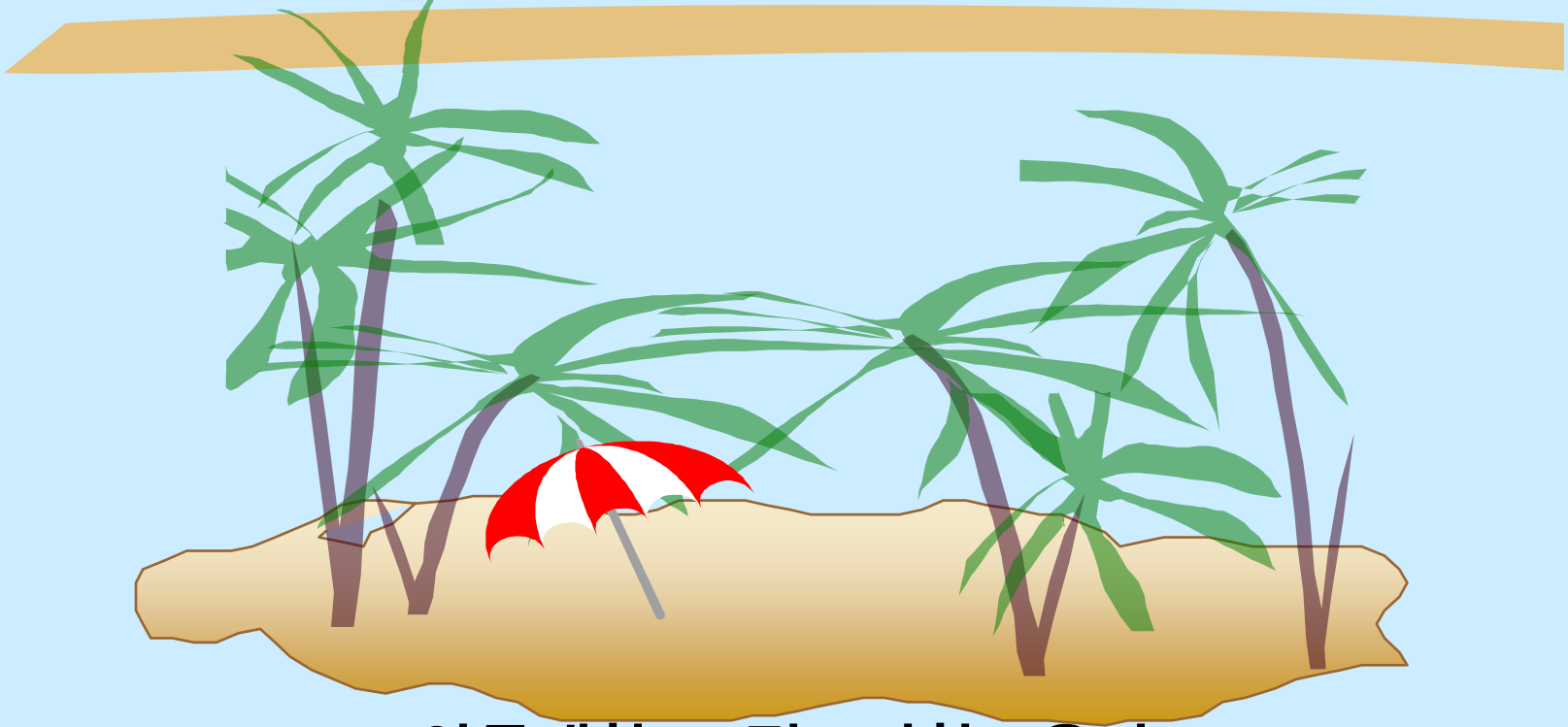
동시성 제어 기법



안동대학교 정보과학교육과

배울 내용

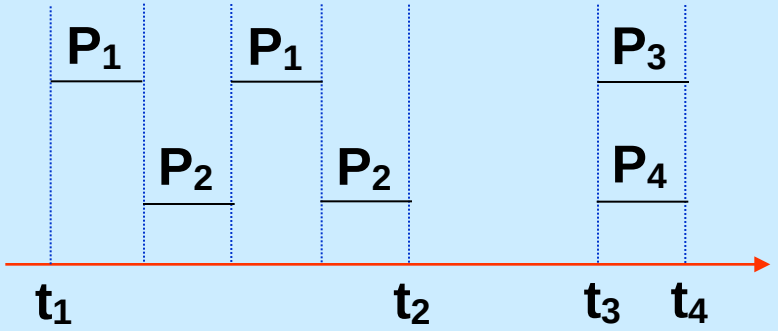
- 동시성 제어
- 로킹 기법
- 시간 스탬프 순서 기법
- 삽입/삭제 연산과 동시성 제어

복수 사용자 DBMS(I)

- 데이터베이스 시스템의 주요 목표
  - 공용성(sharability)
    - 여러 사용자가 데이터베이스를 이용 가능
    - ➔ 복수 사용자 데이터베이스: 병행 (concurrent) 데이터베이스
  - 정확성(accuracy)
    - 공용된 데이터베이스를 정확히 유지
- 동시 공용(Concurrent Sharing)
  - 공용성의 증가
  - 응답 시간의 단축
  - 시스템의 이용 효율성 증대

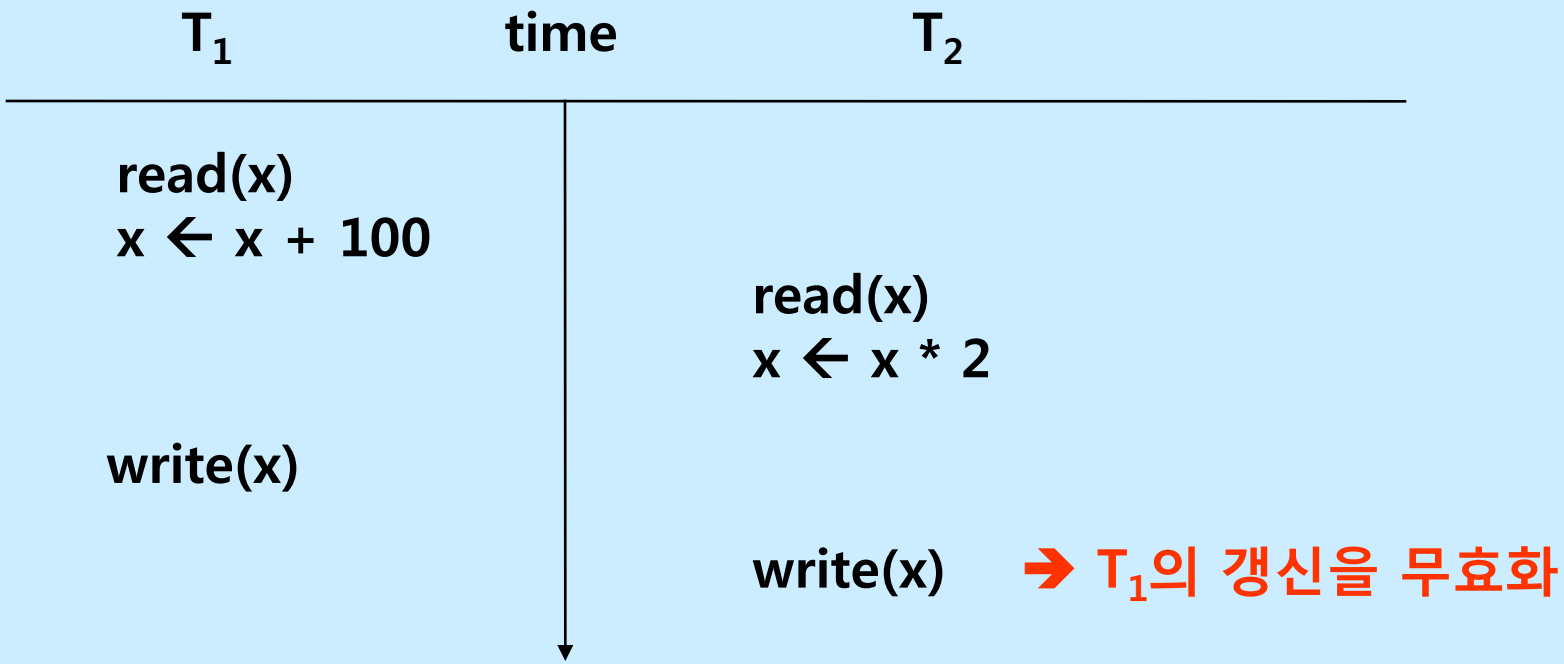
복수 사용자 DBMS(II)

- 병행 데이터베이스 : 다중 사용자 시스템
  - 병행 접근
  - 다중 프로그래밍
    - 인터리브 된(interleaved) 실행
    - 다중처리(multiprocessing)



