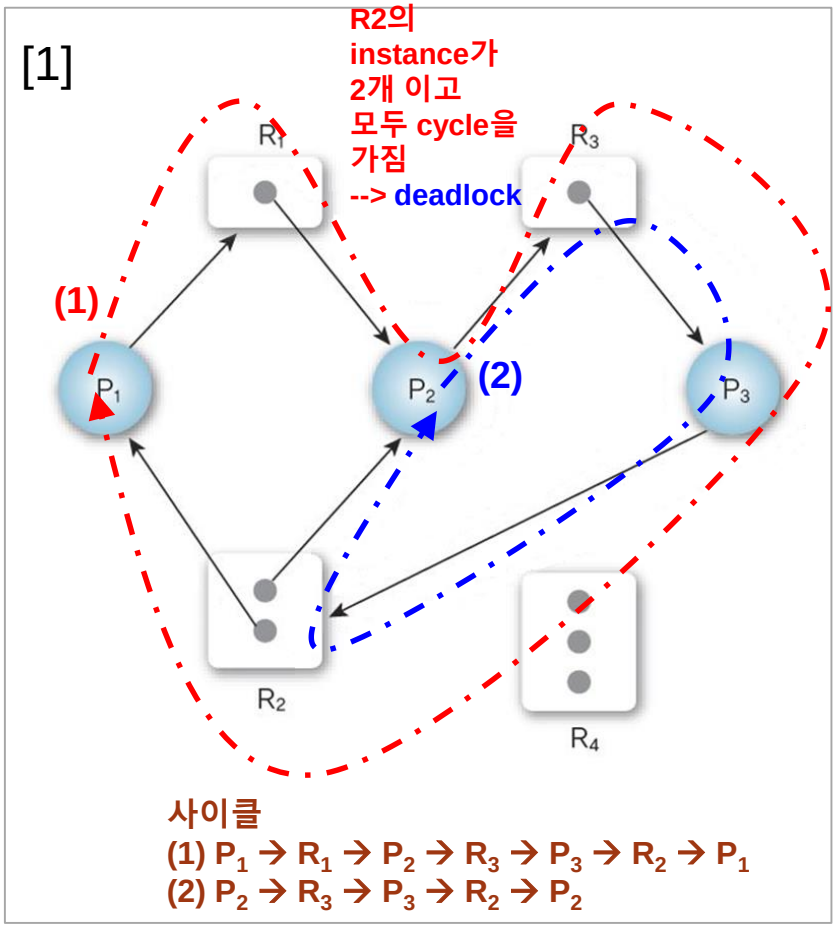
- **식사하는 철학자 문제와 deadlock 필요조건**
  - **mutual exclusion(상호 배제)**: 포크는 한 사람이 사용하면 다른 사람이 사용할 수 없는 배타적인 자원임
  - **no-preemption(비선점)**: 철학자 중 어떤 사람의 힘이 월등하여 옆 사람의 포크를 빼앗을 수 없음
  - **hold and wait(점유와 대기)**: 한 철학자가 두 자원(왼쪽 포크와 오른쪽 포크)을 다 점유하거나, 반대로 두 자원을 다 기다릴 수 없음
  - **circular wait(원형 대기)**: 철학자들은 둥그런 식탁에서 식사를 함, 원을 이룬다는 것은 선후 관계를 결정할 수 없어 문제가 계속 맴돈다는 의미



원형 식탁과 식사하는 철학자 문제
: 낮은 번호의 포크를 우선 들게 하는 방법
→ circular wait 파괴

자원 할당 그래프 – Deadlock의 표현

- deadlock의 할당 그래프와 사이클 예 mutual exclusion, no-preemption, hold and wait 가정



[그림] deadlock의 할당 그래프와 사이클

Deadlock 해결 방법

- deadlock 해결 방법

* deadlock을 사전에 방지하는 방법

1. **deadlock prevention(예방)** : deadlock을 유발하는 네 가지 조건 중 하나가 만족되지 않도록 하는 방식으로 deadlock 조건 4가지에 대하여 예방 방법이 각각 존재 함
2. **deadlock avoidance(회피)** : deadlock이 발생하지 않도록 자원 할당량을 조절하여 deadlock을 회피하는 방식

* deadlock이 생기면 사후에 처리하는 방법

3. **deadlock detection(검출)과 recovery(회복)**:
   deadlock의 발생은 허용하지만 검출 루틴을 만들어서 자원 할당 그래프를 모니터링하고, deadlock 발견 시 recovery(회복) 단계를 진행하도록 함

