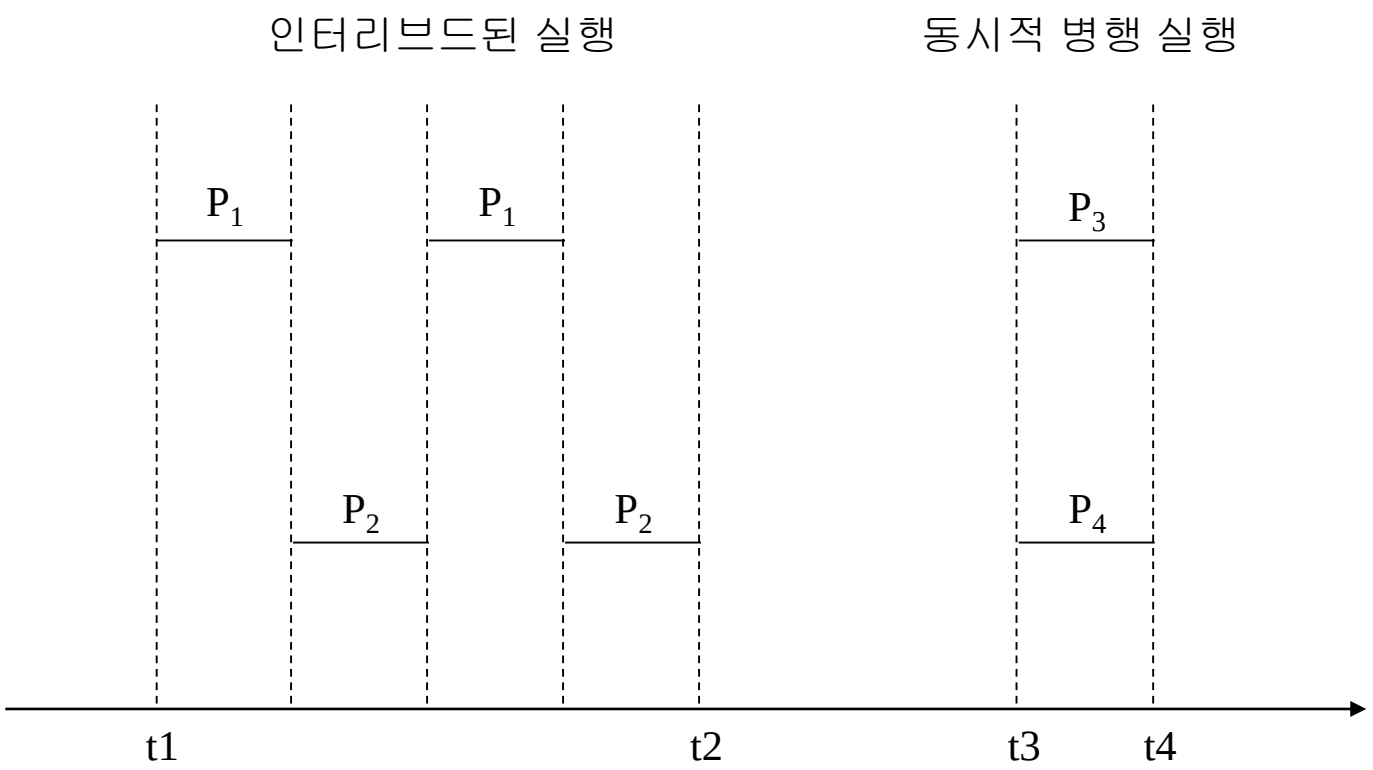
16. 병행 제어

❖ 병행 제어

- 데이타베이스 시스템의 주요 목표
  - 공용성(sharability)
    - 여러 사용자가 데이타베이스를 이용 가능
    - 단일 사용자 DBMS vs 복수 사용자 DBMS(병행 데이타베이스)
  - 정확성(accuracy)
    - 공용된 데이타베이스를 정확히 유지
- 동시 공용(Concurrent Sharing)의 이점
  - 공용도(sharability)의 증가
  - 응답 시간(response time)의 단축
  - 시스템 활용도(system utilization) 증대
- 복수 사용자 시스템
  - 병행 접근
  - 다중 프로그래밍(multiprogramming)
    - 인터리브된(interleaved) 실행 : 하나의 CPU 사용
    - 동시적 병렬 처리(parallel processing) : 복수의 CPU 사용

