VII. 교착상태 (Deadlock)

Topics

1. 개요
2. 교착상태 **예방 Prevention)** 자원 **요청 방법을 제한**하여 교착상태 원천 봉쇄
3. 교착상태 **회피 Avoidance)** 자원 **요청시 교착상태 가능성 검사;** 허용 또는 불허
4. 교착상태 **탐지 Detection)** 주기적으로 교착상태 발생 여부를 검사함
5. 교착상태 **회복 Recovery)** 발생한 교착상태를 제거함

1. 개요

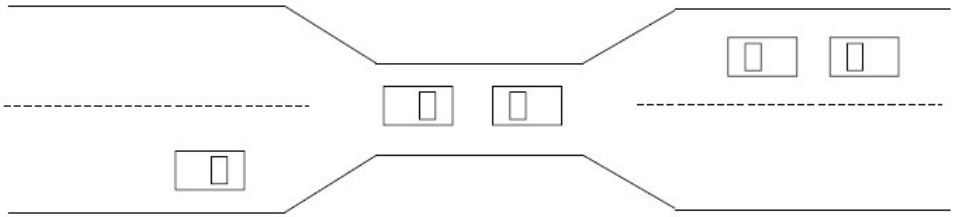
❑ 시스템 모델

유한 개수의 공유 자원	Process가 자원을 사용하는 규약
- 각 자원은 타입을 가짐: R_1, R_2, ..., R_n (예) CPU cycle, Memory space, I/O device, ... - 각 타입의 자원은 다수의 동등한 인스턴스 W_i로 구성됨	- Request (요청) 즉시 허용되지 않으면 대기함 - Use (사용) - Release (방출)

❑ 교착 상태 (Deadlock)

Bridge crossing에서의 deadlock:

어떤 차도 더 이상 진행할 수 없음



(Qt) 시스템은 두 개의 **disk drive D1, D2**를 가짐.

현재 **P1**은 **D1**을, **P2**는 **D2**를 보유한 상태.

이때, **P1**이 **D2**를, **P2**가 **D1**을 요청하면 어떤 상황이 발생하는가?

단, **process**는 이미 획득한 자원을 보유한 채로 **blocked** 상태가 됨.

- blocked process의 집합. 이들은 할당받은 자원을 여전히 소유하고 있음.
- 각 blocked process는 집합 내의 다른 프로세스가 소유한 자원을 요구함.

⇨ 어떤 프로세스도 자신이 원하는 자원을 획득할 수 없음.

Process temporarily give up이 없음

⇨ 집합 내 모든 프로세스는 영원히 "대기" 상태에 머묾.

⇨ 집합 내의 프로세스들이 **deadlock** 상태에 빠짐.

❑ Deadlock(교착 상태 발생 조건)-다음 4가지 필요조건이 동시에 성립할 때 발생.

- **Mutual exclusion 상호 배제**
  하나의 자원은 오직 한 process만이 사용 가능.

- **Hold and wait 점유한 채 대기**
  프로세스는 자신의 자원(s)을 보유한 채로,
  다른 프로세스가 보유 중인 자원을 획득하기 위해 기다림.

- **No preemption of resources**
  보유 자원은 일시적으로 방출될 수 없음.
  (자원은 소유자 프로세스가 사용 완료 후 자발적으로 방출함)

- **Circular wait 순환 대기**
  P_0는 P_1이 보유중인 자원 요청(대기),
  마찬가지로, P_1은 P_2, ..., P_{n-1}은 P_n, P_n은 P_0가 보유중인 자원 요청.

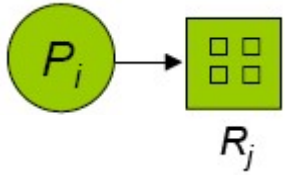
(note) 이 4가지 조건 중 하나라도 깨지면 deadlock은 발생하지 않음.