* deadlock이 생겨도 무시하고 책임지지 않는 방법

4. **deadlock ignorance(무시)**: deadlock을 책임지지 않음

1. Deadlock prevention (예방)

- deadlock 조건 4가지에 대하여 각각의 방식이 존재
  (mutual exclusion, no-preemption, hold and wait, circular wait)

(1) mutual exclusion(상호 배제) 예방

- 시스템 내의 **독점적으로 사용할 수 있는 모든 자원**(상호 배타적인 모든 자원)**을 없애버리는 방법**
  - 현실적으로 **모든 자원을 공유 가능한 상태로 할 수는 없음**

(2) no-preemption(비선점) 예방

- 자원을 빼앗을 수 있도록(preemptive) 만드는 방법 (예: CPU)
  - 현실적으로 Preemptive 방식을 적용할 수 없는 자원들이 있기 때문에 **모든 자원에 적용 할 수는 없음**

1. Deadlock prevention (예방)

(3) hold and wait(점유와 대기) 예방

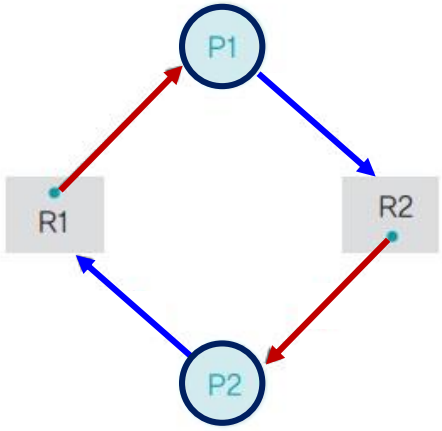
- process가 자원을 점유한 상태에서 다른 자원을 기다리지 못하게 하는 방법
- **해결방법**: '**전부 할당하거나 아니면 아예 할당하지 않는**' 방식을 적용

  → 필요한 모든 자원을 획득하지 못하면 가지고 있는 자원을 모두 버려야 함

  - **방식1. process 시작 시 모든 필요한 자원을 할당 받고 끝날 때 돌려줌**

    → 자원 낭비의 문제

    방식2. 필요할 경우 보유한 자원을 모두 놓고 모두 다시 요청하는 방식



P2가 R1 자원을 사용을 획득하려면,
보유하고 있는 R2자원을 반납하고
모든 자원을 다시 획득해야 함

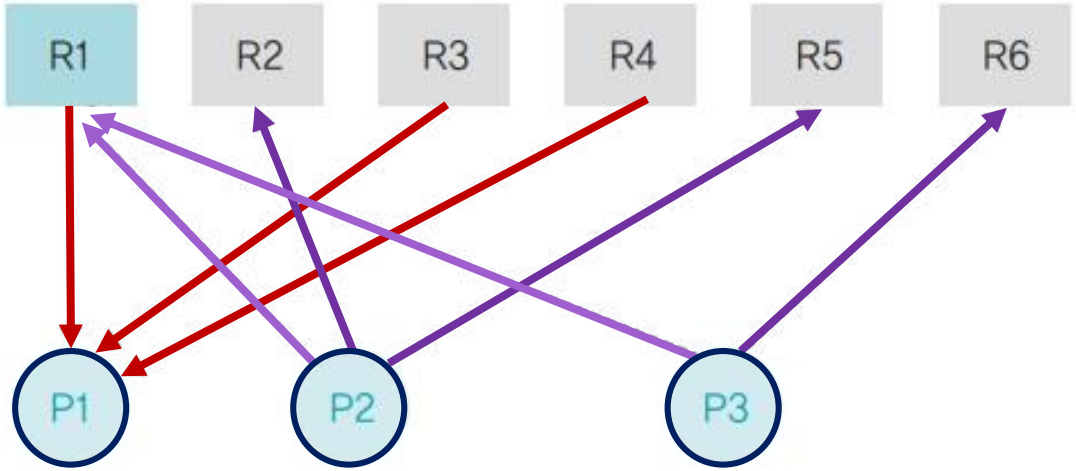
[그림] 전부 할당 또는 아얘 할당하지 않는 방식

1. Deadlock prevention (예방)

- **hold and wait 예방의 단점**

'전부 할당하거나 아니면 아예 할당하지 않는' 방식을 적용

  - process가 자신이 사용하는 모든 자원을 자세히 알기 어려움
  - 자원의 활용성이 떨어짐
  - 많은 자원을 사용하는 process가 적은 자원을 사용하는 process보다 불리함
  - 결국 **일괄 작업 방식**으로 동작 **(자원을 획득한 process 순서대로 실행되고 다음 프로세스는 앞의 process 가 끝나야 실행 가능)**