▶ 인터리브 대 동시적 병행 실행

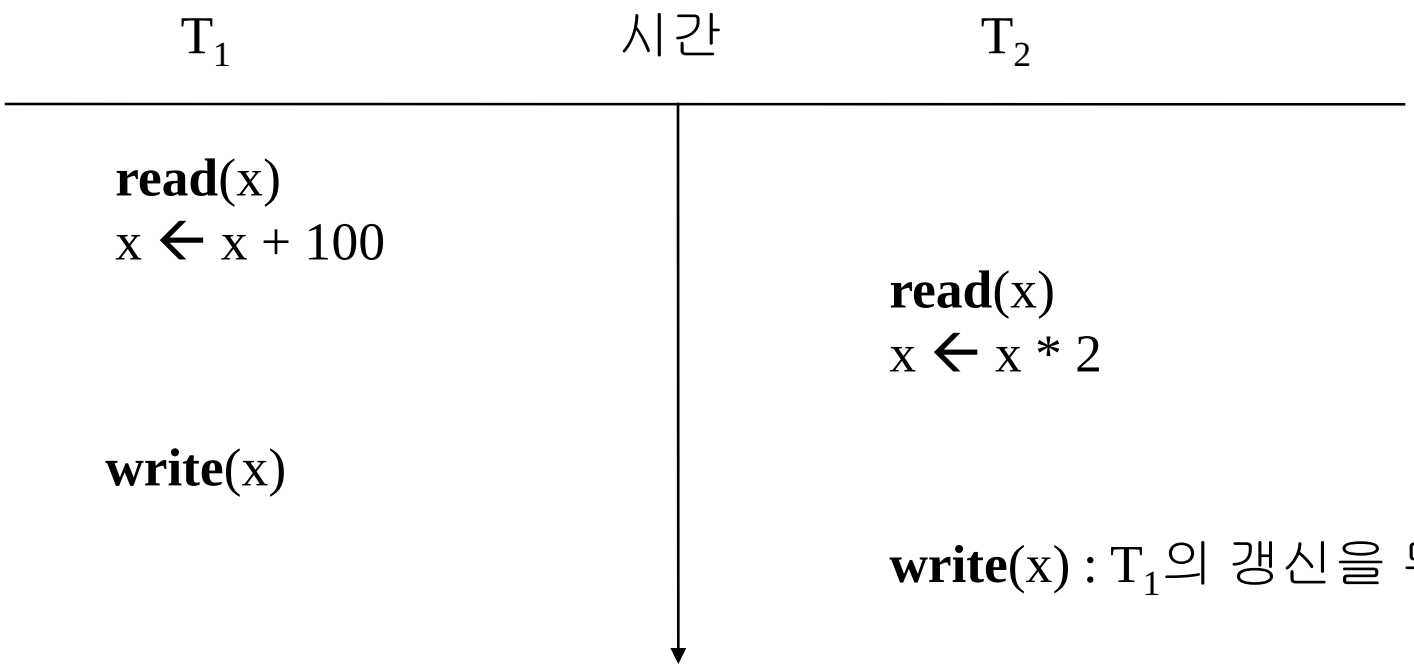


▶ 무제어 동시 공용의 문제점

- 문제점
  1) 갱신 분실(lost update)
     - 탐지 불가능
  2) 모순성(inconsistency)
     - 데이타베이스의 출력내용과 모순
  3) 연쇄 복귀(cascading rollback)
     - 상호 의존(tangled dependencies)
     - 연쇄 회복(cascade recovery)

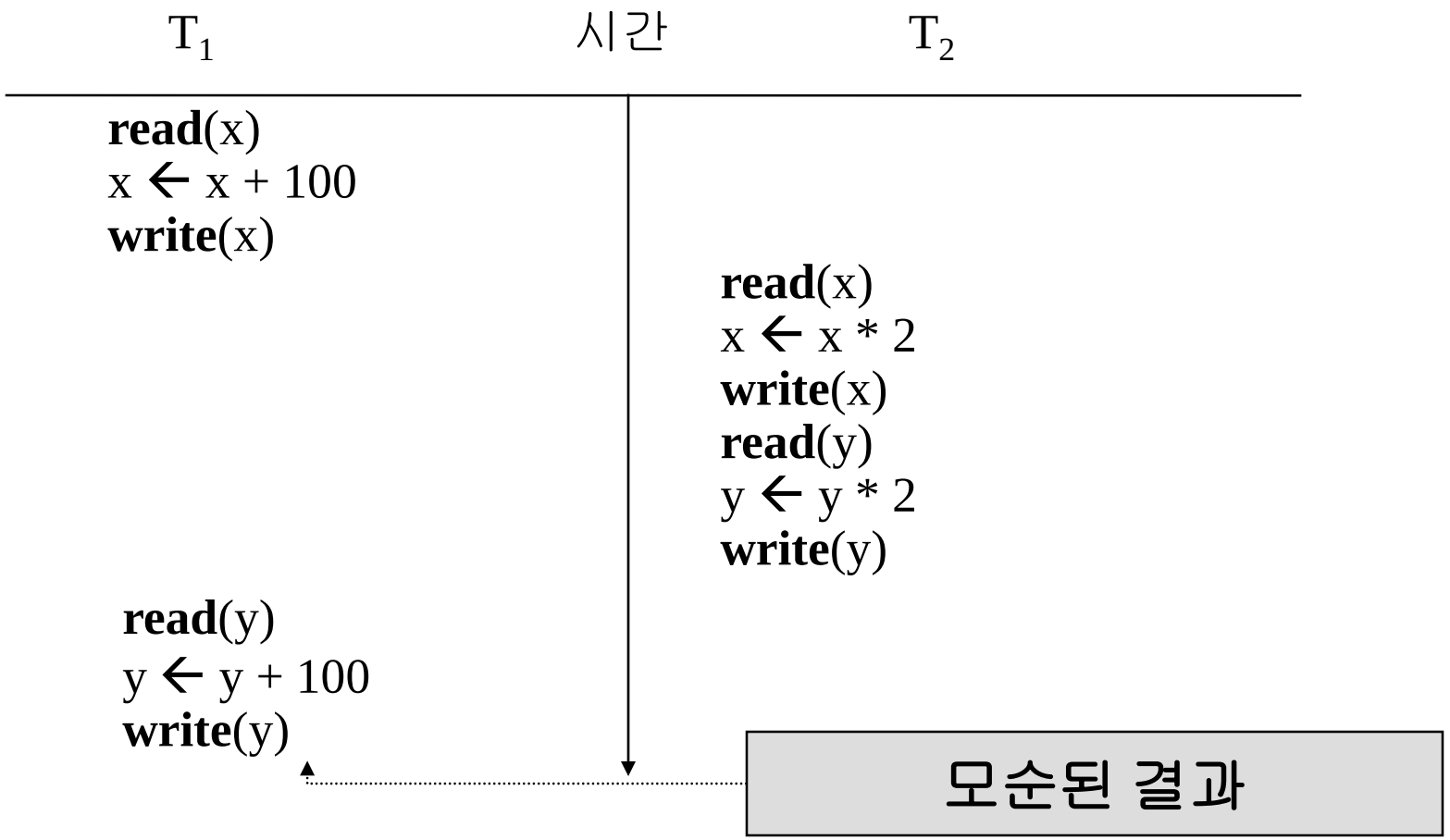
(1) 갱신분실(Lost Update)

