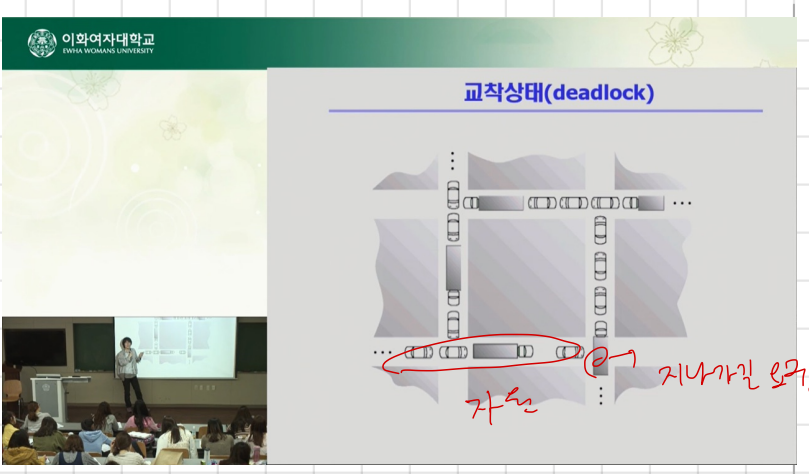
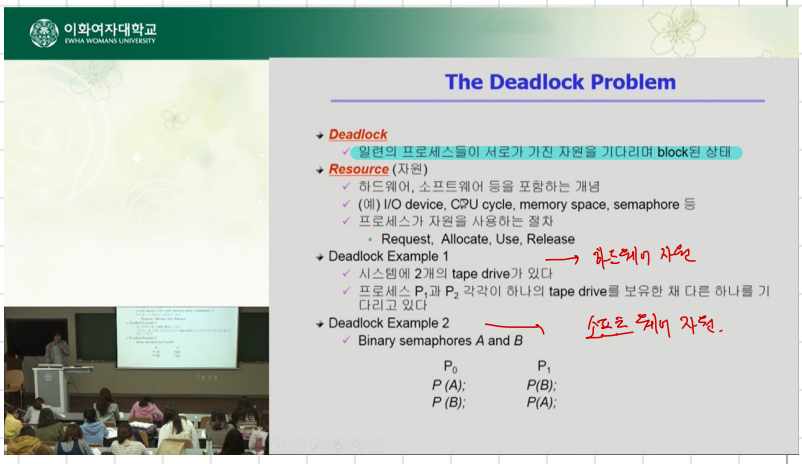
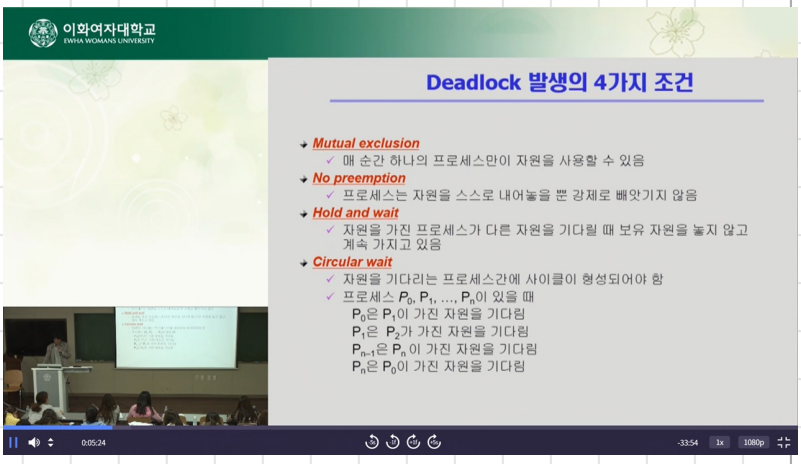
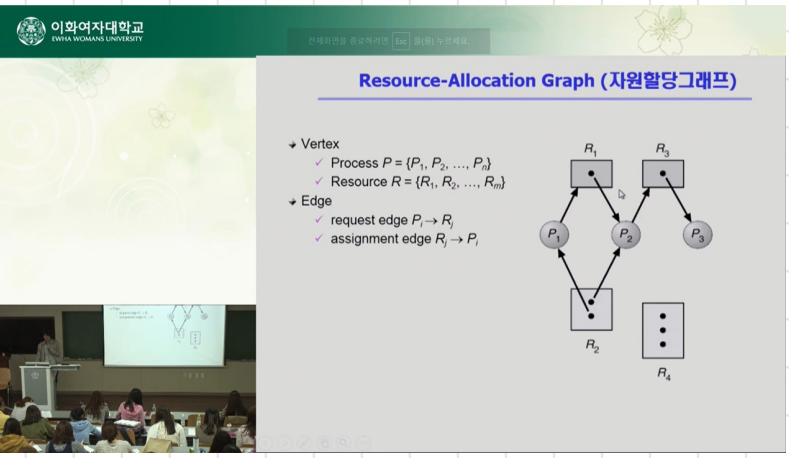