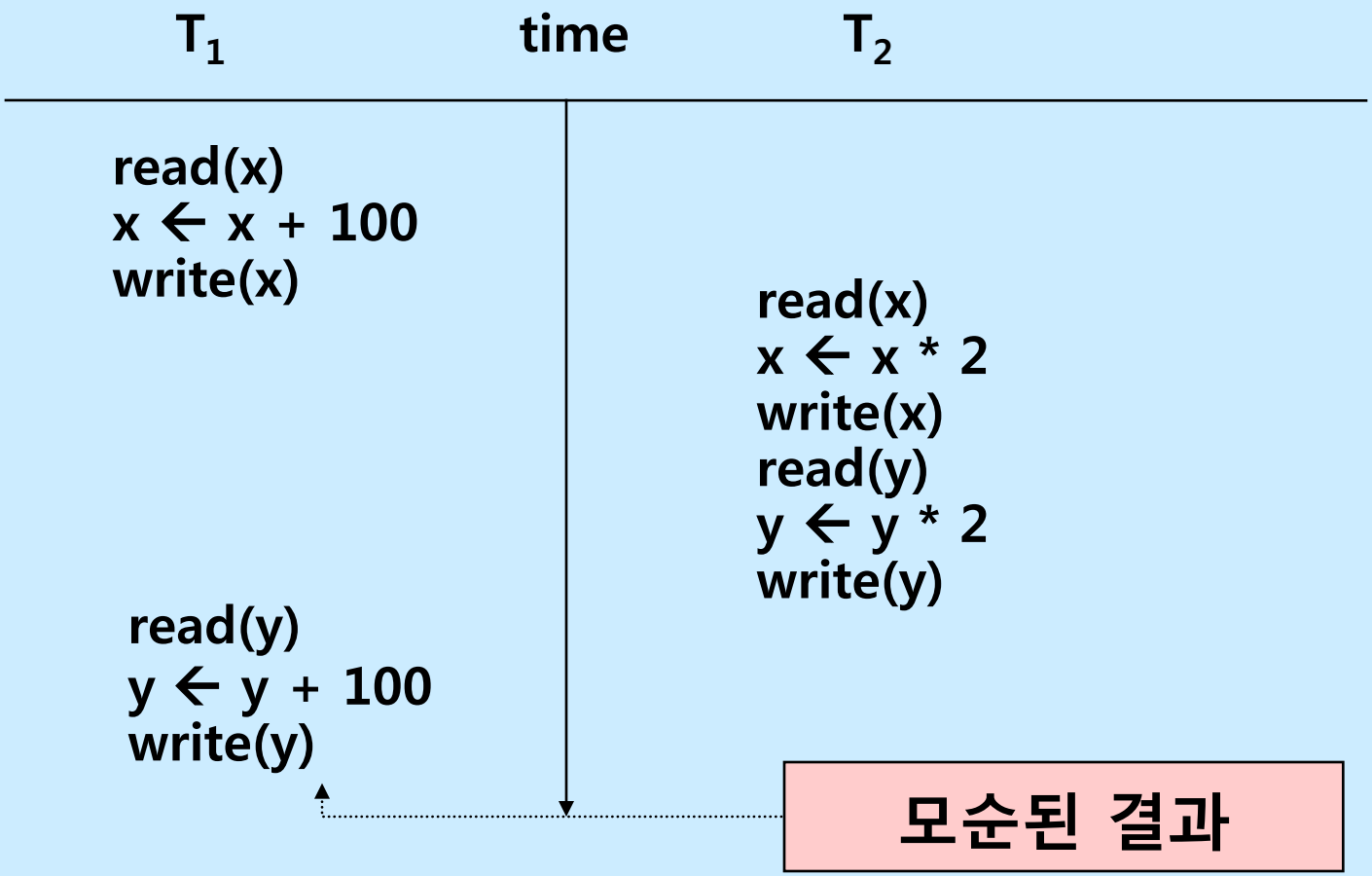
무제어 동시 공용의 문제점(I)

- 갱신 분실(lost update)
  - 탐지 불가능



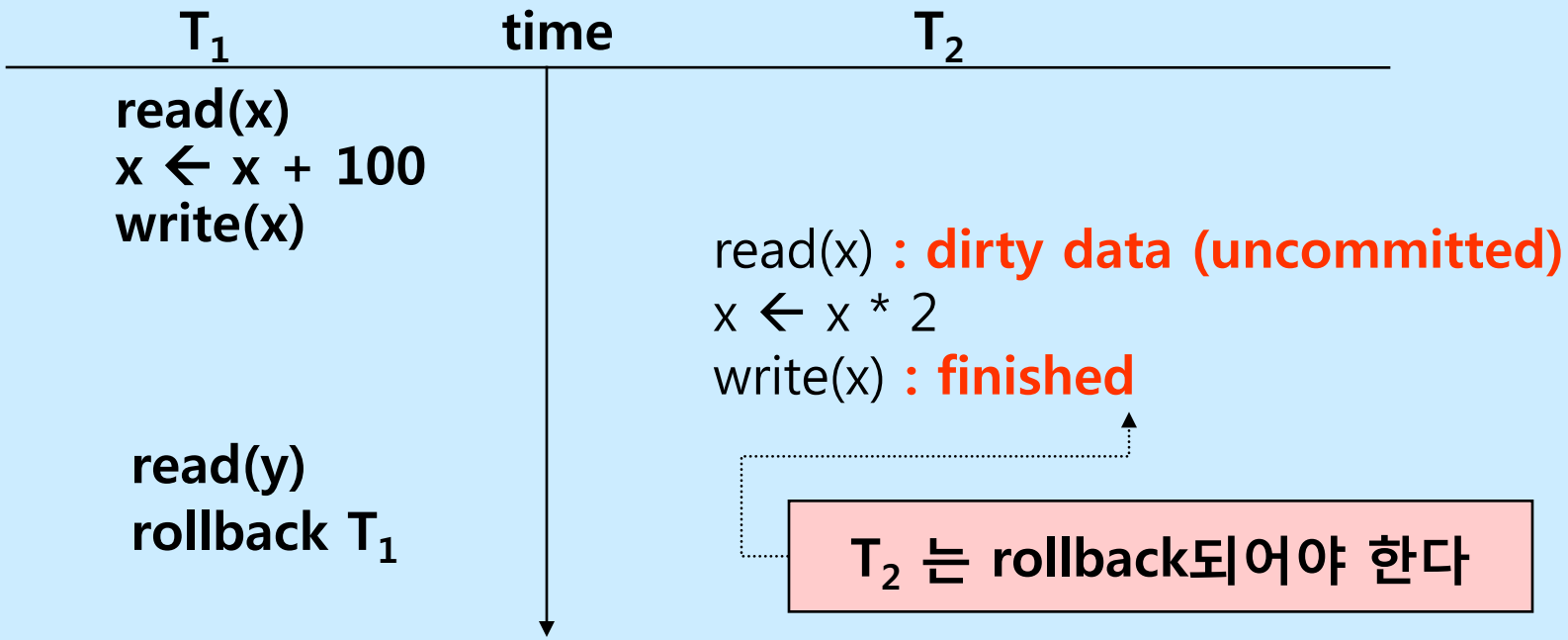
무제어 동시 공용의 문제점(II)

- 모순성(inconsistency)
  - 데이터베이스의 출력내용과 모순



무제어 동시 공용의 문제점(III)

- 연쇄 복귀(cascading rollback)
  - 상호 의존(tangled dependencies)
  - 연쇄 회복(cascade recovery)



- 완료되지 않은 데이타 접근

무제어 동시 공용의 문제점(IV)

- 갱신분실/모순성/연쇄복귀의 원인
  - 공용하는 충돌된 데이터를 통해 트랜잭션 사이에 간섭이 일어나기 때문
- 병행 제어
  - 충돌 데이터의 관리
- 충돌(conflict)
  - 동일한 데이터 객체에 대한 두 연산
  - 적어도 하나는 write 연산
    - $read_i(x)$와 $write_j(x)$
    - $write_i(x)$와 $read_j(x)$
    - $write_i(x)$와 $write_j(x)$

직렬 가능성의 개념 – Review(I)

- 스케줄(Schedule)
  - 실행 순서
  - 트랜잭션 연산들의 순서
- 직렬 스케줄
  - 트랜잭션 $\{T_1, ..., T_n\}$의 순차적 실행
  - 인터리브 되지 않은 스케줄
  - 스케줄의 각 트랜잭션 T_i의 모든 연산 $<T_{i1}, ..., T_{in}>$이 연속적으로 실행 ➔ n! 가지의 방법
  - 직렬 스케줄은 정확하다고 가정
- 비직렬 스케줄
  - 인터리브 된 스케줄
  - 트랜잭션 $\{T_1, ..., T_n\}$의 병렬 실행