❑ 자원 할당 그래프 (Resource Allocation Graph)

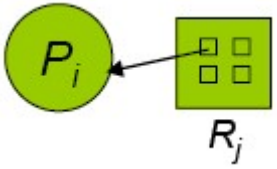
- **Process**

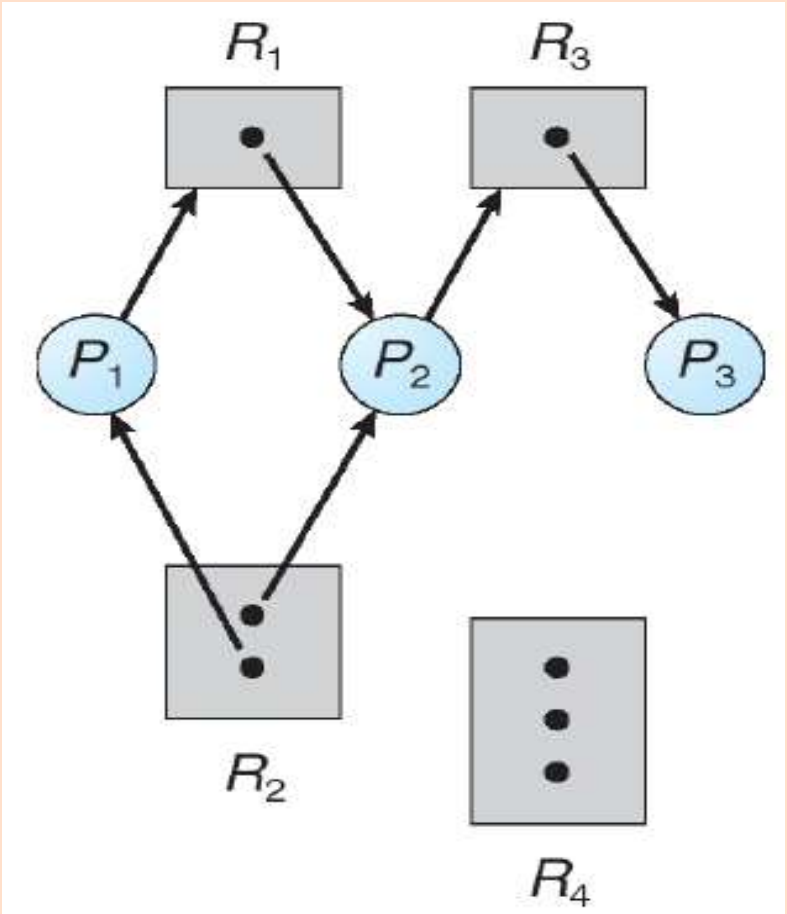
- **Resource type** with 4 instances

- **Request edge** (요청 간선)



- **Assignment edge** (할당 간선)





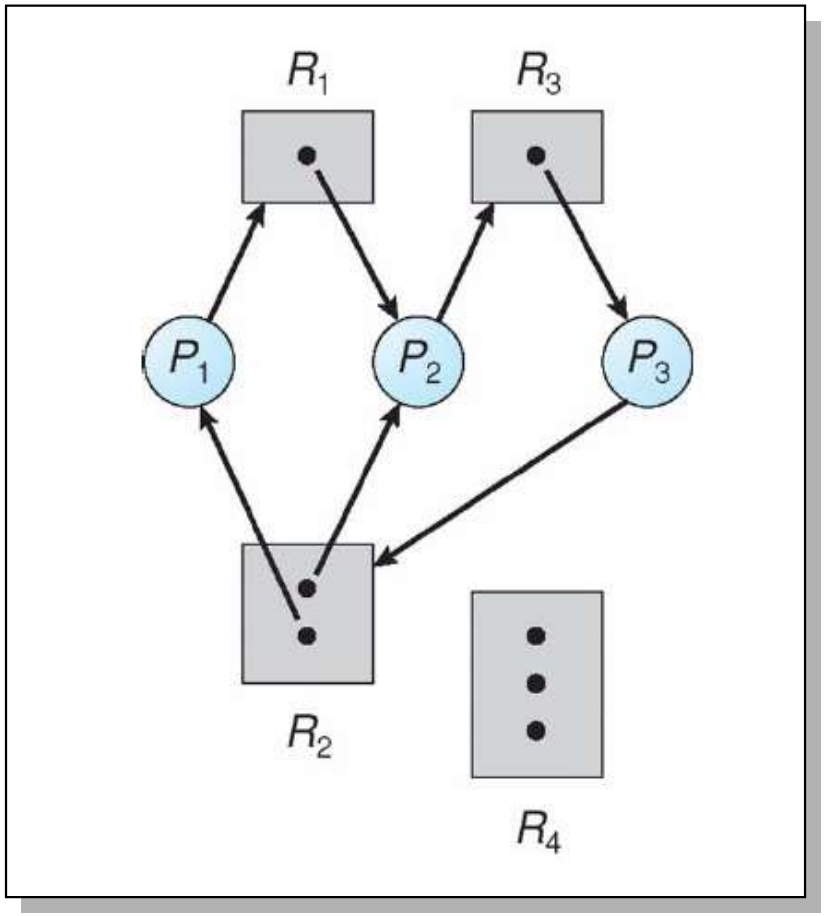
Basic fact 1) **No cycles - No deadlock.**