Deadlock 발생 4조건 (모두 만족)

1. 나 혼자만 독점적으로 자원사용.
2. 강제로 빼앗기지 않음. (강제)
3. 가진 자원은 보유하며 추가자원을 기다리므로. (자발)
4. 서로 필요로 하는 자원이 Cycle을 형성하는 경우.



$R_i \rightarrow P_j$ 화살표 : P_j가 자원을 가지고 있다.

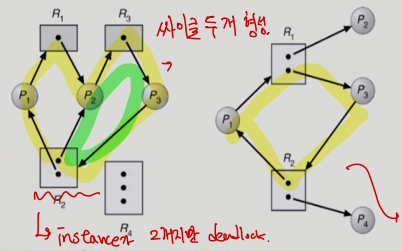
$P_i \rightarrow R_j$ 화살표 : P_i가 자원 요청을 했으나 미획득.

R_i내 작은 점: resource instance 수.

→ 설명. P_1이 R_2 가지고있고 P_1이 R_1 요청, P_2가 R_1가 지고 있고... P_3는 R_3 가짐.

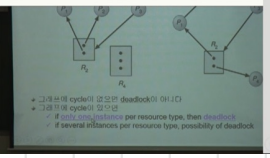
이화여자대학교 EWHA WOMANS UNIVERSITY

Resource-Allocation Graph



사이클 두개 형성

↳ instance 2개씩일 deadlock.



- 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

자원이 하나씩만 있으면 deadlock이어,
P_2, P_4가 자원 사용 후 반납하면 deadlock 해결.

사이클 존재,
but, 자원이 여러개 존재함으로 deadlock 일수 있고, 아닐수 있음.

이화여자대학교 EWHA WOMANS UNIVERSITY

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 detection 루틴을 두어 deadlock 발견시 recover
- ***Deadlock Ignorance***
  - Deadlock을 시스템이 책임지지 않음
  - UNIX를 포함한 대부분의 OS가 채택

Deadlock이 자주 발생하는 것이 아니기 때문에,
미연에 방지하는 오버헤드가 큼.
따라서 Deadlock 발생시 사용자가 처리

Deadlock 미연에 방지

앞의 deadlock 처리방법.

이화여자대학교 EWHA WOMANS UNIVERSITY

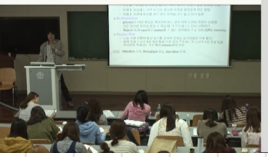
I. Deadlock Prevention

- ***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 자원 비효율
  - 방법 2. 자원이 필요할 경우 보유 자원을 모두 놓고 다시 요청 다 뺄고 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)
- ***Circular Wait***
  - 모든 자원 유형에 할당 순서를 정하여 정해진 순서대로만 자원 할당
  - 예를 들어 순서가 3인 자원 R_i를 보유 중인 프로세스가 순서가 1인 자원 R_j을 할당받기 위해서는 우선 R_i를 release해야 한다

⇨ Utilization 저하, throughput 감소, starvation 문제

현재 사용 상태를 쉽게 Save, restore 가능.
반대로 쓰다가 빼앗으면 오버헤드크면 빼앗지 말기.

누군가 3을 가지고 있는데 1번 달라하면 -.
이럴 경우 deadlock인데 1→5, 5→3 이런 식으로 순서 정하면 deadlock 해결.



이화여자대학교 EWHA WOMANS UNIVERSITY

Deadlock Avoidance

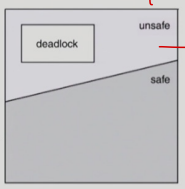
- Deadlock avoidance
  - 자원 요청에 대한 부가정보를 이용해서 자원 할당이 deadlock으로부터 안전(safe)한지를 동적으로 조사해서 안전한 경우에만 할당
  - 가장 단순하고 일반적인 모델은 프로세스들이 필요로 하는 각 자원별 최대 사용량을 미리 선언하도록 하는 방법임
- *safe state*
  - 시스템 내의 프로세스들에 대한 *safe sequence*가 존재하는 상태
- *safe sequence*
  - 프로세스의 sequence $<P_1, P_2, \ldots, P_n>$이 safe하려면 P_i $(1 \le i \le n)$의 자원 요청이 "가용 자원 + 모든 P_j $(j < i)$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_j $(j < i)$가 종료될 때까지 기다린다
    - P_{i-1}이 종료되면 P_i의 자원요청을 만족시켜 수행한다

평행선 자원을 미리 알고 있어서 자원요청시 deadlock 발생할 것 같으면 허용해주지 않음.