[그림] 필요한 자원을 모두 할당한 후 실행

R1 때문에
일괄처리로 동작함

1. Deadlock prevention (예방)

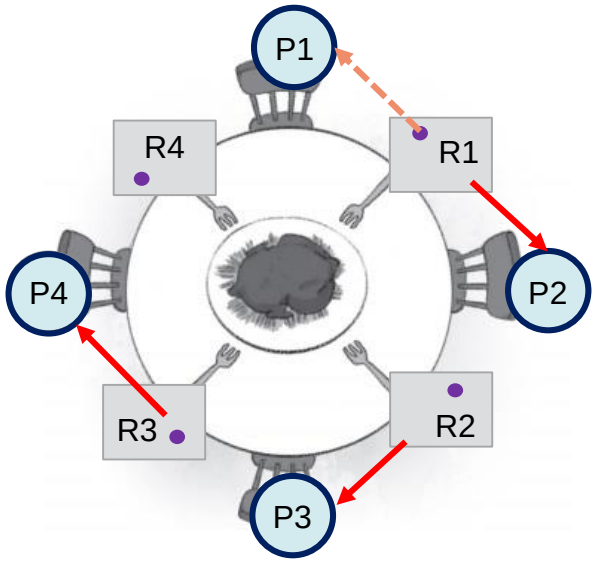
(4) circular wait(원형 대기) 예방

- 점유와 대기를 하는 process들이 **원형을 이루지 못하도록 막는 방법**

(방법) 자원을 획득하는 순서를 정함

> 예)
> 모든 자원에 숫자를 부여하고
> 어떤 순서에 따라서 자원을 할당

- circular wait 예방의 단점
  - 자원번호의 부여 방식의 선택 문제
  - process 작업 진행에 유연성이 떨어짐



양쪽 자원(포크) 중 번호가 낮은 것 부터 획득하도록 하면 원형을 이루지 못함
(예: P1은 R1과 R4중 낮은 번호인 R1을 취함)

[그림] 식사하는 철학자- 자원에 번호, 순서를 부여

1. Deadlock prevention (예방)

- **Deadlock prevention방법의 특징**
  - Deadlock을 유발하는 **네 가지 조건이 일어나지 않도록 제약**을 가하는 방법
  - **Deadlock은 자주 일어나지 않는데 prevention을 쓰는 것은 비효율적임**
    - 자원을 점유했다가 내려 놓기를 반복
    - 남는 자원이 있음에도 순서를 정해서 자원을 점유 못하게 막는 경우가 생김
    - 특히 hold and wait, circular wait는 process 작업 방식을 제한하고 자원을 낭비하기 때문에 사용할 수 없음

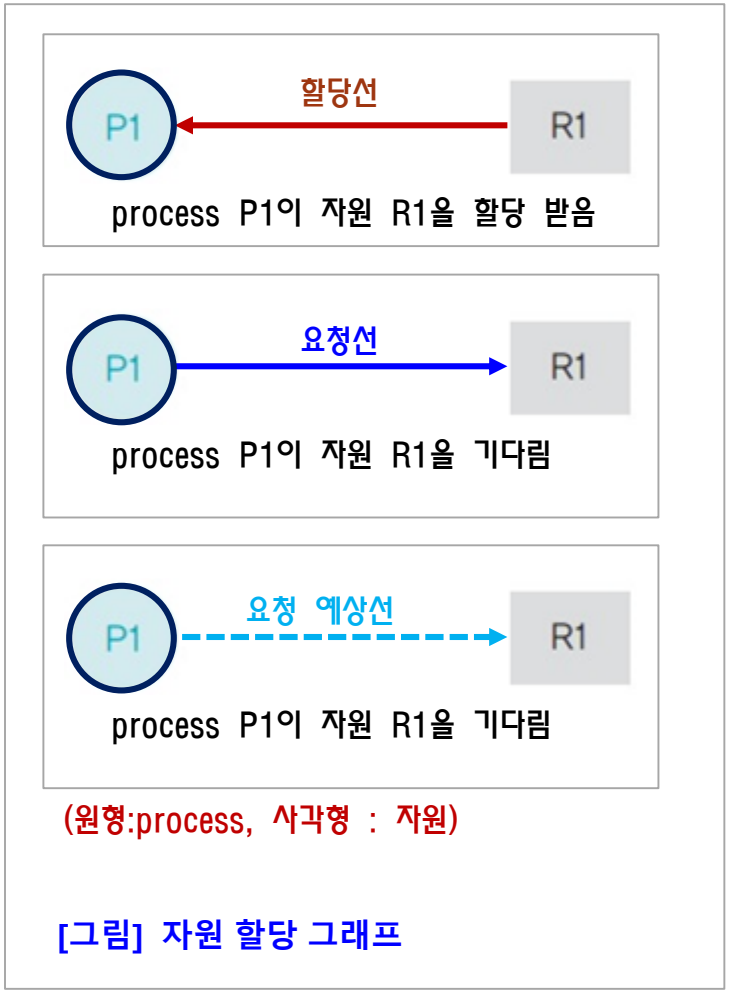
2. Deadlock avoidance (회피)