직렬 가능성의 개념 – Review(II)

- 직렬 가능한 스케줄

  n개의 트랜잭션 T_1, ..., T_n에 대한 스케줄 S가 동일한 n개의 트랜잭션에 대한 어떤 직렬 스케줄 S'과 동등하면 스케줄 S는 직렬 가능 스케줄

  - 직렬 스케줄 S_1 : <T_1, T_2, T_3>

$$S_1 : \overbrace{< O_{11}, O_{12}, O_{13}, O_{14} >}^{T_1} \overbrace{< O_{21}, O_{22}, O_{23} >}^{T_2} \overbrace{< O_{31}, O_{32} >}^{T_3}$$

  - 비직렬 스케줄 S_2

$$S_2 : < O_{11}, O_{21}, O_{22}, O_{12}, O_{31}, O_{23}, O_{13}, O_{32}, O_{14} >$$

  - S_2가 직렬 스케줄 {T_1, T_2, T_3}과 동등하다면 S_2는 직렬 가능한 스케줄

직렬 가능성의 이용

- 스케줄의 직렬 가능성 검사는 현실적으로 어려움
- 대부분의 시스템에서는

  트랜잭션을 실행시킨 다음,

  스케줄 자체에 대한 직렬 가능성 검사를 하지 않고도

  직렬 가능성이 보장되는 방법을 사용

  → 트랜잭션 작성시 규약을 따르면, 그 트랜잭션이 참여하는 스케줄의 직렬성을 보장

  - 로킹(locking)
  - 시간 스탬프(timestamp)

배울 내용

- 동시성 제어

- 로킹 기법
- 시간 스탬프 순서 기법
- 삽입/삭제 연산과 동시성 제어

로킹(I)

- 정의
  - 상호 배제(독점 제어) 제공 ➔ 잠금 된 데이터 집합 생성
    - lock(잠금)을 수행한 트랜잭션만 독점적으로 접근
    - 다른 트랜잭션으로부터 간섭이나 방해를 받지 않음을 보장
    - lock(잠금)을 수행한 트랜잭션만 unlock(해제/풀림)할 수 있음
- 로킹 규약(locking protocol)
  - 트랜잭션 T가 read(x)나 write(x) 연산을 하려면 먼저 lock(x) 연산을 실행하고, 종료 전 unlock(x) 실행함
  - 트랜잭션 T는 다른 트랜잭션에 의해 이미 x에 lock이 걸려 있으면 다시 lock(x)를 실행시키지 못함
  - 트랜잭션 T는 x에 lock을 자기가 걸어 놓지 않았다면 unlock(x)를 실행시키지 못함

로킹(II)

- 로킹 모드의 확장
  - 공용 로크 lock-S
    - 공용된 접근 - read 연산만 허용
  - 전용 로크 lock-X
    - 배타적 접근 - read / write 연산을 허용
- 양립성(Compatibility)

T_i \ T_j	S	X
S	T	F
X	F	F

T : 접근 허용

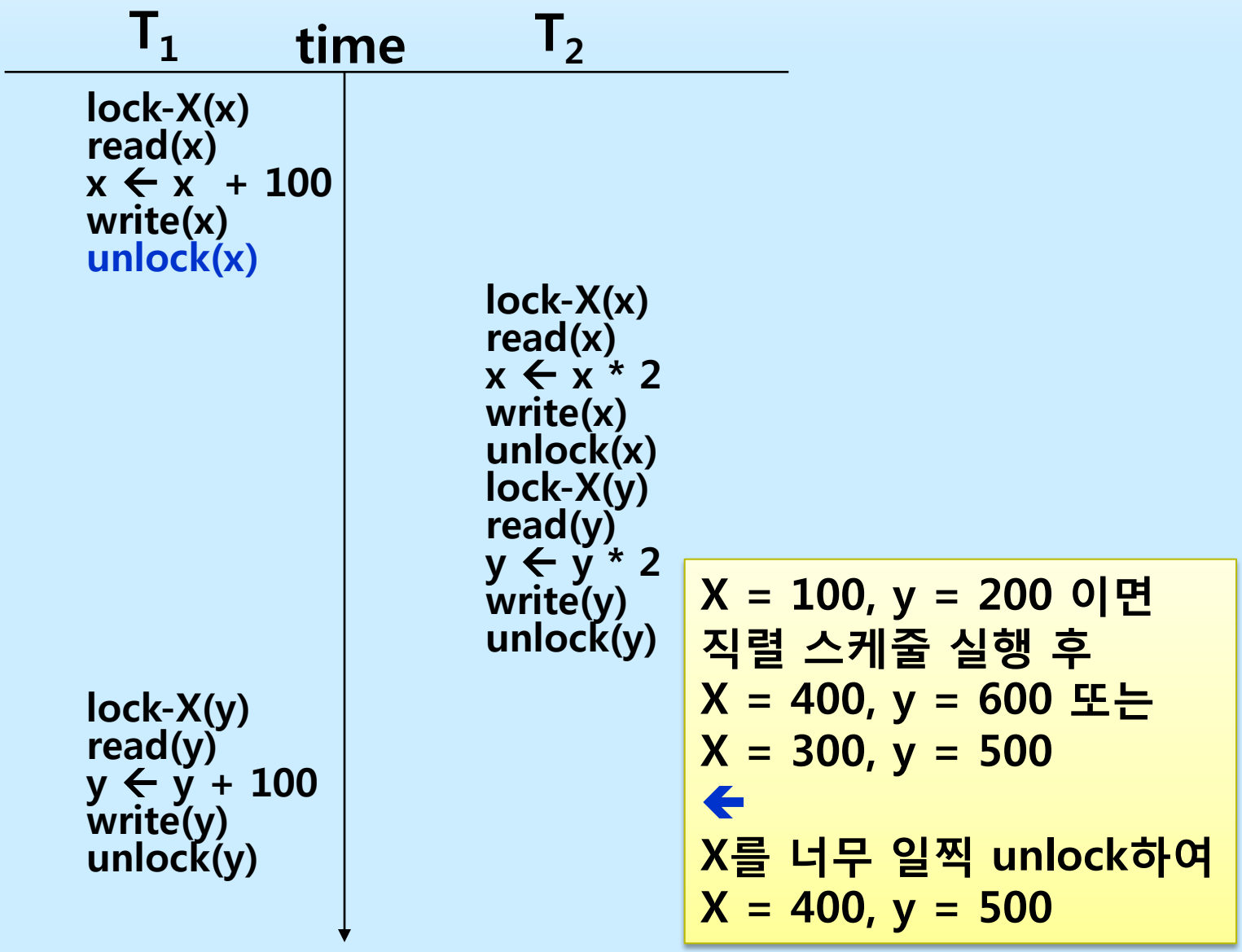
F : 대기

로킹(III)

❑ 공용 로킹 규약(shared locking protocol)

- ❑ 트랜잭션 T가 read(x) 연산을 실행하려면 먼저 lock-S(x) 또는 lock-X(x) 연산을 실행해야 한다
- ❑ 트랜잭션 T가 write(x) 연산을 실행하려면 먼저 lock-X(x) 연산을 실행해야 한다
- ❑ 트랜잭션 T가 lock-S(x)나 lock-X(x) 연산을 하려 할 때 x가 이미 다른 트랜잭션에 의해 양립될 수 없는 유형으로 lock이 걸려있다면 그것이 모두 풀릴 때까지 기다려야 한다
- ❑ 트랜잭션 T가 모든 실행을 종료하기 전에는 lock을 걸은 모든 x에 대해 반드시 unlock(x)를 실행해야 한다
- ❑ 트랜잭션 T는 자기가 lock을 걸지 않은 데이터 항목에 대해 unlock을 실행할 수 없다

공용 로킹 규약으로도 직렬가능이 아닌 스케줄



X = 100, y = 200 이면
직렬 스케줄 실행 후
X = 400, y = 600 또는
X = 300, y = 500
←
X를 너무 일찍 unlock하여
X = 400, y = 500

2단계 로킹 규약

- 2단계 로킹 규약(2PL)
  - 확장 단계(growing phase)

    트랜잭션은 lock만 수행하고 unlock은 수행할 수 없는 단계
  - 축소 단계(shrinking phase)

    트랜잭션은 unlock만 수행하고 lock은 수행할 수 없는 단계
- 스케줄 내의 모든 트랜잭션들이 2단계 로킹 규약을 준수한다면 그 스케줄은 직렬 가능