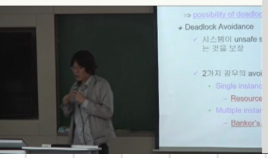
이화여자대학교 EWHA WOMANS UNIVERSITY

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



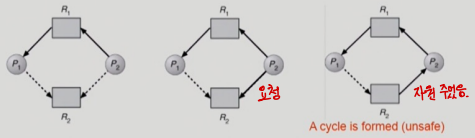
↳ 잘 발생하지 않음.
→ unsafe하다하여 deadlock은 아님.

이화여자대학교 EWHA WOMANS UNIVERSITY

Resource Allocation Graph algorithm

자원당 instance가 하나인 경우.

- *Claim edge* $P_i \rightarrow R_j$
  - ✓ 프로세스 P_i가 자원 R_j를 미래에 요청할 수 있음을 뜻함 (점선으로 표시)
  - ✓ 프로세스가 해당 자원 요청시 request edge로 바뀜 (실선)
  - ✓ R_j가 release되면 assignment edge는 다시 claim edge로 바뀐다
- request edge의 assignment edge 변경시 (점선을 포함하여) cycle이 생기지 않는 경우에만 요청 자원을 할당한다
- Cycle 생성 여부 조사시 프로세스의 수가 n일 때 $O(n^2)$ 시간이 걸린다



P_1이 R_2 자원요청하는 순간, deadlock 발생.

점선 : 프로세스가 자원에 가는 방향만 존재
적어도 한 번은 사용할 일이 있다.
(Avoid 관점이므로 요구사항 미리 파악)
요청하면 점선 → 실선 변경

최악의 상황 가정. deadlock이 있을 것 같으면 안줌. (점선포함 사이클 생성)
P_1의 자원 사용은 허락.

이화여자대학교 EWHA WOMANS UNIVERSITY

Example of Banker's Algorithm

최대요청량을 따져 Processing 자원 줄 지 말지 결정

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

	Allocation	Max	Available	Need (Max – Allocation)
	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

* sequence < P_1, P_3, P_4, P_2, P_0>가 존재하므로 시스템은 safe state

확실한 상황 때문에 매우 비관론적으로 돌아감

Allocation : 현 상황.
Max : 최대 사용 자원.
Available : 사용가능자원 – 할당자원.
Need : 추가 요청 가능량

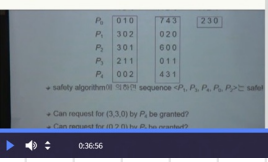
이화여자대학교 EWHA WOMANS UNIVERSITY

P_1 request (1,0,2)

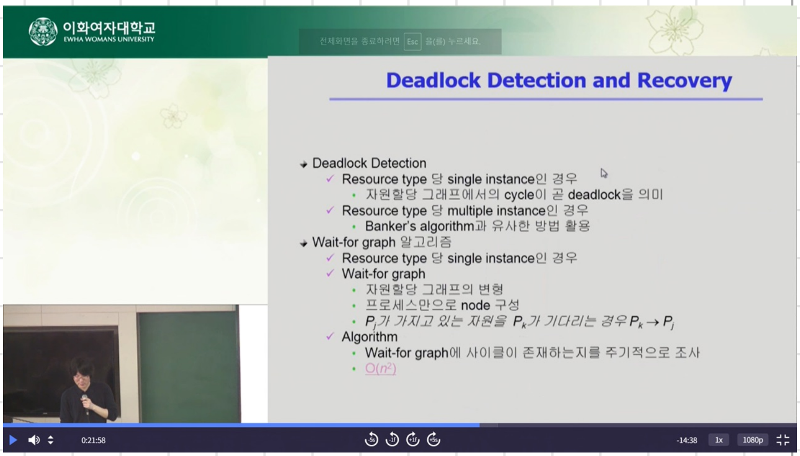
- Check that Need ≤ Available, that is, (1,2,2) ≤ (3,3,2) ⇒ *true*.