- deadlock avoidance의 개념
  - process에 자원을 할당할 때 **어느 수준 이상의 자원을 나누어주면 deadlock이 발생하는지 파악**하여 **그 수준 이하로 자원을 나누어주는 방법**
    - process가 생성되어서 소멸 될 때 까지 **최대 얼마 만큼의 자원을 할당 받을 지에 대한 정보를 미리 안다고 가정**
    - **Deadlock이 발생하지 않는 범위 내에서만 자원을 할당**하고, deadlock이 발생하는 범위에 있으면 process를 대기시킴

      → **할당되는 자원의 수를 조절하여 deadlock을 피함**
  - **두 경우의 avoidance 알고리즘**
    - Single instance per resource types – Resource allocation graph algorithm 이용 (또는 Banker's algorithm 이용)
    - Multiple instance per resource types – Banker's algorithm 이용

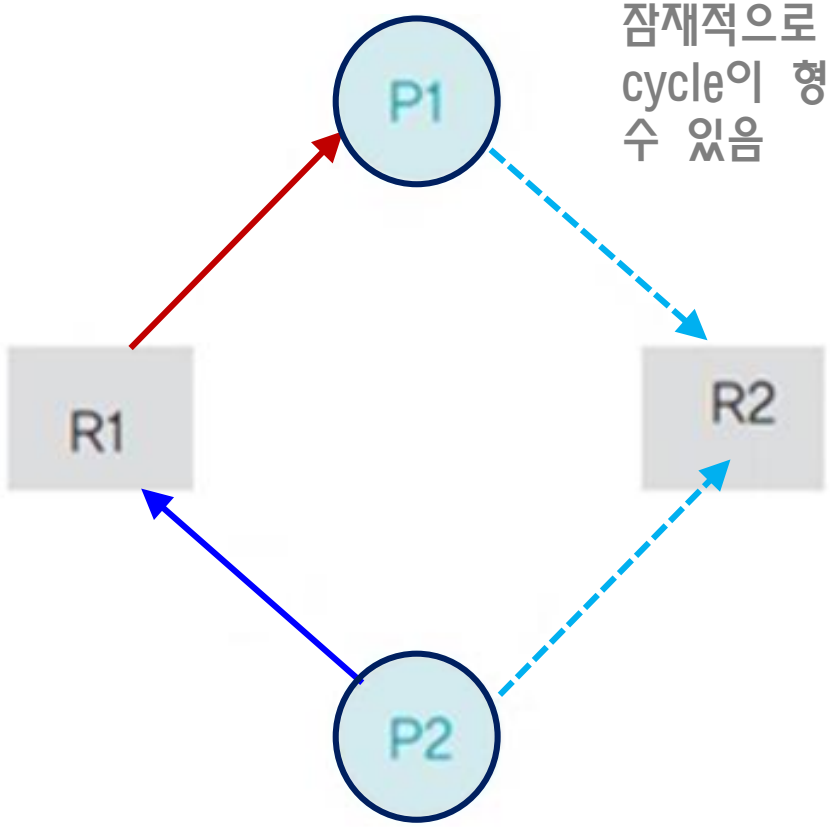
2. Deadlock avoidance (회피) 자원할당그래프 이용

- Resource Allocation Graph Algorithm(자원할당그래프) 이용(single instance)
  - 요청 예상선을 포함하여 cycle이 생기면 자원을 할당하지 않음