Cycle 있는 경우:

Basic fact 2) If **one** instance per resource type, then deadlock

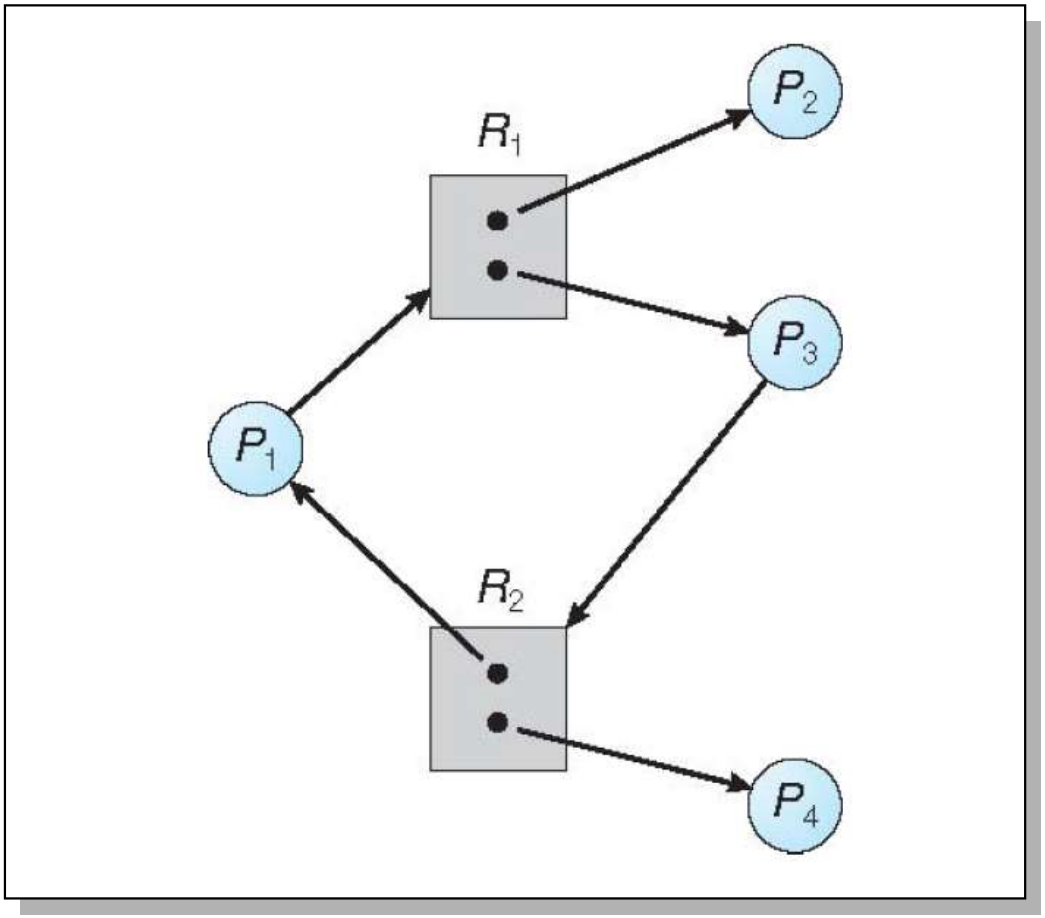
Basic fact 3) If **several** instances per resource type, then possibility of deadlock

(예) Deadlock



(예) A cycle but NO deadlock

- P2나 P4 종료 후 자원이 반환됨.