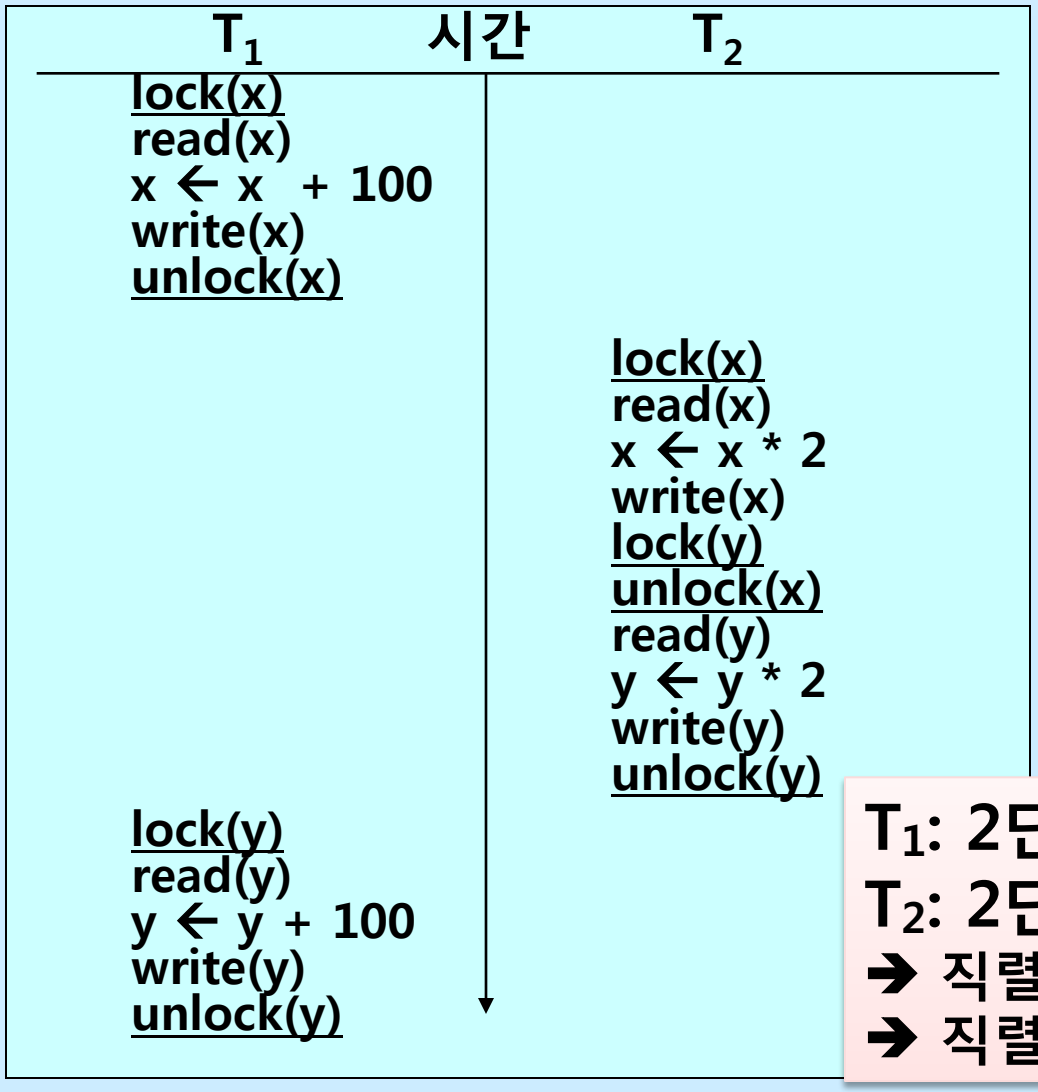
☞ Notes

- 2단계 → 직렬 가능을 보장
- 2단계는 직렬 가능성의 충분조건이며 필요조건은 아님

예 – 스케줄 S_1

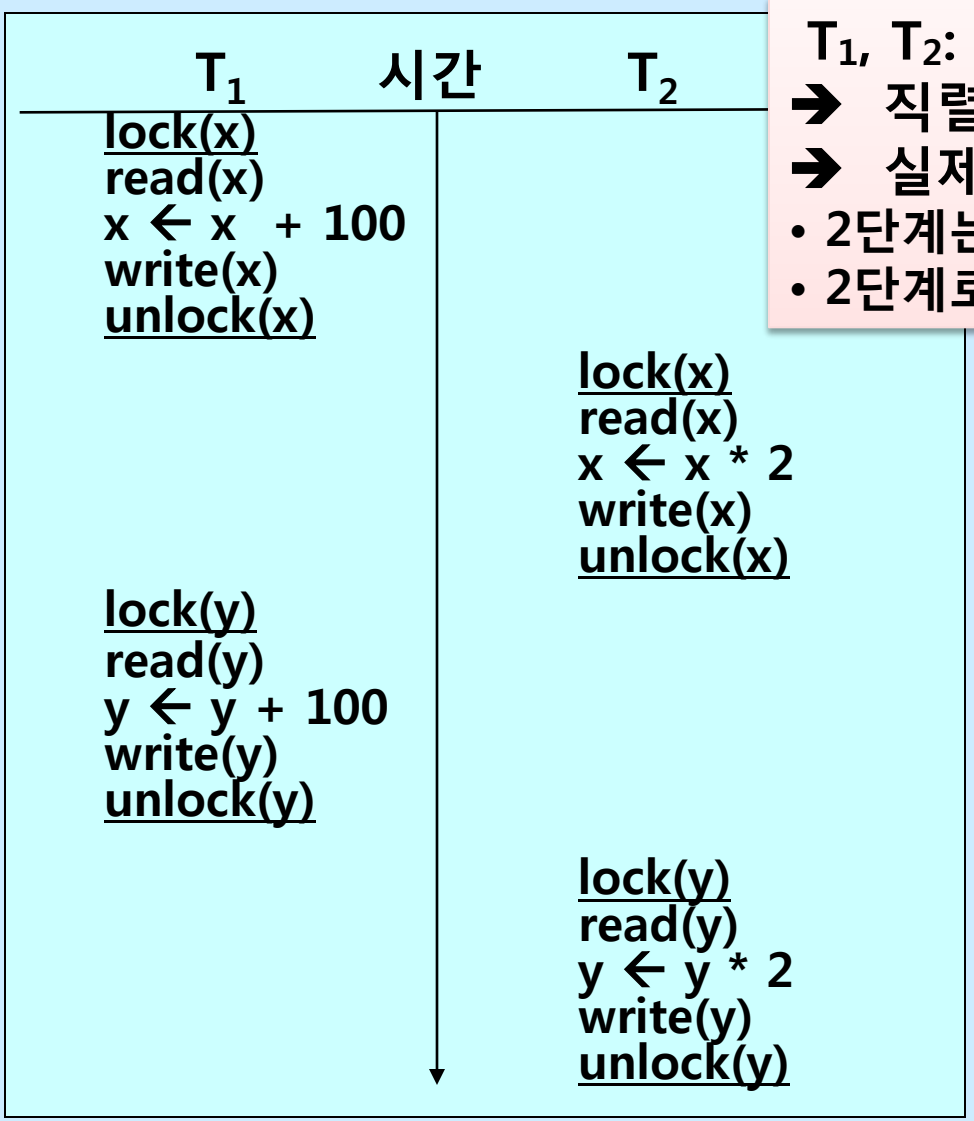
T_1, T_2 : 2단계 로킹 규약 준수
➔ 직렬가능

예 – 스케줄 S_2



T_1: 2단계 로킹 규약 미준수
T_2: 2단계 로킹 규약 준수
➔ 직렬 가능성을 보장하지 못함
➔ 직렬 가능한 스케줄이 아님

예 – 스케줄 S_3



T_1, T_2: 2단계 로킹 규약 준수하지 않음

➔ 직렬가능성을 보장하지 않음

➔ 실제, 직렬 가능 스케줄

- 2단계는 충분조건이고 필요조건이 아님
- 2단계로는 생성되지 않는 직렬 가능한 스케줄

Strict 2PL

- 완료 시 lock-X의 unlock
- 연쇄복귀문제 미발생

Rigorous 2PL

- 완료 시 모든 lock을 unlock
- 완료 순으로 직렬화

교착 상태(Deadlock)(I)

- 조건
  - ① 상호 배제(mutual exclusion)
  - ② 대기(wait for)
  - ③ 선취 금지(no preempt)
  - ④ 순환 대기(circular wait)

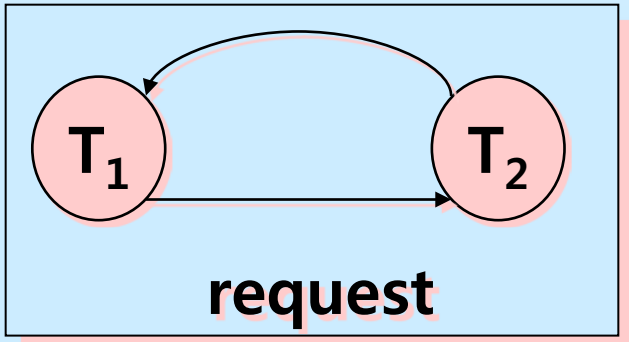
- 해결책
  - 탐지(detection) : 교착 상태가 일단 일어난 뒤에 교착 상태 발생 조건의 하나를 제거
  - 회피(avoidance) : 자원을 할당할 때마다 교착 상태가 일어나지 않도록 실시간 알고리즘을 사용하여 검사
  - 예방(prevention) : 트랜잭션을 실행시키기 전에 교착 상태 발생이 불가능하게 만드는 방법