- 탐지 불가능(Undetectable)



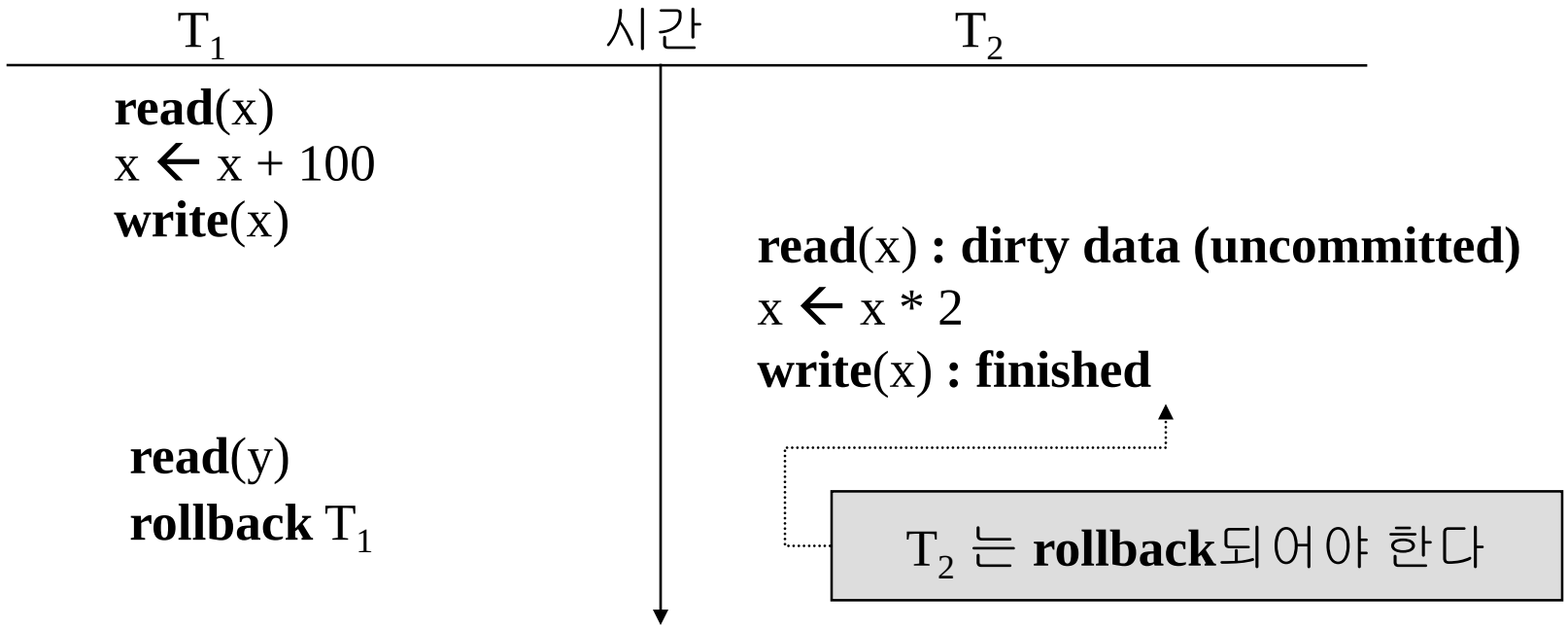
(2) 모순성(inconsistency)

- 데이타베이스의 출력 내용과 모순



(3) 연쇄 복귀(Cascading Rollback)

- 상호 의존(tangled dependencies)
- 연쇄 회복(cascade recovery)



- 완료되지 않은 데이타 접근

(3) 연쇄 복귀(Cascading Rollback)

- 문제의 원인
  - 충돌(conflict) :
    ① 상이한 트랜잭션에 속하고 있으면서 동일한 데이타 아이템을 처리 대상으로 하는 두 연산
    ② 최소한 하나의 기록(**write**) 연산
  - 공용하는 충돌된 데이타를 통해 트랜잭션 사이에 간섭이 일어나기 때문
  - 충돌이 일어나는 경우
    - **read**$_i$(x)와 **write**$_j$(x), **write**$_i$(x)와 **read**$_j$(x), **write**$_i$(x)와 **write**$_j$(x)
  - 충돌이 일어나지 않는 경우
    - **read**(x)와 **write**(y), **write**(x)와 **read**(y), **write**(x)와 **write**(y)

- 병행 제어
  - 충돌 데이타의 관리

❖ 직렬 가능성의 개념(1)