❑ 교착 상태 처리 방법

- 시스템이 **deadlock** 상태에 들어가지 않도록 보장함: **Prevention, Avoidance**

  - **예방(Prevention)** 요청 방법을 제한하여 **deadlock** 발생을 미연에 방지함.
    즉, 4가지 조건 중 적어도 하나가 성립하지 않도록 보장함.

  - **회피(Avoidance)** 프로세스의 향후 자원 요청 정보를 바탕으로
    요청 時 허용/불허(대기)를 결정함.

- **회복 Recovery**

  교착상태 발생 후 이를 탐지(Detection)하고 회복함(Recovery)

- **무시 Ignorance**
  - 결국 시스템이 정지하면 수작업으로 **Restart**
  - **UNIX를 포함한 대부분의 OS가 취하는 방법**
    ※ 교착상태가 아주 드물게 발생하므로 오히려 ***cost-effective***

2. 교착 상태 예방 Deadlock Prevention

요청 방법을 제한하여 4가지 필요조건 중
적어도 하나가 성립하지 않도록 보장함

(1) Mutual exclusion ⇨ 자원의 동시 접근을 허용함

어떤 자원은 근본적으로 공유가 불가능하므로 예방책이 되지 못함.

- ***Non-sharable resource*** (예) 프린터 (printer)
- ***Sharable resource*** (예) 읽기 전용 파일 (read-only files)

(2) Hold and wait ⇨ 프로세스가 자원을 보유하지 않을 때만 요청하도록 함

(방법 1) 프로세스는 실행 前 모든 자원을 일괄 요청하고 할당받아야 함

(방법 2) 자원을 보유하지 않을 때만 자원을 요청할 수 있음
(먼저 보유 자원을 방출하고, 추가 요청을 함)

(예) ① DVD drive에서 Disk file로 복사 ② disk file 정렬 ③ printer에 출력

(방법 1) 실행 전 {DVD drive, disk file, printer} 일괄 요청

(방법 2) 차례로 요청

먼저 {DVD drive, disk file} 요청 → (복사 및 정렬) → 방출

연후에 {disk file, printer} 요청 → (복사 및 출력)

(note) Starvation 발생 가능

(3) No preemption of resources ⇨ 요청한 자원 획득 불가 時 보유 자원을 일시적 방출

❶ 요청한 자원을 즉시 할당받을 수 없으면,
프로세스는 현재 보유중인 자원을 방출하고 대기함

❷ preempted resources는 (요청한 프로세스의) 자원 대기 리스트에 추가함

❸ preempted resources, 새로 요청한 자원 모두 획득 가능時 프로세스 재계

(note)

- 상태 보존/환원이 가능한 자원에 적용 가능
  (예) CPU, register, memory space

- Printer, Tape drive 등에는 적용 **불가능**

(4) Circular wait ⇨ 순환 대기가 발생하지 않도록 함

- **모든 자원 유형에게 total ordering 부여하고, 오름차순으로만 요청함.**
  (Note) 순서를 정하는 자체만으로 예방되지는 않음.
  프로그램이 순서를 지켜 요청해야 함.

- Lock-Order Verifier
  (예) BSD UNIX의 "witness" 프로그램:
  lock-order 관계를 동적으로 유지하면서 검증함.
  (예) P1이 mutex1, mutex2 순서로 lock 획득하였다고 가정.
  P2가 mutex2를 먼저 획득하려고 시도하면 경고메시지 출력함.

❑ 예방의 단점

자원 사용을 제한하므로 Resource Utilization⬇, System Throughput⬇

3. 교착상태 회피 Deadlock Avoidance

자원을 요청할 때마다 자원할당 상태를 검사하여
교착상태가 없는 안전한 상태가 보장되면 자원을 할당함;
그렇지 않으면 대기함.

- **자원할당 상태**: {가용 자원 개수, 할당된 자원 개수, **사전 정보**}
  - 사전 정보 예) 각 프로세스가 요청할 자원별 **최대 개수**

- **안전한 상태** ***safe state***

  모든 프로세스에 대해,
  교착상태를 유발하지 않는 자원 할당 순서가 존재하는 상태.

  (note) safe sequence(안전한 순서)가 존재하면 시스템은 Safe State에 있음.