[그림] 자원 할당 그래프

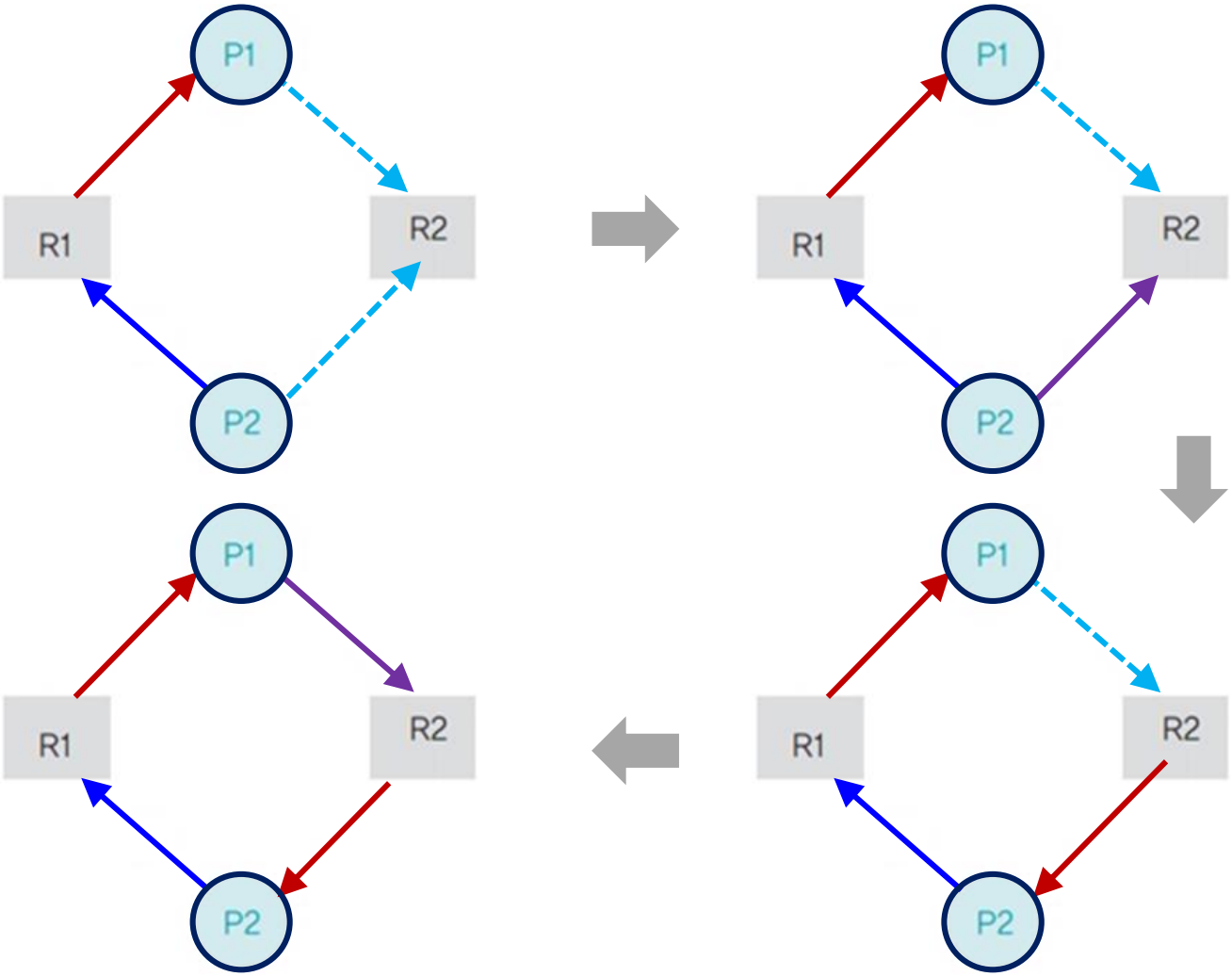
R2자원을 P2에 할당 할 것인가? No



2. Deadlock avoidance (회피) 자원할당그래프 이용

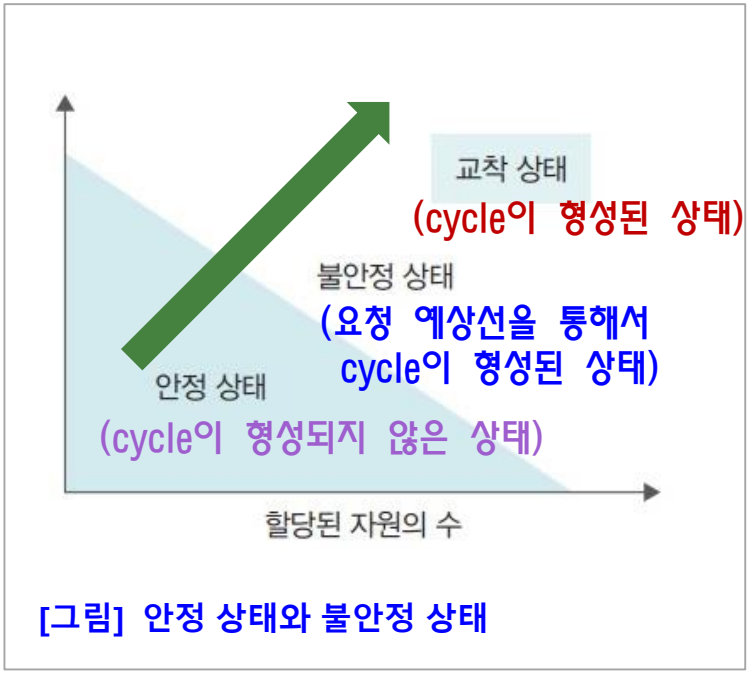
- Resource Allocation Graph Algorithm(자원할당그래프) 이용