- 스케줄
  - 실행 순서
  - 트랜잭션 연산들의 순서
- 직렬 스케줄(serial schedule)
  - 트랜잭션 $\{T_1, \ldots, T_n\}$의 순차적 실행
  - 인터리브드되지 않은 스케줄
  - 스케줄의 각 트랜잭션 T_i의 모든 연산 $<T_{i1},\ldots,T_{in}>$가 연속적으로 실행
  - n! 가지의 방법
  - 직렬 스케줄은 정확하다고 가정
- 비직렬 스케줄(nonserial schedule)
  - 인터리브된 스케줄
  - 트랜잭션 $\{T_1,\ldots,T_n\}$의 병렬 실행

❖ 직렬 가능성의 개념(2)

- 직렬 가능 스케줄

  n개의 트랜잭션 $T_1,...,T_n$에 대한 스케줄 S가 똑같은 n개의 트랜잭션에 대한 어떤 직렬 스케줄 S'과 동등하면 스케줄 S는 직렬 가능 스케줄

  - 직렬 스케줄 $S_1 : <T_1, T_2, T_3>$

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

  - 비직렬 스케줄 S_2

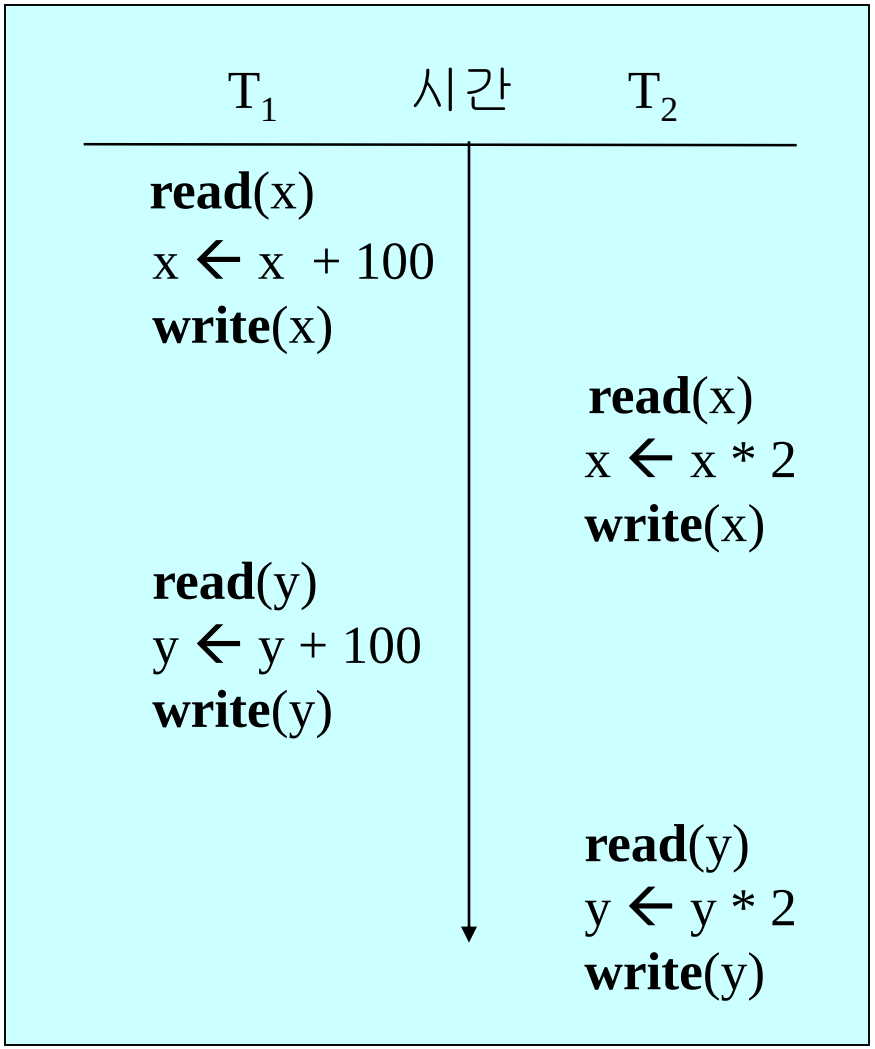
    $$S_2 : < O_{11}, O_{21}, O_{22}, O_{12}, O_{31}, O_{23}, O_{13}, O_{32}, O_{14} >$$

  - S_2가 직렬 스케줄 $<T_1,T_2,T_3>$과 동등하다면 S_2는 직렬 가능한 스케줄

예 – 두 개의 트랜잭션

T_1	T_2
read(x)	**read**(x)
x ← x + 100	x ← x * 2
write(x)	**write**(x)
read(y)	**read**(y)
y ← y + 100	y ← y * 2
write(y)	**write**(y)

예 – 비직렬 스케줄


T1
시간
T2
read(x)
x ← x + 100
write(x)
read(x)
x ← x * 2
write(x)
read(y)
y ← y + 100
write(y)
read(y)
y ← y * 2
write(y)

❖ 스케줄 동등

- 결과 동등(**result equivalent**)
  - 최소한도의 만족성을 가진 스케줄 동등성 정의
  - 데이타베이스에 대한 스케줄의 결과만을 비교
    - 동일한 초기 상태
    - 두 개의 상이한 스케줄
    - 똑같은 생성된 데이타베이스 최종 결과