교착 상태(II)

- 교착상태 예방
  - 트랜잭션 스케줄링 :
    - 충돌되는 데이터를 필요로 하는 트랜잭션
    - ➔ 병행 실행 불가
  - 실행 전에 필요한 데이터의 로크
    - ➔데이터에 대한 사전 지식이 요구
    - ➔기아문제 발생 가능함

교착 상태(IV)

- 교착상태의 회피(avoidance)
  - 요구 거부(Request Rejection)
    - 요구가 즉시 교착 상태를 유발
    - ➔로크 요구(선점 : preempt)를 거부

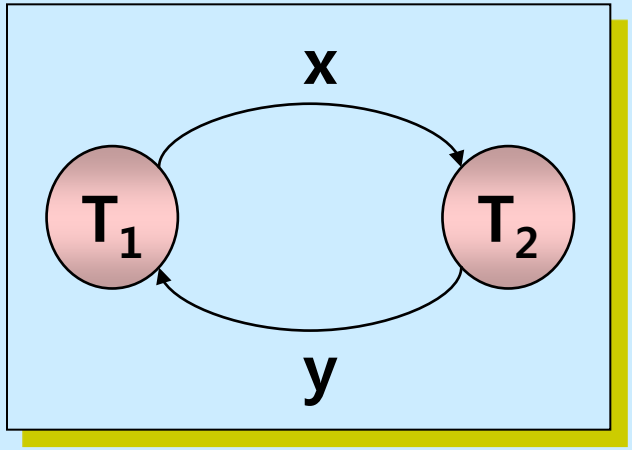


T_1을 거부하고 취소

  - 트랜잭션 재시도 : T_2에 의해 로크된 x를 T_1이 요구할 때
    - wait-die 기법 : 트랜잭션 T_i가 이미 T_j가 로크한 데이터 아이템을 요청할 때 만일 T_i의 시간 스템프가 T_j의 것보다 작은 경우(즉 T_i가 고참인 경우)에는 T_i는 기다린다. 그렇지 않으면 T_i는 복귀(즉 die)하고 다시 시작한다
    - wound-wait 기법 : 트랜잭션 T_i가 이미 트랜잭션 T_j가 로크한 데이터 아이템을 요청할 때 T_i의 시간 스템프가 T_j의 것보다 클 경우 (즉 T_j가 고참인 경우)에는 기다린다. 그렇지 않으면 T_j는 복귀해서(즉 T_i는 T_j를 상처 입힌다) 다시 시작한다

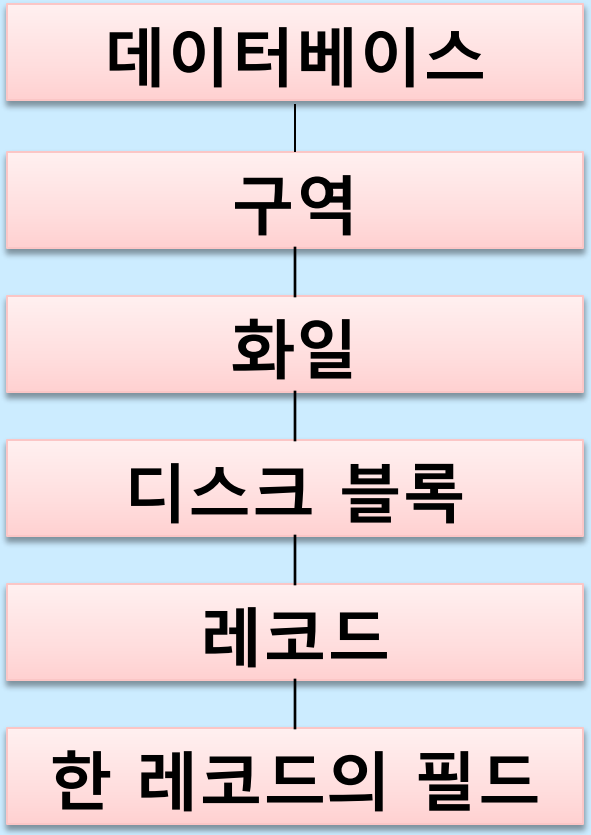
교착 상태(III)

- 교착상태 탐지(detection)
  - 시스템의 정보 유지
    - 데이터의 병행 할당
    - 대기중인 데이터(pending data)
  - 알고리즘 : 교착상태 탐지
    - 대기 그래프(V,E)
      - V : 트랜잭션
      - E : ($T_i \rightarrow T_j$) T_i 가 T_j를 대기 중
      - 사이클 ⇔ 교착상태
  - 교착 상태에서 회복
    - 취소할 트랜잭션 선택 : 최소 비용
    - 복귀(rollback) : 취소, 재 시작
    - 기아(starvation) : 같은 트랜잭션이 계속 취소
      - 완료되지 못함
      - FCFS가 도움이 됨



로킹단위 (I)

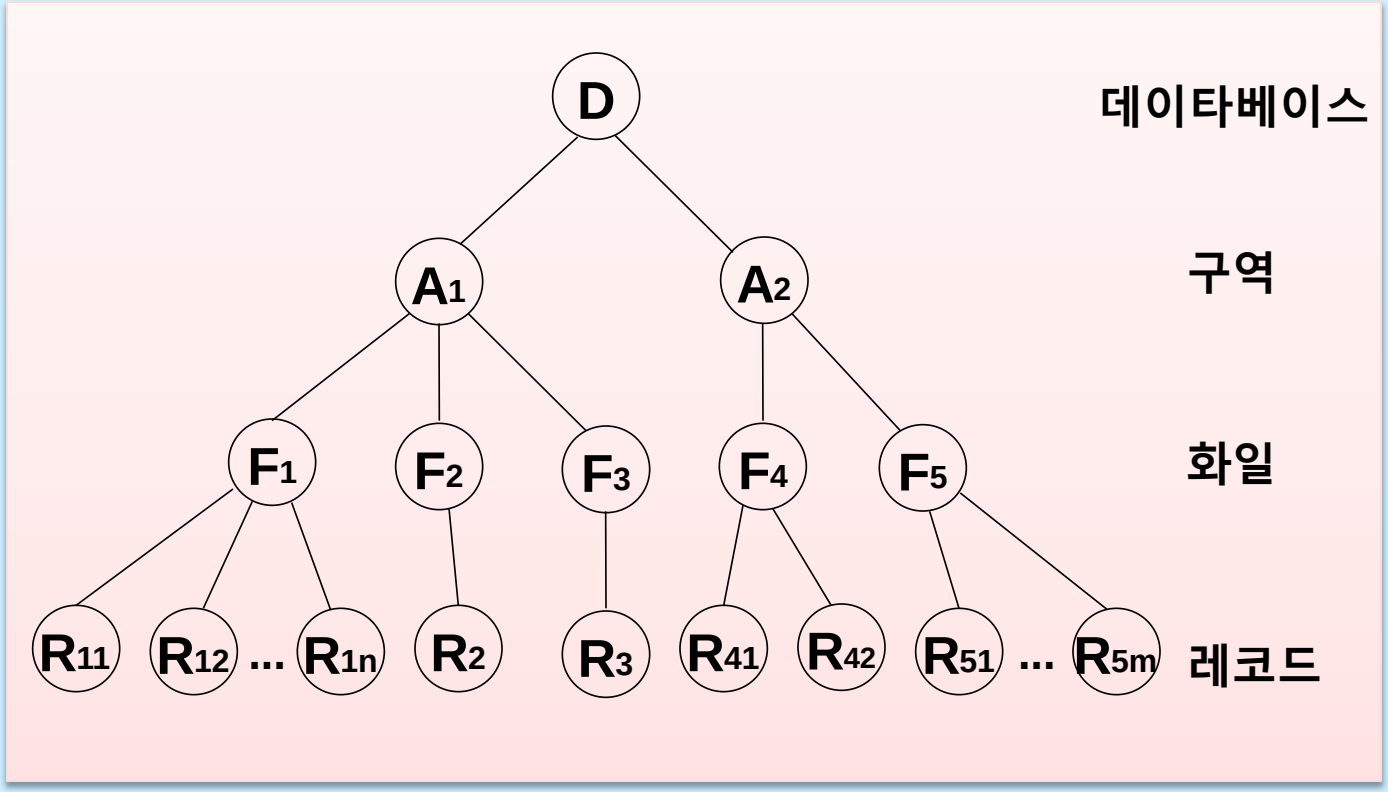
- 데이터 객체의 단위
  - 데이터 객체의 크기
    - 병렬 처리의 단위
  - 큰 로킹 단위
    - 낮은 병행성
  - 작은 로킹 단위
    - 많은 로크
    - 관리의 오버헤드