❑ 안전한 순서 Safe Sequence

현재 자원할당 상태에서 다음 조건이 성립하면

<P_1, P_2, ..., P_{i-1}, P_i, ..., P_n>은 safe sequence이다:

프로세스 P_i의 향후 자원 요청은

(현재 가용자원 개수 + Pi 앞에 있는 모든 프로세스가 보유한 자원 총수)

에 의해 만족됨.

- 만약 요청 자원이 가용하지 않으면 Pi는 $P_{j(j<i)}$가 종료할 때까지 대기하고
- Pj 종료 時, Pi는 필요한 자원 획득/실행/반환/**종료**함.
- Pi종료 時, P_{i+1}은 필요한 자원 획득 가능함, ...

Basic fact 1) ***Safe*** **state** ⇨ ***No*** **deadlock**

Basic fact 2) ***Unsafe*** **state** ⇨ ***Possibility*** **of deadlock**

Basic fact 3) ***Avoidance*** ⇨ 시스템이 **unsafe state**가 되지 않음을 보장함

(예) 시스템은 12개 Tape를 가짐.

	최대 소요량 (사전 정보)	현재 보유수	향후 자원 요청	할당 순서		
				P1	P0	P2
P0	10	5	5		+5	
P1	4	2	2	+2		
P2	9	2	7			+7
		현재 남은 수	③	⑤	⑩	종료 시 남은 수

⇨ 교착상태를 유발하지 않는 safe sequence가 존재하므로 시스템은 safe state.

	최대 소요량	현재 보유수	향후 요청	할당 순서	
				P1	P2 or P3
P0	10	5	5		할당 불가
P1	4	2	2	+2	
P2	9	3	6		할당 불가
		현재 남은 수	②	④	종료 시 남은 수

⇨ P1 종료 시, 가용한 자원이 ④개, 더 이상 자원 할당 불가능.

⇨ 시스템은 unsafe ⇨ deadlock 발생 (가능).

❑ 교착상태 회피 알고리즘

(1) 자원 할당 그래프 알고리즘 Resource-Allocation Graph Algorithm

각 자원 타입이 오직 하나의 **Instance**를 가지는 경우 사용.

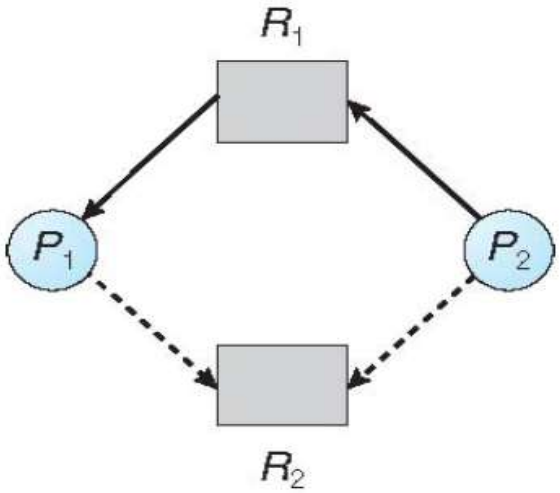
(2) 은행원 알고리즘 Banker's Algorithm

각 자원 타입이 여러 개 **Instance**를 가지는 경우 사용.

❑ 자원 할당 그래프 알고리즘 Resource-Allocation Graph Algorithm

- **예약 간선 Claim Edge** - *dashed arrow*) ***future request***를 나타냄.
  [프로세스 실행 前] 모든 예약 간선을 표시함
  [자원 요청 時] 예약 간선을 요청 간선으로 변환
  [자원 방출 時] 할당 간선을 예약 간선으로 변환