      ⇒ 결과 동등

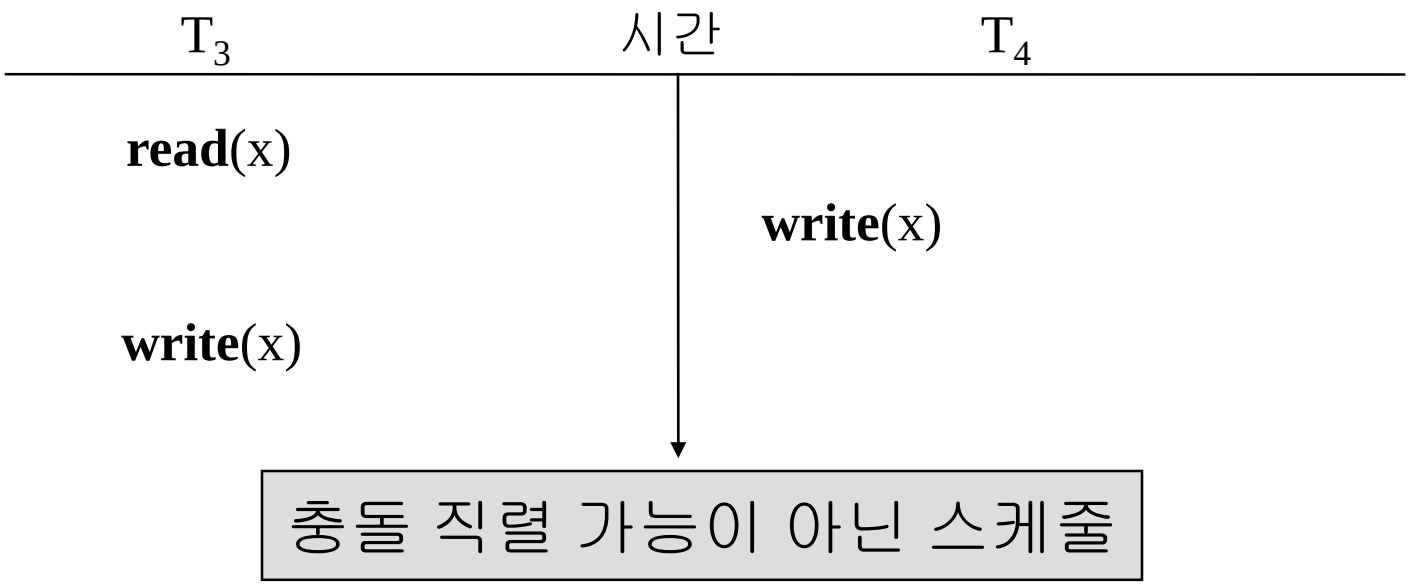
Note

연산자의 유형이나 피연산자의 값에 따라 우연히 최종 결과가 같을 수도 있음

⇒ 항상 동일한 결과를 생성하는 것을 보장 안 함

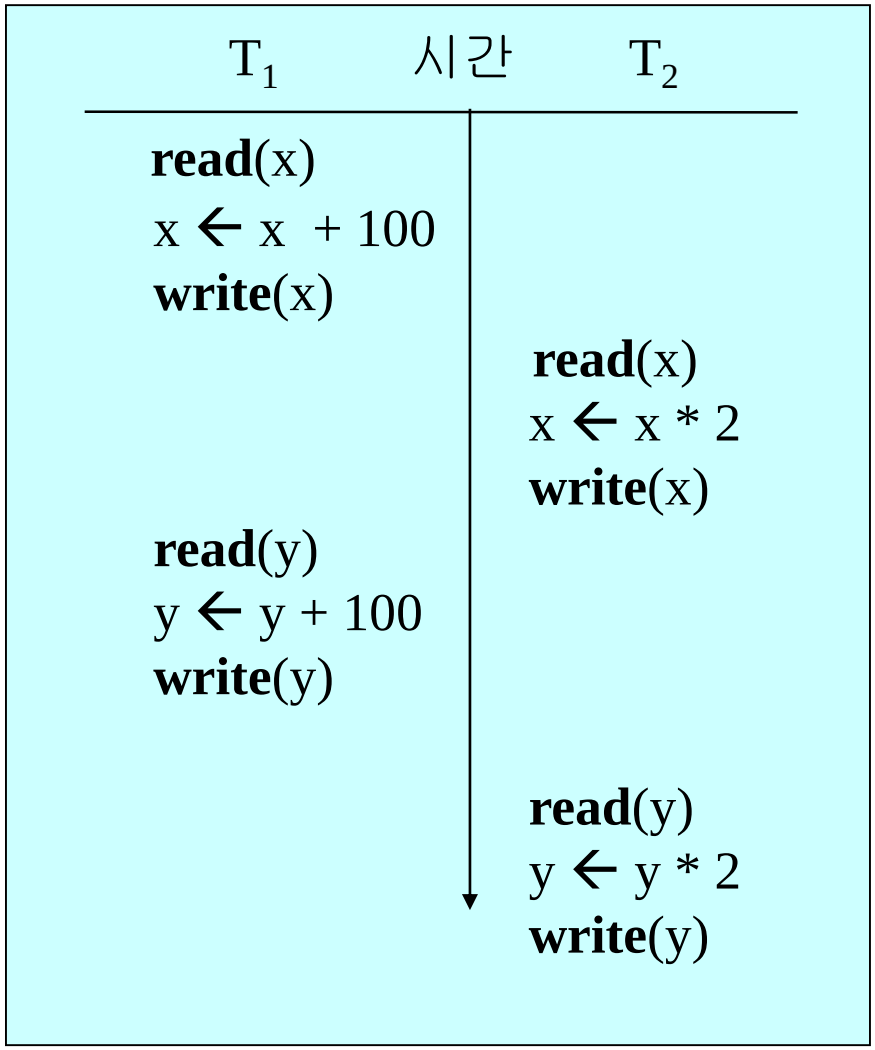
충돌 동등(Conflict Equivalent)