로킹단위 (II)

- 다단계 로킹(Multigranularity)
  - 빈번한 로크 – 로크 해제 연산의 오버헤드
    - 다단계 로킹 매커니즘의 필요
  - 데이터 아이템의 여러 크기
    - 긴 트랜잭션 : 큰 로킹 단위(coarse granularity)
    - 짧은 트랜잭션 : 작은 로킹 단위(fine granularity)
- 다단계 로킹 계층 트리

다단계 로킹 계층 트리 인스턴스



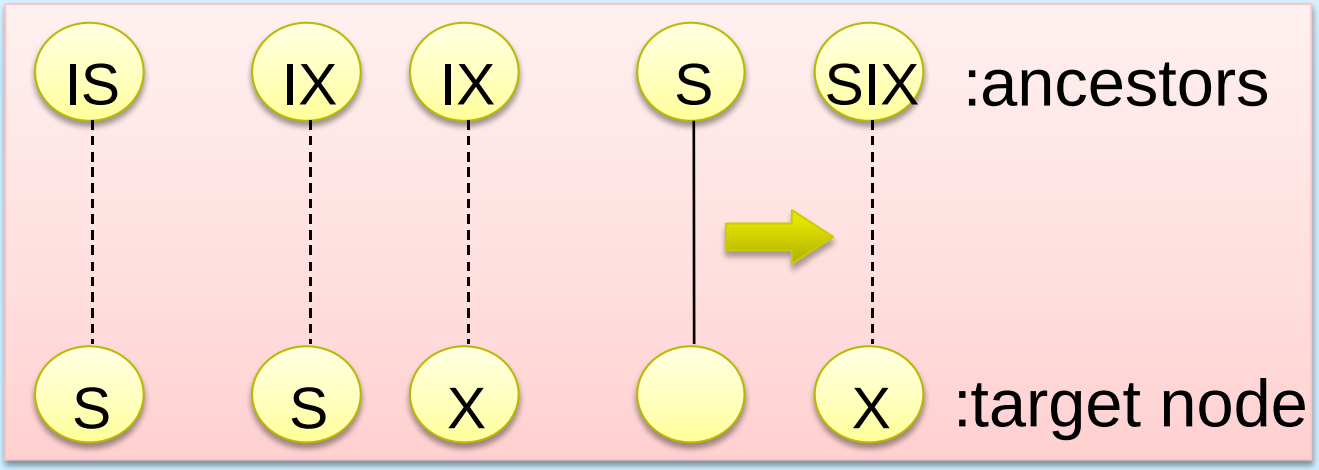
- 아이디어
  - 노드를 명시적으로 로크

➔ 로크 된 노드의 자손노드는 묵시적으로 같은 모드로 로크 됨

다단계 로킹 기법(I)

- 충돌 검사의 간단함
  - ➔ "경로 로킹" : 상위레벨에 대해 "의도형 로크 모드"
- 노드에 의도형 로크
  - 원하는 목표 노드를 명시적으로 로크하기 전에 그 노드의 모든 선조 노드들에 먼저 로크를 걸어야 함
  - 노드 N을 로크
  - 노드 N의 자손들은 노드 N과 같은 로크가 묵시적으로 이루어짐
- 의도형 로크
  - IS (의도 공용 로크)
    - ➔ 자손 노드를 S형 로크로 걸겠다는 것을 의미
  - IX (의도 전용 로크)
    - ➔ 자손 노드를 X형이나 S형 로크로 걸겠다는 것을 의미
  - SIX (공용 의도 전용 로크)
    - ➔ 현재 이 노드를 루트로 하는 서브트리가 명시적 S형 로크로 걸려 있는데 자손 노드를 명시적으로 X형 로크로 변경하겠다는 것을 의미

다단계 로킹 기법(II)

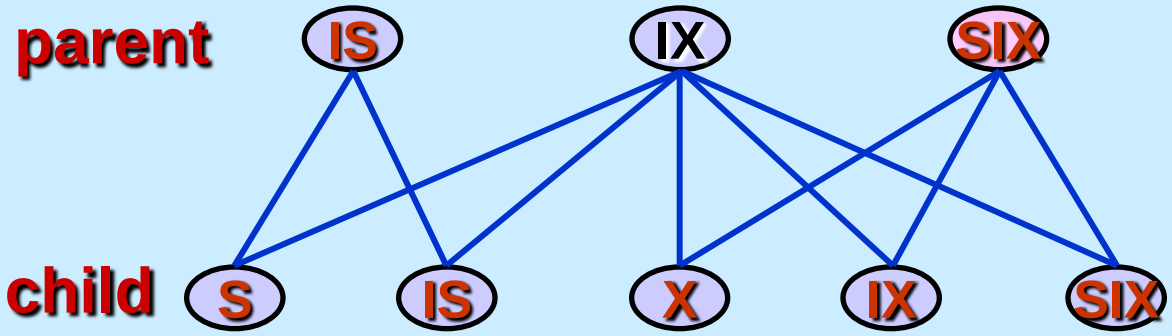


- 의도형 로크의 양립성

T_i \ T_j	X	S	IS	SIX	IX
X	F	F	F	F	F
S	F	T	T	F	F
IS	F	T	T	T	T
SIX	F	F	T	F	F
IX	F	F	T	F	T

다단계 로킹 기법(III)

- 다단계 로킹 규약
  - 로크의 양립성이 준수되어야 하고 로크는 반드시 트리의 루트부터 걸어야 한다
  - T_i가 N의 부모노드를 현재 IX나 IS로 로크했다면 T_i는 N을 S나 IS로 로크할 수 있고, IX나 SIX로크했다면 T_i는 N을 X, SIX 또는 IX로 로크할 수 있다
  - T_i가 어떤 유형의 unlock 연산을 수행한 적이 없어야 lock을 수행할 수 있다 (2단계 로킹)
  - T_i가 로크한 N의 자손 중에 T_i에 의해 로크된 것이 없을 때 T_i는 N을 unlock할 수 있다
- 부모 / 자식 노드 간의 의도형 로크 관계



예제

- 트랜잭션
  - T_1: F_1에 있는 레코드 R_{11}을 판독하기 위해서는 D, A_1 그리고 F_1을 IS형으로 로크한 다음 R_{11}을 S형으로 로크
  - T_2: F_1에 있는 R_{12}를 갱신하기 위해서는 D, A_1, F_1을 IX형으로 로크한 다음 R_{12}를 X형으로 로크
  - T_3: F_1을 판독하기 위해서는 D, A_1을 IS형으로 로크한 다음 F_1을 S형으로 로크
  - T_4: 데이터베이스 D를 판독하기 위해 D를 S형으로 로크
- 트랜잭션 T_1, T_3, T_4는 데이타베이스를 병행 접근 가능
- 규약은 병행성을 증대시키고 로크 부하를 감소

배울 내용

- 동시성 제어
- 로킹 기법

- 시간 스탬프 순서 기법
- 삽입/삭제 연산과 동시성 제어

타임스템프 순서 기법(I)

- 트랜잭션을 인터리브로 실행
- 시간 스탬프 TS(T_n)
  - 트랜잭션 T_n의 실행 시작시간: 시스템 클럭 값 또는 논리적 카운터
  - T_i가 T_j보다 오래되면 TS(T_i) < TS(T_j)
- 타임스탬프 순서 기법의 아이디어 : TS(T_i) < TS(T_j)
  ➔시스템이 <T_i, T_j>의 직렬 실행과 결과가 일치하도록 보장
- 데이터 아이템 x의 타임스템프
  - read-TS(x)
    데이터 아이템 x의 판독시간 스템프로 read(x)를 성공적으로 수행한 트랜잭션의 타임스탬프 중에서 제일 큰 타임스탬프
  - write-TS(x)
    데이터 아이템 x의 기록시간 스탬프로 write(x)를 성공적으로 수행한 트랜잭션의 타임스탬프 중에서 제일 큰 타임스탬프

시간 스탬프 순서 기법(II)