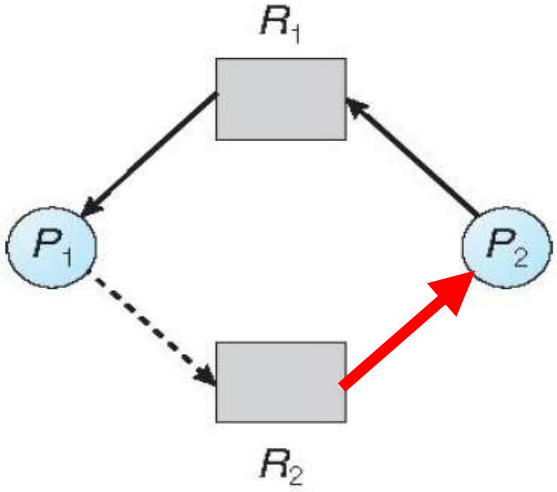
- **프로세스 P_i가 자원 R_j 요청 時, 요청 간선을 할당 간선으로 변환하여도 cycle이 형성되지 않으면 요청을 허용함.**



(R1이 P1에게 할당, P2가 R1 요청 후)
P_2가 자원 R_2 요청 時,

이를 허용하면
cycle 형성하므로

시스템은 ***Unsafe***.



❑ 은행원 알고리즘 Banker's algorithm

- **자원 할당 상태를 나타내는 자료구조**

Available[m]	현재 가용 개수	**Available[i]=k**	현재 R_i 자원 k개 가용
Max[n,m]	최대 요청 개수 **(사전 정보)**	**Max[i,j]=k**	P_i가 최대 k개 R_j 요청 가능
Allocation[n,m]	현재 할당 개수	**Allocation[i,j]=k**	현재 P_i가 k개 R_j 사용 중
Need[n,m]	미래 요청 개수	**Need[i,j]=k**	**향후** P_i가 k개 R_j **요청** 가능

표기법	P, n : 프로세스/개수 R, m : 자원 타입/개수	**X≦Y** : 모든 i에 대해, X[i] ≦ Y[i] 단, X, Y: 크기 n인 벡터. (예) if X=(1,7,3,2) and Y=(0,3,2,1) then X ≧ Y.

• 자원 요청 알고리즘 Resource-Request Algorithm

프로세스 P_i가 자원 요청 時:

$Request_i$: 프로세스 P_i의 요청 벡터.

1. $Request_i$ > $Need_i$ 이면 Error

2. $Request_i$ > Available 이면 Waiting

3. P_i에게 자원 할당을 가정하고, 새로운 자원할당 상태를 계산함:

$$Available = Avaliable - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

4. 새로운 자원할당 상태가 **safe state 인지 검사함** (Safety Algorithm):
   - safe state 이면 Allocation
   - 그렇지 않으면 {자원할당 상태를 환원함; Waiting}

- **안전성 알고리즘 Safety Algorithm** - finds a safe sequence.

```
4.1 Work[m]  Available[m]의 임시 값: Work = Available.
    Finish[n]  프로세스들의 종료 상태:  For all i, Finish[i]=false.

4.2 while (true) {
        // 현재 가용한 자원으로 할당 가능한 프로세스 검색
        find i that satisfies (Finish[i]≡false) and (Needi≦Work);

        // 발견 時, Pi 종료(로 가정) 및 자원 방출
        if (found) {
            Finish[i] = true;
            Work = Work + Allocationi ;
        } else {
            break; // 더 이상 할당 할 수 없는 경우
        }
    }

4.3 모든 i에 대해 Finish[i]≡true 이면 시스템은 안전 상태임.
    (하나라도 finish[i]≡false 이면 시스템은 교착 상태임.)
```

(예) 프로세스: {P0, P1, P2, P3, P4}
자원: {A, B C}, Initially Available[3] = (10, 5, 7)

	Max (사전)			−	t_0	Allocation			=	Need (향후)		
	A	B	C			A	B	C		A	B	C
P0	7	5	3		P0		1			7	4	3
P1	3	2	2		P1	2				1	2	2
P2	9		2		P2	3		2		6		0
P3	2	2	2		P3	2	1	1		0	1	1
P4	4	3	3		P4			2		4	3	1

- t_0에서 시스템은 safe state?
  - [4.1] At t_0 Available = Work = (10-7=3, 5-2=3, 7-5=2).
  - [4.2] 할당 가능한($Need_i \leqq Work$) P_i는?
    <P1, P3, P4, P2, P0> 순서로 할당 가능함.
  - [4.3] 따라서 safe sequence 존재함.
    t_0에서 시스템은 safe state 임.