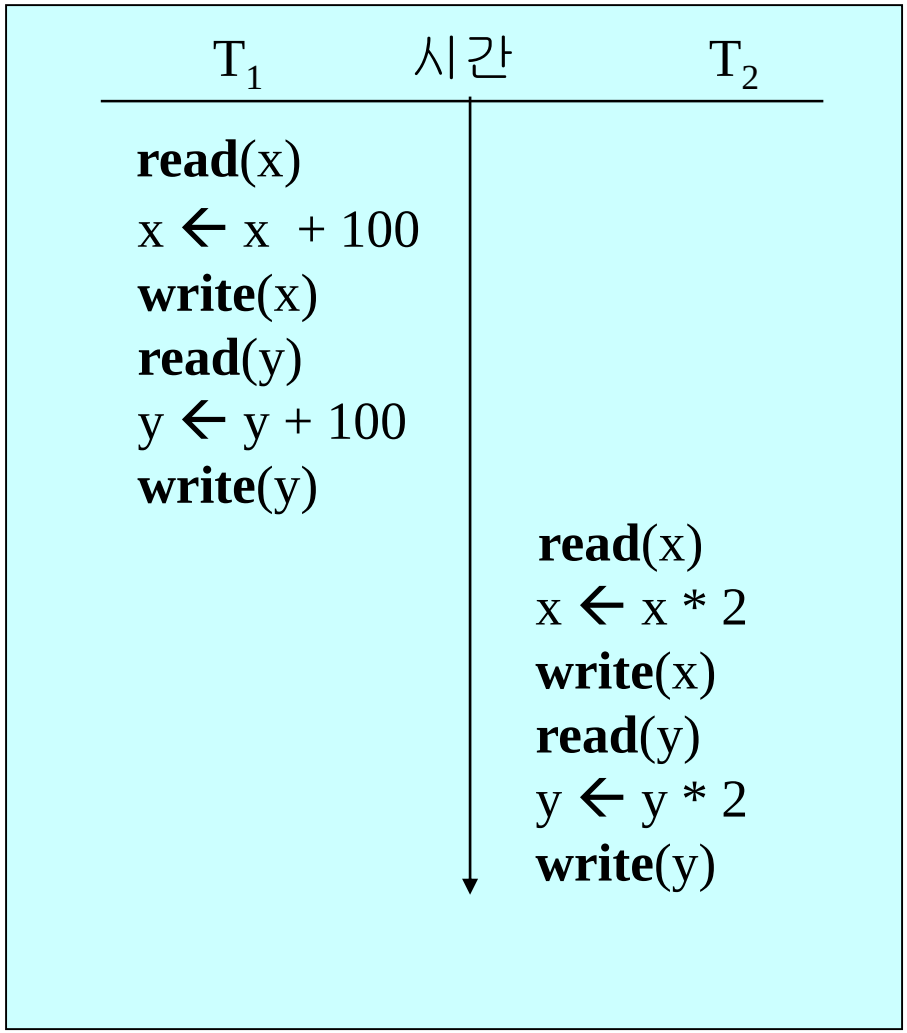
- 충돌 동등
  - 두 스케줄 내의 충돌 연산의 순서가 동일
- 충돌 직렬 가능 스케줄
  - 어떤 직렬 스케줄 S'과 충돌 동등한 스케줄



충돌 동등(Conflict Equivalent)

- 충돌 동등 스케줄 예





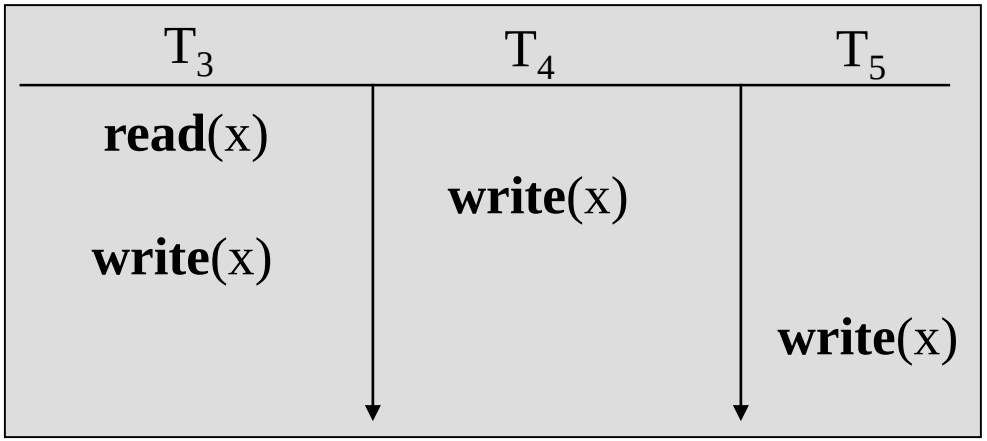
뷰 직렬가능 스케줄(1)