R2자원을 P2에 할당 할 것인가? No



2. Deadlock avoidance (회피)

- **안정 상태와 불안정 상태**
  - deadlock avoidance는 자원의 총수와 현재 할당된 자원의 수를 기준으로 시스템을 **안정 상태**safe state와 **불안정 상태**unsafe state로 나누고 **시스템이 안정 상태를 유지하도록 자원을 할당**
    (deadlock은 불안정 상태의 일부)
  - **할당된 자원이 적을 수록 안정 상태가 됨**
    (할당된 자원이 늘어날수록 불안정 상태)
  - Deadlock은 불안정 상태의 일부분이며, **불안정 상태가 커질수록 deadlock이 발생할 가능성이 높아짐**



[그림] 안정 상태와 불안정 상태

  - deadlock avoidance는 **안정 상태를 유지** 할 수 있는 범위 내에서 **자원을 할당**함 으로써 deadlock을 피함

2. Deadlock avoidance (회피) 은행원 알고리즘 이용

- **Banker’s algorithm(은행원 알고리즘) 이용** (Multiple instance)
  - deadlock avoidance를 구현하는 대표적인 알고리즘
  - 은행이 대출을 해주는 방식, 즉 대출 금액이 대출 가능한 범위 내이면(안정 상태이면) 허용되지만 그렇지 않으면 거부되는 것과 유사한 방식
- **은행원 알고리즘의 변수**
  - **전체 자원(total)** : 시스템 내 전체 자원의 수 (각 자원의 총 수)
  - **할당 자원(Allocation)** : 각 process에 현재 할당된 각 자원의 수
  - **최대 자원(Max)** : 각 process가 선언한 최대 자원의 수
  - **기대 자원(Expect)** : 각 process가 앞으로 사용할 자원의 수

    기대자원 = 최대자원 - 할당자원
  - **가용 자원(Available)** : 시스템 내 현재 사용할 수 있는 자원의 수

    가용자원 = 전체자원 - 모든 process의 할당자원

2. Deadlock avoidance (회피) 은행원 알고리즘 이용

- **은행원 알고리즘에서 자원 할당 기준**
  - 각 process의 **기대자원과 비교하여 가용자원이 하나 이상이면 자원을 할당** (크거나 같으면)
  - 가용 자원이 어떤 기대 자원보다 크지 않으면 할당하지 않음

- **안정상태 예** (자원은 1종류, multiple instance)

Total=14 Available= 2 → 6 → 8 → 14

Process	Max	Allocation	Expect
P1	5	2	3
P2	6	4	2
P3	10	6	4

[그림] 은행원 알고리즘(안정 상태, multiple instance)

- **불안정상태 예** (자원은 1종류, multiple instance)

Total=14 Available=1

Process	Max	Allocation	Expect
P1	7	3	4
P2	6	4	2
P3	10	6	4

[그림] 은행원 알고리즘(불안정 상태, multiple instance)

* **안정상태**: 각 process의 기대 자원과 비교하여 가용 자원이 크거나 같은 경우가 한 번 이상인 경우

2. Deadlock avoidance (회피) 은행원 알고리즘 이용

- multiple instance 예제 (시간 t_0에서의 snapshot)
  - 5개의 process(P_0, P_1, P_2, P_3, P_4),
    3개의 자원 종류 및 instance 수 (A-10개, B-5개, C-7개)

전체 자원(total)

* Max로 요청 했을 때, available자원으로 처리가 가능 한가를 따져 보고 자원 요청의 허락 여부 결정

프로세스	Allocation	Max	Expect	Available
	ABC	ABC	ABC	ABC
P_0	010	753	743	332
P_1	200	322	122	532
P_2	302	902	600	743
P_3	211	222	011	745
P_4	002	433	431	755 → 1057
할당량	725			