	Max (사전)				t_1	Allocation				Need (향후)		
	A	B	C			A	B	C		A	B	C
P0	7	5	3		P0		1			7	4	3
P1	3	2	2	-	P1	2→3		0→2	=	0	2	0
P2	9		2		P2	3		2		6		0
P3	2	2	2		P3	2	1	1		0	1	1
P4	4	3	3		P4			2		4	3	1

- t_1에서 P1이 추가 요청: $Request_1 = (1,0,2)$. Can be granted?
  - [3] Available = (2,3,0).
  - [4.1] Work = (2,3,0).
  - [4.2] 할당 가능한($Need_i \leqq Work$) P_i는?
    <P1, P3, P4, P0, P2> 순서로 할당 가능함.
  - [4.3] 따라서 safe sequence 존재함. t_1에서 시스템은 safe state 임.
  - [4] 따라서 할당(가능) 함.

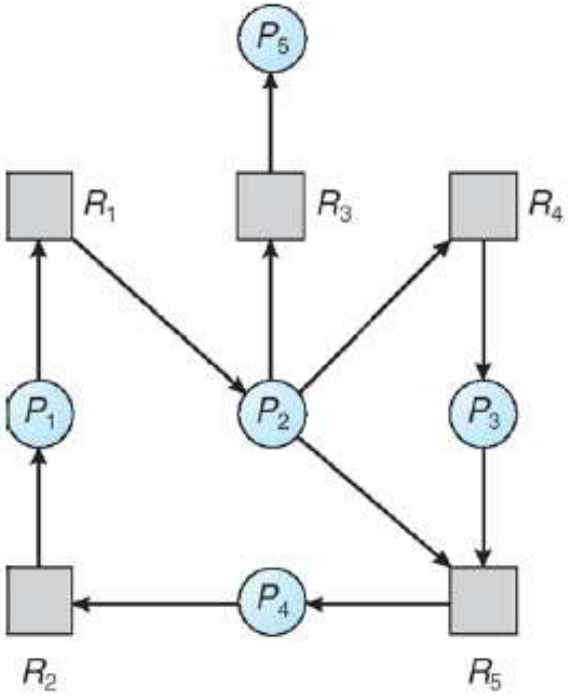
- (Question 1) Can request for (3,3,0) by P4 be granted?
- (Question 2) Can request for (0,2,0) by P0 be granted?

4. 교착상태 탐지 Deadlock Detection

❑ 자원 유형별로 하나의 자원이 있는 경우:

주기적으로 대기 그래프 Wait-For Graph를 검사함.
Cycle이 존재하면 Deadlock이 존재함.

자원할당그래프 RAG



RAG를 대기그래프로 변환:

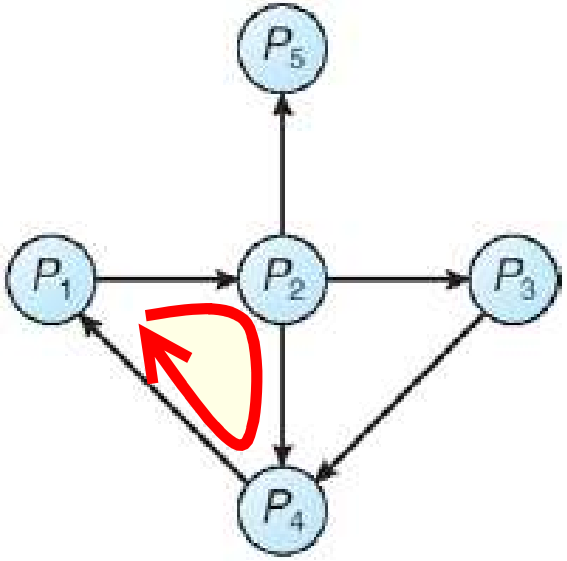
($P_i \rightarrow R_q$ & $R_q \rightarrow P_j$)를

$P_i \rightarrow P_j$로 변환함

- $P_i \rightarrow P_j$

  P_i는 P_j가 보유중인 자원을 기다림

대기 그래프



❑ 자원 유형별로 여러 개의 자원이 있는 경우

Available[m]	현재 가용 개수	**Available[i] = k**	현재 **k**개 자원 R_i 가용
Allocation[n,m]	현재 할당 개수	**Allocation[i,j] = k**	현재 P_i가 **k**개 R_j를 사용 중
Request[n,m]	현재 요청 개수	**Request[i,j] = k**	현재 P_i가 **k**개 R_j를 요청 가능