- 뷰 동등(S≡S')
  - i. 각 데이타 아이템 x에 대해 스케줄 S에서 x의 초기값을 T_i가 읽는다면 스케줄 S'에서도 x의 초기값을 T_i가 읽어야 함
  - ii. 스케줄 S에서 트랜잭션 T_i가 수행하는 모든 **read**(x)연산의 x값이 트랜잭션 T_j가 수행한 **write**(x) 연산으로 생성된 값이라면 스케줄 S'에서도 T_i가 수행하는 모든 **read**(x) 연산의 x값이 트랜잭션 T_j의 **write**(x) 연산으로 생성된 값이어야 함
  - iii. 스케줄 상에서 **write**(x) 연산이 수행되는 각 데이타 아이템 x에 대해, 스케줄 S에서 T_i가 **write**(x)를 마지막으로 실행하면 스케줄 S'에서도 T_i가 **write**(x)를 마지막으로 실행해야 함

- 뷰 직렬 가능 스케줄
  - 어떤 직렬 스케줄 S'과 뷰 동등한 스케줄

충돌 vs 뷰 직렬 가능 스케줄

- 충돌 직렬 가능 스케줄 ⊆ 뷰 직렬 가능 스케줄

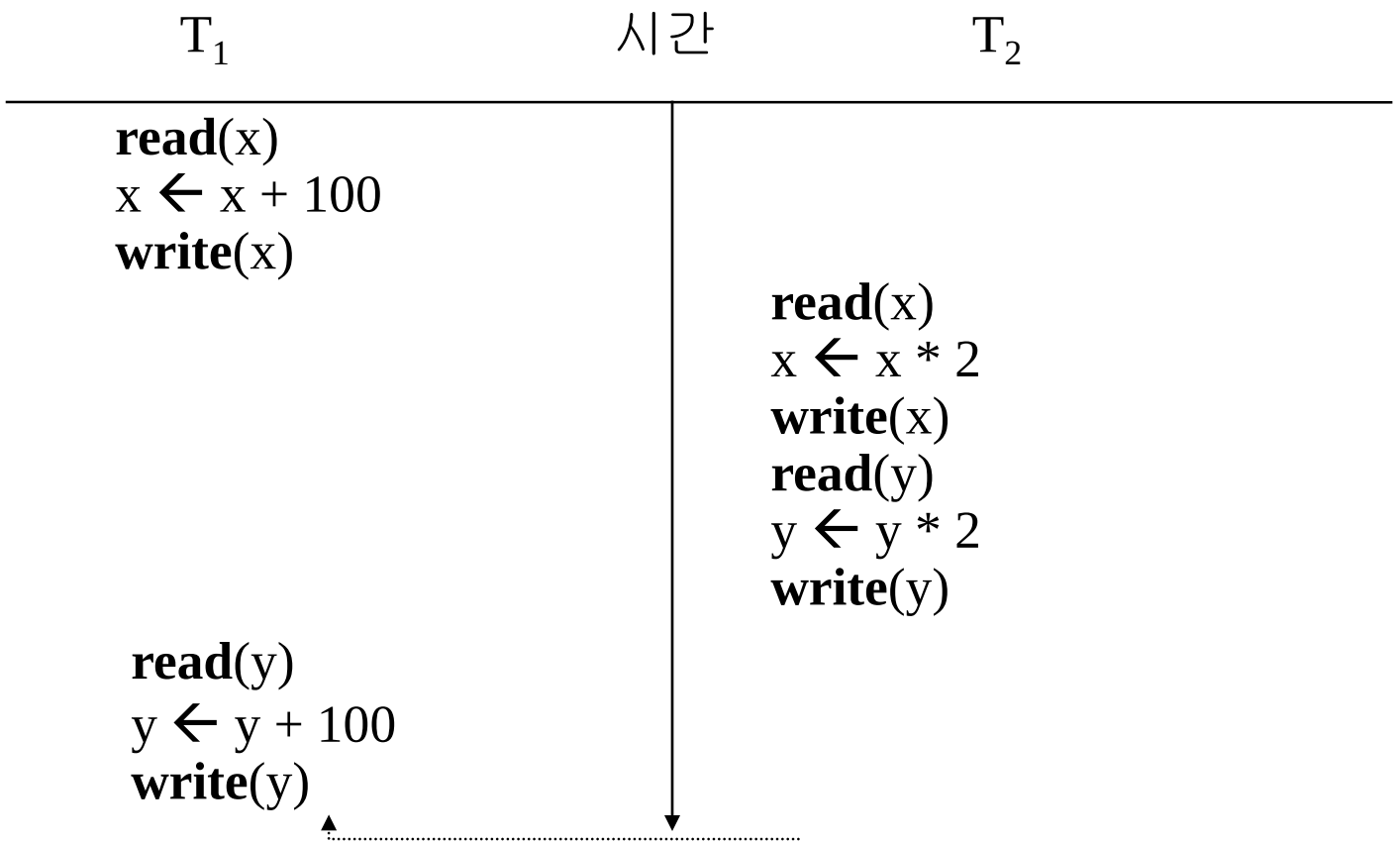


$\equiv <T_3, T_4, T_5>$

뷰 직렬 가능 스케줄

충돌 vs 뷰 직렬 가능 스케줄

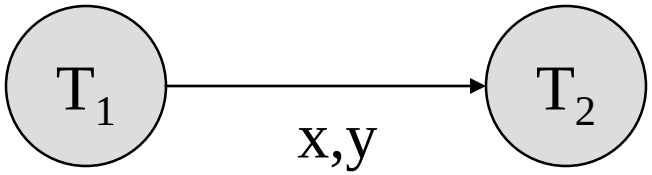
- 뷰 직렬 가능이 아닌 스케줄



직렬 가능성 검사

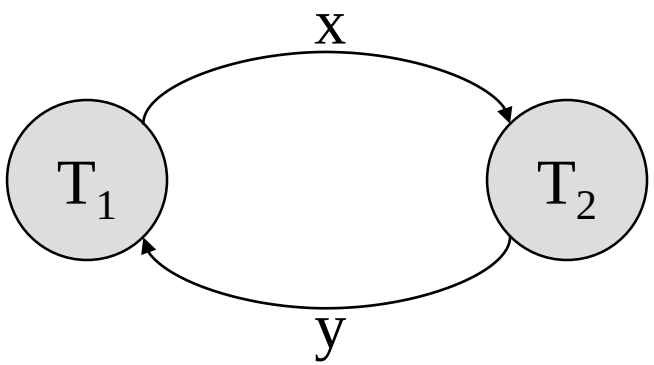
- 충돌 직렬 가능성 검사
  - 선행 그래프(precedence graph)의 구성
    - 방향 그래프 : (N,E)
    - 노드 T_i: 스케줄 S의 트랜잭션들 집합
    - 간선 : $T_i \rightarrow T_j$ 이 나타나는 경우
      - ① T_i가 **write**(x)한 x의 값을 T_j가 read(x)를 수행하는 경우
      - ② T_i가 read(x)한 뒤에 T_j가 write(x)하는 경우
      - ③ T_i가 write(x)를 한 뒤에 T_j가 **write**(x)하는 경우
  - S 충돌 직렬 가능은 선행 그래프에 사이클이 없는 경우에 가능
  - 선행 그래프에서 (선형 순서로) 위상 정렬된 모든 스케줄은 직렬 가능

▶ 예



사이클이 없음

→ 직렬 가능 스케줄



사이클이 있음

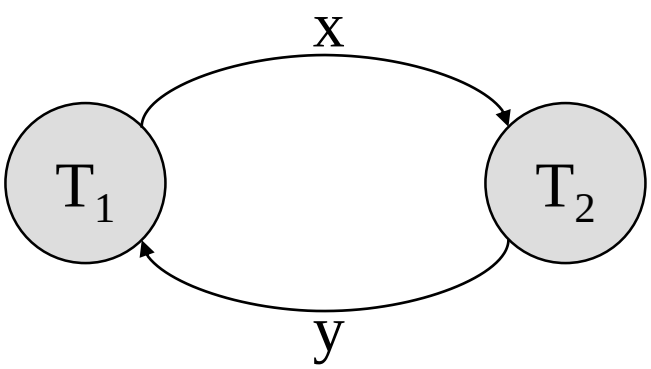
→ 직렬 불가능 스케줄

☞ Notes

- 다음 트랜잭션에 대한 아래 스케줄을 보면

T_1: $x \leftarrow x + 100$	T_2: $x \leftarrow x + 50$
$y \leftarrow y + 100$	$y \leftarrow y + 50$