❑ 시간 스탬프 순서 규약

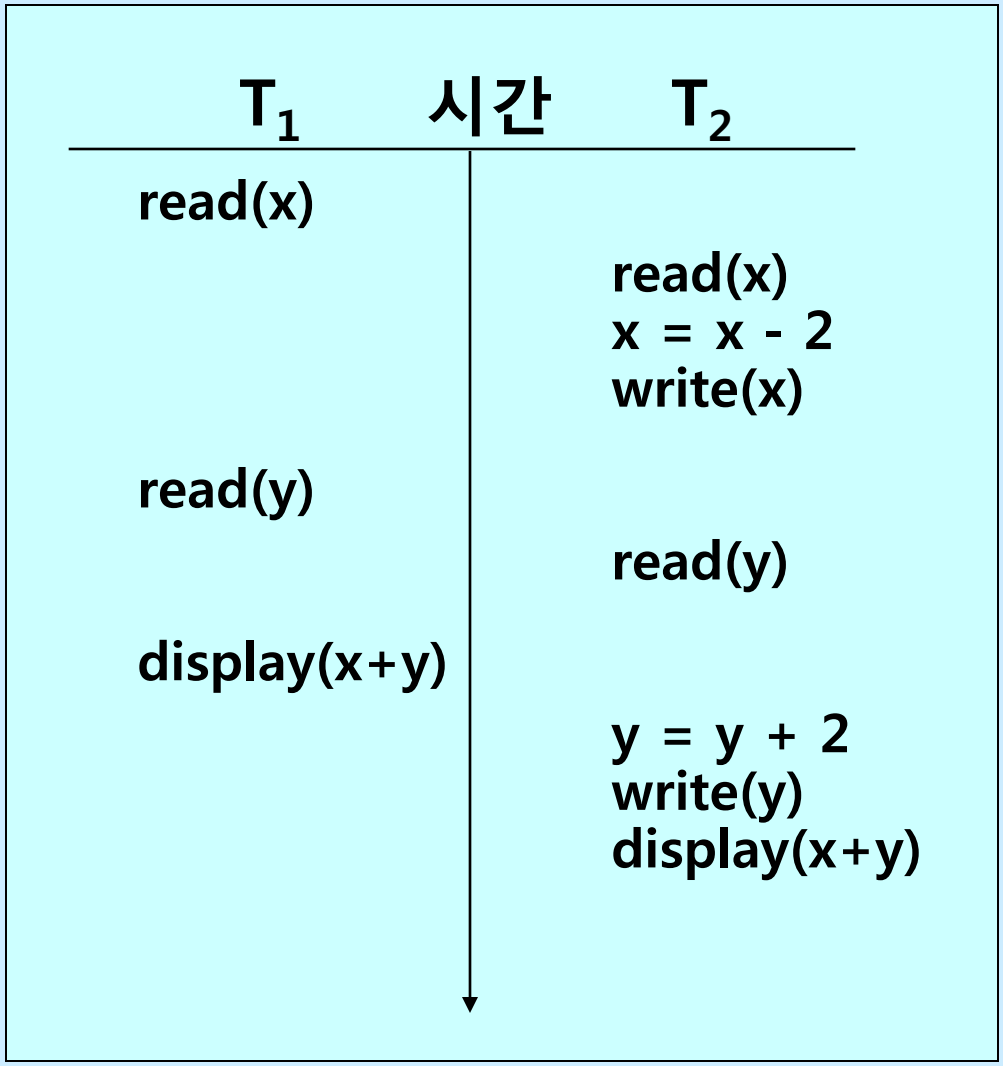
❑ T_i가 **read(x)**를 수행하려 할 때
  TS(T_i) ≥ write-TS(x)이면
    read(x)를 허용하고
    read-TS(x) ← max{ read-TS(x), TS(T_i) }
  아니면
    read(x)를 거부하고, T_i는 복귀된다

❑ T_i가 **write(x)**를 수행하려 할 때
  TS(T_i) ≥ read-TS(x)이고 **TS(T_i) ≥ write-TS(x)**이면
    write(x)를 허용하고,
    write-TS(x) ← TS(T_i)
  아니면
    write(x)를 거부하고, T_i는 복귀된다

시간 스탬프 순서 기법(III)

- 시간 스탬프 순서 기법의 장단점
  - 교착상태가 없음 : 대기가 없기 때문
  - 연쇄 복귀
    - T_i의 복귀가 T_j의 복귀를 유발
  - 순화적 재시작 : 기아(starvation)
    - 연속적인 복귀와 재시작

예제



- 시간 스탬프 순서 스케줄

☞ Notes

- 2단계 로킹으로 생성 가능

Thomas의 기록 규칙(revised TS protocol)(I)

- 잠재적 병행성의 증대

- Write 규칙의 수정(no change in read rule)

  T_i가 write(x)를 수행하려 할 때 :

  만일 $TS(T_i) < read\text{-}TS(x)$ 이면

  write(x)를 거부하고 T_i를 취소시켜 복귀시킨다

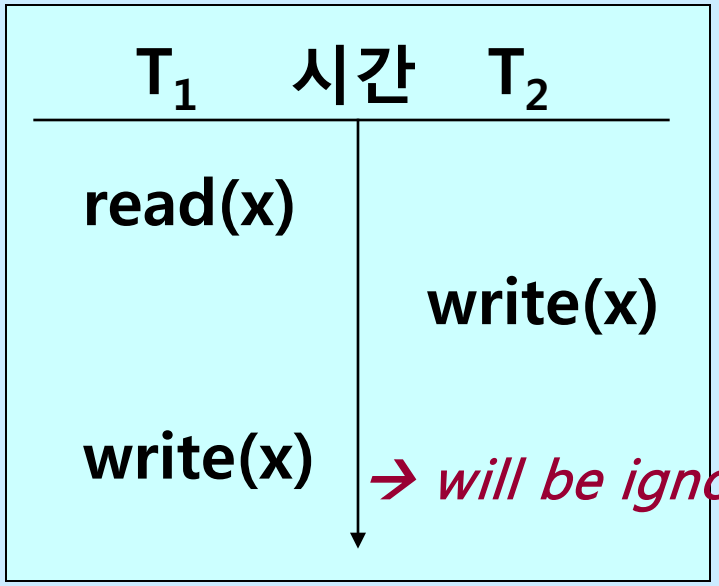
  만일 $TS(T_i) \geq read\text{-}TS(x)$ 이고 $TS(T_i) < write\text{-}TS(x)$ 이면

  write(x)를 단순히 무시한다

  그 이외의 경우는

  write(x)를 허용하고, $write\text{-}TS(x) \leftarrow TS(T_i)$

Thomas의 기록 규칙(II)



TS(T1) >=read-TS(x) and
TS(T1) < write-TS(x)

- 기존의 시간 스탬프 규약에서는 T_1의 write 연산이 거부되면 T_1이 복귀
- Thomas의 기록 규칙은 무용의 write연산을 무시하고 스케줄이 직렬 가능

낙관적 병행 제어(I)

- 데이터베이스 연산 실행 전에 검사

  → 검사 시 오버헤드

- 낙관적 병행 제어 - 트랜잭션의 실행을 3단계로 나눔
  - 판독 단계(R)
    - 지역 변수만을 이용하여 읽기와 갱신 수행
  - 확인 단계(V)
    - 실제 데이터베이스에 반영하기 전에 충돌 직렬 가능성 검사
  - 기록 단계(W)
    - 확인 단계를 통과하면 트랜잭션의 실행결과는 실제로 데이터베이스에 반영
    - 그렇지 않으면, 트랜잭션은 취소되고 재시작

  ➔ 모든 검사를 나중에 한꺼번에...

낙관적 병행 제어 (II)

- 각 트랜잭션에 3가지 타임스탬프 사용
  - Start(T_i)
    - 트랜잭션 T_i가 판독 단계에 들어가면서 실행을 시작한 시간
  - Validation(T_i)
    - 트랜잭션 T_i가 판독단계를 끝내고 확인을 시작한 시간
  - Finish(T_i)
    - 트랜잭션 T_i가 최종 기록 단계를 완료한 시간

- 직렬 가능 순서
  - order of validation(T_i) (= TS(T_i))
  - validation(T_i) < validation(T_j) ⇒ <T_i,T_j>

낙관적 병행 제어 (III)

- T_k의 Validation 검사
  - $TS(T_i) < TS(T_k)$이라 가정
  - 다음의 세 조건 중 하나를 만족

① $Finish(T_i) < Start(T_k)$
T_i 가 T_k 시작 전에 완성

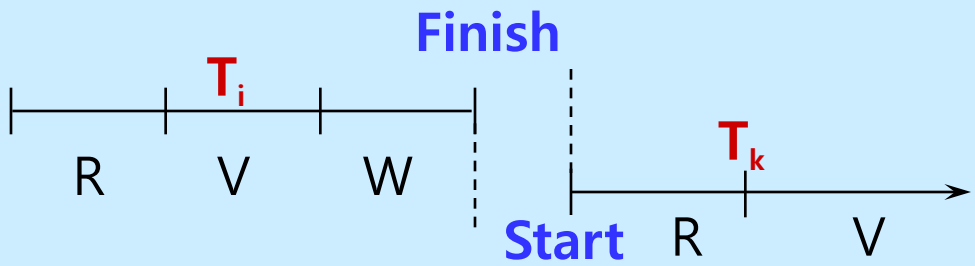
② $Start(T_k) < Finish(T_i) < Validation(T_k)$
and $Write\text{-}set(T_i) \cap Read\text{-}set(T_k) = \varnothing$