[그림] 시간 t_0일 때 시스템의 상태

$P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0 \rightarrow P_2$ 이 존재 : safe

2. Deadlock avoidance (회피)

- deadlock avoidance의 문제점
  - process가 자신이 **사용할 모든 자원을 미리 선언해야 함**
  - 시스템의 **전체 자원 수가 고정적이어야 함**
  - 여유자원이 있어도 할당하지 않음 → **자원이 낭비됨**

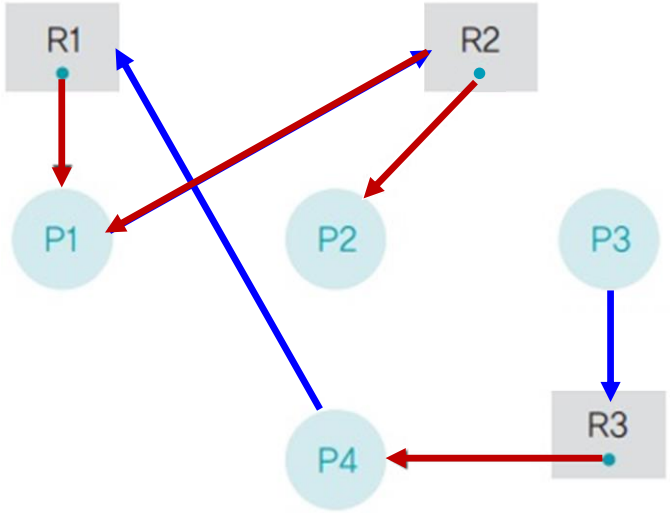
3. Deadlock detection(검출)과 recovery(회복)

- **deadlock 해결 방법**
  - deadlock detection(검출)
    - 운영체제는 **여유자원이 있으면** 무조건 process에게 할당
    - process의 작업을 관찰하면서 **deadlock 발생 여부를 계속 확인하여 검출**
  - Recovery(회복)
    - **Deadlock이 발견**되면 이를 해결하기 위해 **deadlock 회복 단계를 밟음**

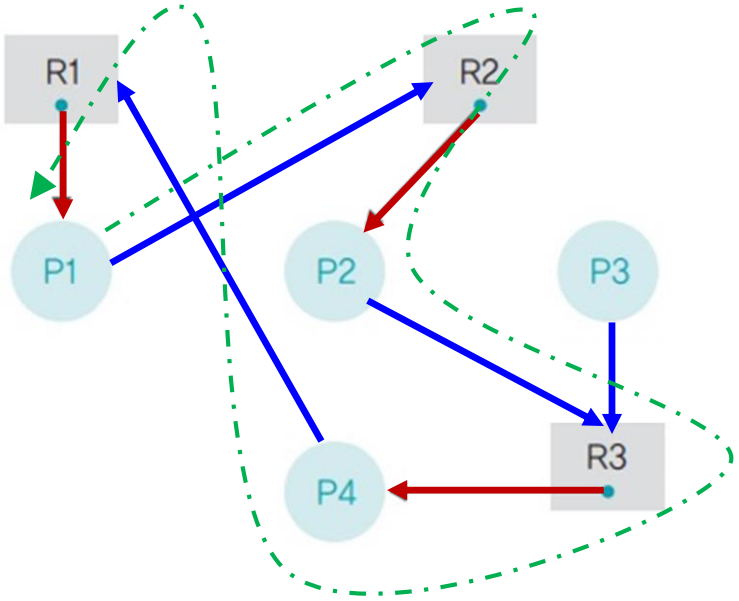
deadlock avoidance에서의 최대 자원(Max)은 고려하지 않음

3. Deadlock detection(검출)과 recovery(회복)

- **single instance 예제 - 자원 할당 그래프를 이용한 deadlock 검출**
  - single instance(단일자원)의 경우 **자원 할당 그래프에 사이클 있으면 deadlock**



(a) deadlock 없음



(b) deadlock detection

[그림] 자원 할당 그래프와 deadlock

3. Deadlock detection(검출)과 recovery(회복)

- multiple instance 예제 (시간 t_0에서의 snapshot)
  - 5개의 process(P_0, P_1, P_2, P_3, P_4),
    3개의 자원 종류 및 instance 수 (A-7개, B-2개, C-6개)

전체 자원(total)

* 요청이 없는 process는 자원을 반납할 것이라고 판단하고, 각 process들이 모든 수행을 마칠 수 있는지 확인

프로세스	Allocation	Request	Available
	ABC	ABC	ABC
P_0	010	000	000 → 010 → 313 → 524 → 526 → 726
P_1	200	202	
P_2	303	000	
P_3	211	100	
P_4	002	002	