표기법	**P**, n : 프로세스/개수 **R**, m : 자원 타입/개수	**X≦Y** : 모든 **i**에 대해, **X[i]≦Y[i]** 단, **X**, **Y**: 크기 **n**인 벡터. (예) **if X=(1,7,3,2) and Y=(0,3,2,1) then X ≧ Y.**

- **Detection Algorithm**

1. **Work[m]** Available[m]의 임시 값: **Work = Available.**
   Finish[n] 프로세스들의 종료 상태:
   For all i, if ($Allocation_i$==0) then **Finish[i]=true** else **Finish[i]=false.**
   //현재 자원 사용하지 않는 프로세스는 교착상태와 무관하므로 종료한 것으로 간주함.

2. while (true) { ⇦ Safety Algorithm: Finds a safe sequence
   // (종료되지 않은,) 현재 가용한 자원으로 할당 가능한 프로세스 검색
   find i that satisfies (Finish[i]≡false) and ($Request_i$≦Work);

   // 발견 時, Pi 종료(로 가정[1]) 및 자원 방출
   if (found) {
   Finish[i] = true;
   Work = Work + $Allocation_i$;
   } else {
   break; // 종료되지 않았으나 할당 받지 못하는 프로세스 존재함
   }
   }

3. 하나라도 finish[i]≡false 이면 시스템은 교착 상태.
   (note) break 時 Pi가 deadlock chain을 완성한 마지막 프로세스.

---

1) 현재까지는 Pi는 deadlock과 무관. 더 이상 요구 않고 종료할 것으로 (낙관적으로) 기대.
만약 나중에 deadlock을 유발하면 차후 detection 時 탐지될 것임.

교착상태 탐지 (예 1)

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), 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	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

[2] Po와 P2의 요청을 만족시킬 수 있으므로 차례로 finish 처리 함.

[2] Work = (3,1,3)가 되므로 모든 process가 종료 가능.

교착상태 탐지 (예 2)

- P_2 requests an additional instance of type *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

Work = (0,1,0)

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

❑ 탐지 알고리즘의 사용

- 호출 시점

  - 자원 요청이 허용되지 않는 즉시

    deadlocked processes 뿐만 아니라
    deadlock을 야기한 프로세스도 식별 가능 **(위의 예 2에서 P2)**

  - 탐지 부하를 줄이기 위해 가끔씩 호출

    (예) 한 시간에 한 번, CPU utilization이 40% 이하로 떨어질 때

- 호출 빈도는 교착상태 발생 빈도수와 연루된 프로세스 수에 의존적

5. 교착상태 회복 Deadlock Recovery

❑ 교착상태로부터 회복 (2가지 방법)

- 하나 이상의 deadlocked process를 강제 종료 (*aborting*)
- 하나 이상의 deadlocked process로부터 자원 강제 회수 (*preemption*)

❑ **Process Termination**

- 2가지 방법
  - 모든 deadlocked process abort ⇨ 재실행 비용
  - 교착상태 제거時까지 하나씩 abort ⇨ abort 時마다 탐지알고리즘 실행

- 희생자 프로세스 선정 기준
  - 프로세스 우선순위
  - 사용한 자원 유형 및 수
  - 종료되어야 할 프로세스 수
  - 실행한 총시간, 완료까지 필요한 시간
  - 완료까지 추가로 필요한 자원의 수
  - 프로세스 유형: 대화형, 일괄처리

❑ Resource Preemption (자원을 일시적으로 회수함)

- **Successively preempt some resources from deadlocked processes and give them to other processes until the deadlock cycle is broken**

- 고려 사항
  - 희생자 프로세스, 보유 자원
    ※ 비용 최소화 (보유 자원 수, 실행한 총 시간, 등등)

  - 프로세스 롤백(rollback) 위치
    - ***safe state***로 롤백 시키고 차후에 재시작 시킴
    - ***total rollback***) 프로세스를 강제 종료시키고 재시작 시킴

  - starvation에 대한 고려
    - (예) 롤백 횟수를 비용에 포함시킴