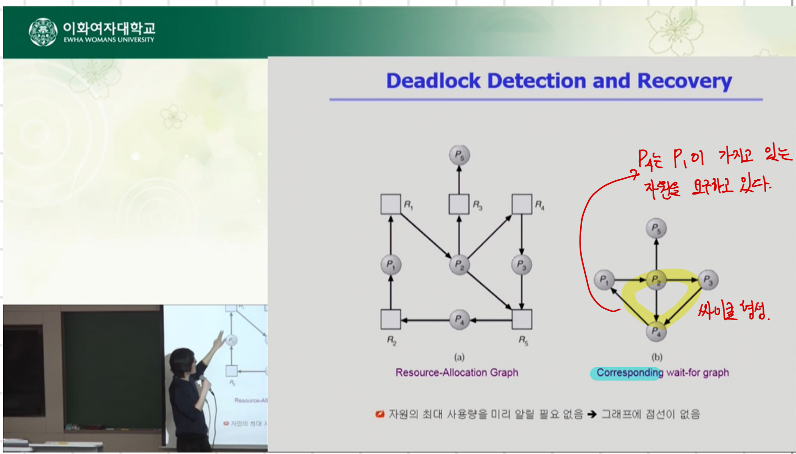
	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

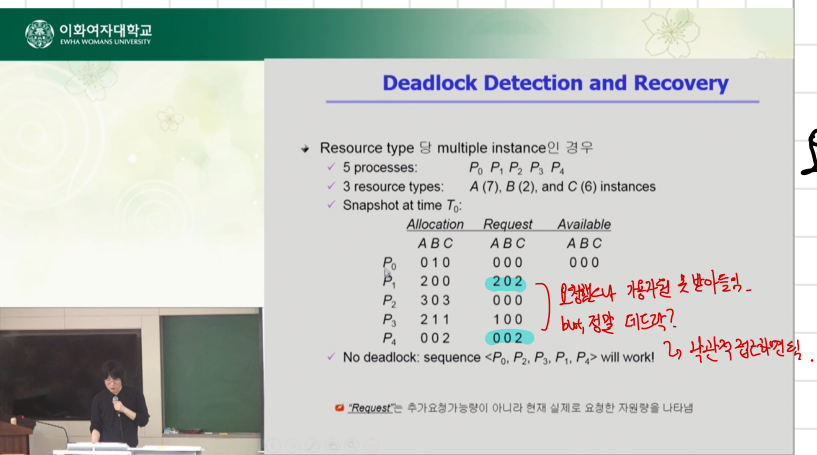
- safety algorithm에 의하면 sequence <P_1, P_3, P_4, P_0, P_2>는 safe!
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?



P_1이 (1,0,2) 요청했다 가정.
Available이 (3,3,2)였으므로 줄 수 있음.
but, P_1이 추가 요청할 수 있는 양이 가용자원으로 모두 충족되는지
(3,3,2) – (3,2,2) = (0,1,0)
↳ 보수적 접근, 몇 개를 요청해도 이것만 봄.
P_0의 경우만 요청하면, 최대요청인 경우 lock걸릴수있음에 안줌. 다른 P_i가 Allocation 뱉어내서 충족되면 준다!







싸이클 확인.

한 Point마다 다른 Point. 선택 → n-1 경우.

총 $(n-1)n = O(n^2)$

(preventer 달리)

낙관적 접근으로 deadlock 판별.

요청하지 않은 프로세스에 대해 반납할 것이다 가정.
P_0, P_2

Available → A→3, B→1, C→3

P_1이 요구한 A 2개, C 2개 요구 ok?

→ P_4 : P_1 반납 후 A→5 B→1 C→3,

P_4 수행? ok.

Deadlock 없는 상황.

If. P_2가 자원 1개 요청하면 Deadlock!

Process 원래 가진 자원을 자발적으로 내놓지 않는다이기 때문에.

이화여자대학교 EWHA WOMANS UNIVERSITY

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

Deadlock이 연루된 모든 프로세스 죽이기.

하나씩 프로세스를 죽이면서 Deadlock이 풀리는지 파악.

자원뺏는 패턴 변화. (Starvation 문제 및 돌일 deadlock.)

방법은 자세히 다루지 않겠음.

이화여자대학교 EWHA WOMANS UNIVERSITY

Deadlock Ignorance

- Deadlock이 일어나지 않는다고 생각하고 아무런 조치도 취하지 않음
  - ✓ Deadlock이 매우 드물게 발생하므로 deadlock에 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

운영체제가 Deadlock 생기든 말든 냅둠

시스템이 느려지거나, 멈출경우 사용자가 알아서 처리.