자원이 모두 할당 되어 있음

[그림] 시간 t_0일 때 시스템의 상태

$P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1$ 이 존재 : safe

3. Deadlock detection(검출)과 recovery(회복)

- multiple instance 예제 (시간 t_0에서의 snapshot)
  - 5개의 process(P_0, P_1, P_2, P_3, P_4),
    3개의 자원 종류 및 instance 수 (A-7개, B-2개, C-6개)

프로세스	Allocation	Request	Available
	ABC	ABC	ABC
P_0	010	000	000 ↓ 010
P_1	200	202	
P_2	303	001	→ deadlock
P_3	211	100	
P_4	002	002	

[그림] 시간 t_0일 때 시스템의 상태

3. Deadlock detection(검출)과 recovery(회복)

- **deadlock recovery**
  - deadlock이 검출된 후 deadlock을 푸는 후속 작업을 하는 것
  - deadlock 회복 단계

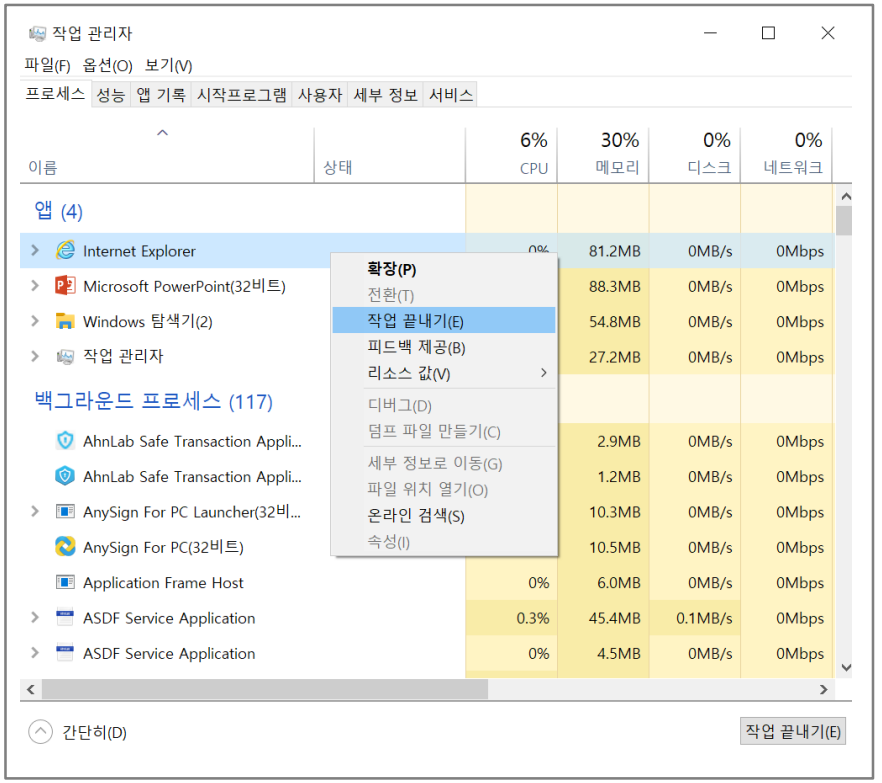
1. **process termination** : deadlock을 유발한 process를 강제로 종료
   - ❶ Deadlock을 일으킨 **모든 process를 동시에 종료**
   - ❷ Deadlock을 일으킨 **process 중 하나를 골라 순서대로 종료**
     (우선순위에 따라 우선순위가 낮은 process를 먼저 종료시킴)

2. **Resource Preemption** : deadlock을 관계된 process들 중 하나에서 자원을 강제로 빼앗는 방법
   - **cost를 최소화 시킬 수 있는 process**가 희생하도록 선택함
     (최소의 자원을 내놓아서 deadlock을 회복시킬 수 있는 process)
   - 희생한 process가 계속적인 deadlock이 생길 때, 해당 process는 starvation 가능성이 생김

     → 희생 횟수를 고려해서 resource를 뺏어야 함

4. Deadlock ignorance(교착 상태 무시)

- deadlock ignorance
  - deadlock이 발생해도 그 어떤 조치도 취하지 않음
    - deadlock은 자주 발생하지 않음
    - deadlock 회복처리는 **시스템에 overload를 발생** 시킴
  - 현대 대부분의(유명한, 상용화된) OS에서는 deadlock을 무시함 (사용자가 직접 deadlock을 처리)



감사합니다.