③ $Write\text{-}set(T_i) \cap Read\text{-}set(T_k) = \varnothing$ and
$Write\text{-}set(T_i) \cap Write\text{-}set(T_k) = \varnothing$

- 장점 vs 단점
  - 교착상태가 없음(no deadlock)
  - 연쇄 복귀가 없음(no cascading rollback)
  - 순환적 재시작(cyclic restart (starvation))

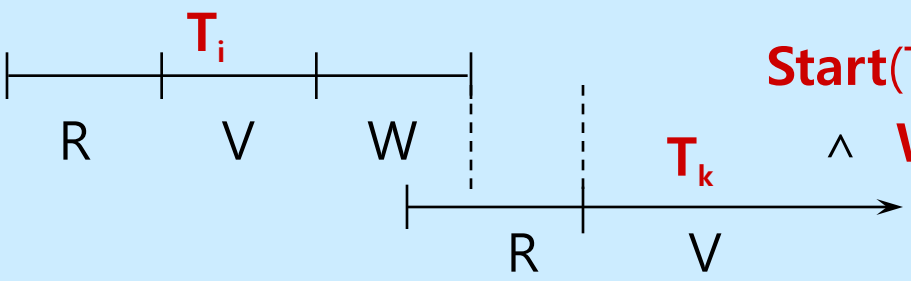
확인 검사 조건 : $TS(T_i) < TS(T_k)$

①



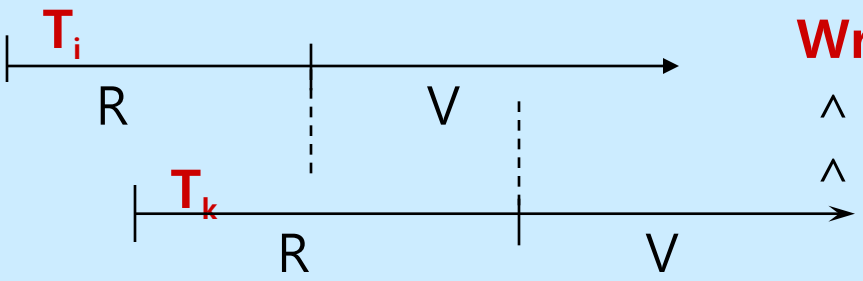
$Finish(T_i) < Start(T_k)$

②



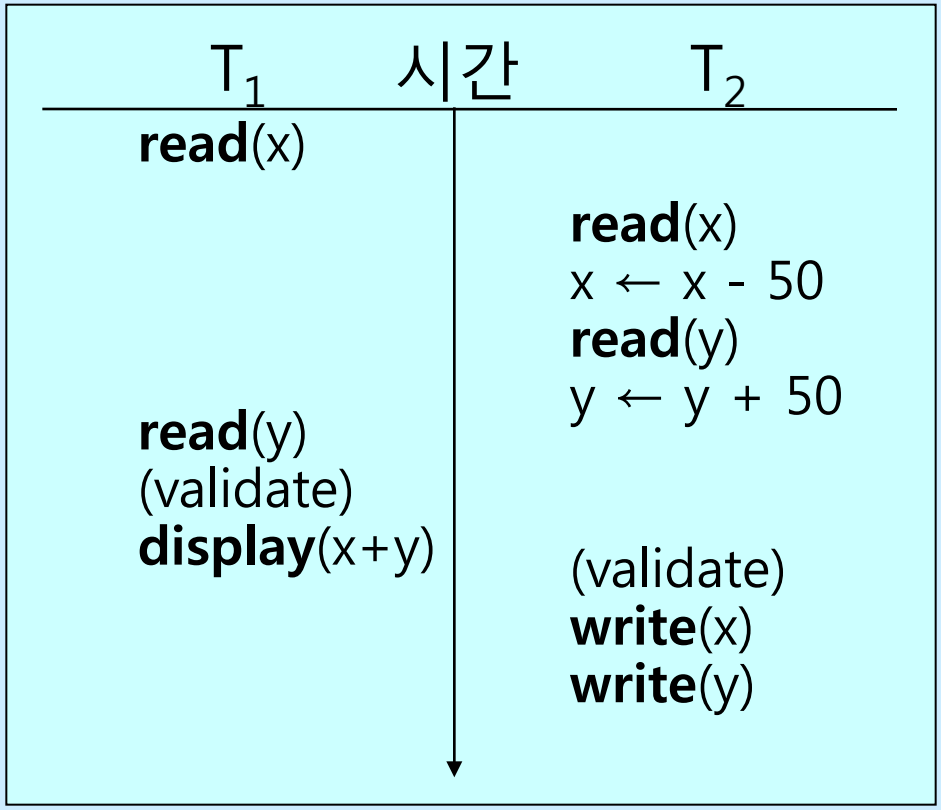
$Start(T_k) < Finish(T_i) < Validation(T_k)$
$\wedge$ $Write\text{-}set(T_i) \cap Read\text{-}set(T_k) = Æ$

③



$Write\text{-}set(T_i) \cap Read\text{-}set(T_k) = Æ$
$\wedge$ $Write\text{-}set(T_i) \cap Write\text{-}set(T_k) = Æ$
$\wedge$ $Validation(T_i) < Validation(T_k)$

예제



- 낙관적 기법에 의한 직렬 가능 스케줄

☞ Note: 2PL이나 타임스탬프 기법으로 생성 불가능

팬텀 충돌(Phantom Conflict)

T_1 : SELECT SUM(SAL)
FROM EMP
WHERE DEPT = 'COMP ENG'
T_2 : INSERT INTO EMP
VALUES(E123, 'LEE', 'COMP ENG', 150)

T_1과 T_2는 데이타베이스에서 공통 투플을 접근하지 않음
즉, 트랜잭션 T_1과 T_2는 실제 데이타에 있어서 서로 충돌하지 않음

$$< T_1, T_2 > \neq < T_2, T_1 >$$

원인 : 데이타베이스에 삽입되어질 투플, 즉 팬톰 투플에 대해 T_1과 T_2 가 서로 충돌되기 때문

☞ Note: 오직 투플 단위에서만 적용

팬텀 충돌 현상의 해결책

- 팬텀이 아닌 실제 데이터의 충돌을 유도
  - 로킹 단위를 크게
    - 로킹 대상 데이터 단위를 투플이 아니라 릴레이션으로.
  - 인덱스 로킹 기법 이용 (릴레이션과 그것의 인덱스를 갱신하는 것을 의미)
    - 모든 릴레이션은 적어도 하나의 인덱스를 가지고 있어야 한다
    - 트랜잭션 T_i는 접근하려는 릴레이션의 투플 t에 대한 포인터가 있는 인덱스 버켓에 S형 로크를 걸었을 때에만 그 투플 t에 대해 S형 로크를 걸 수 있다
    - 트랜잭션 T_i는 갱신하려는 릴레이션의 투플 t에 대한 포인터가 있는 인덱스 버켓에 X형 로크를 걸었을 때에나 그 투플 t에 대해 X형 로크를 걸 수 있다
    - 트랜잭션 T_i는 투플을 삽입하기 전에 릴레이션의 모든 인덱스를 갱신하여야 하고 갱신하려는 모든 인덱스 버켓에 X형 로크를 걸어야 한다
    - 로킹은 2단계 로킹 규약에 따라야 한다

배울 내용

- 동시성 제어
- 로킹 기법
- 시간 스탬프 순서 기법

- 삽입/삭제 연산과 동시성 제어

삽입 / 삭제 연산과 병행 제어 (I)

insert(x) : x 이미 존재

delete(x) : x 새로 생성

❑ 양립성

T_j \ T_i	read	write	delete	insert
insert	x	x	x	x
delete	x	x	x	x

x : 충돌

삽입 / 삭제 연산과 병행 제어 (II)

$read_i(x)$: 논리적 오류
$insert_j(x)$

$insert_j(x)$
$read_i(x)$: 성공

$delete_i(x)$
$read_j(x)$: 논리적 오류

$read_j(x)$
$delete_i(x)$: 성공

- insert/delete 연산은 모두 write연산으로 취급
  - 2단계 로킹 규약에서는 전용 lock을 사용
  - 시간 스탬프 순서 규약에서도 write 연산으로 취급

요약

- 동시성 제어
- 로킹 기법
- 시간 스탬프 순서 기법
- 삽입/삭제 연산과 동시성 제어

다음 배울 내용 : 회복 기법