T_1	시간	T_2
read(x)		
x ← x + 100		
write(x)		
		read(x)
		x ← x + 50
		write(x)
		read(y)
		y ← y + 50
		write(y)
read(y)		
y ← y + 100		
write(y)		



●결과는 직렬과 동일하나
직렬 불가능한 스케줄

직렬 가능성 이용

- 직렬 가능성 검사의 어려움
  - 트랜잭션을 임의로 수행시킨 뒤 직렬 가능성 검사
    - 직렬 가능이 안 될 때 스케줄 취소
  - 시스템에 트랜잭션들이 계속 들어올 경우
    - 어떤 스케줄이 언제 시작해서 언제 끝나는지 결정 어려움

  ⇒ 문제 복잡, 직렬 가능 검사 불가능

- 직렬 가능성 이론 이용
  - 직렬 가능성 검사 하지 않고 직렬 가능성 보장
  - 기법
    - 로킹(locking)
    - 타임스탬프(timestamp)

❖ 로킹(1)

- 정의
  - 상호 배제(독점 제어)의 과정
    다시 말해 잠금이 된 데이타 집합을 생성
    - 데이타 객체에 배타적으로 할당
    - 무간섭을 보장

- 성질

  i. 데이타 객체의 비공유, 비중첩

  ii. 부분 효과의 배제 (all or nothing)

  iii. 단일 소유자

  iv. 로크한 트랜잭션만이 로크를 해제

❖ 로킹(2)

- 로킹 규약
  - ① 트랜잭션 T가 **read**(x)나 **write**(x) 연산을 하려면 반드시 먼저 **lock**(x) 연산을 실행해야 함
  - ② 트랜잭션 T가 실행한 **lock**(x)에 대해서는 T가 모든 실행을 종료하기 전에 반드시 **unlock**(x) 연산을 실행해야 함
  - ③ 트랜잭션 T는 다른 트랜잭션에 의해 이미 **lock**이 걸려 있는 x에 대해 다시 **lock**(x)를 실행시키지 못 함
  - ④ 트랜잭션 T는 x에 **lock**을 자기가 걸어 놓지 않았다면 **unlock**(x)를 실행시키지 못 함

- 필요 조건
  - 명세 : 완전, 최소성
  - 충돌 검사 : 간단, 효과적

❖ 로킹(3)

- 로킹 모드의 확장
  - i. 공용 로크 **lock-S**
    - 공용된 접근
    - **read** 연산만 허용
  - ii. 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(4)

- 공용 로킹 규약(shared locking protocol)
  - 공용 **lock** 허용 시 따라야 하는 규약

① 트랜잭션 T가 데이타 아이템 x에 대해 **read**(x) 연산을 실행하려면 먼저 **lock-S**(x)나 **lock-X**(x) 연산을 실행해야 함

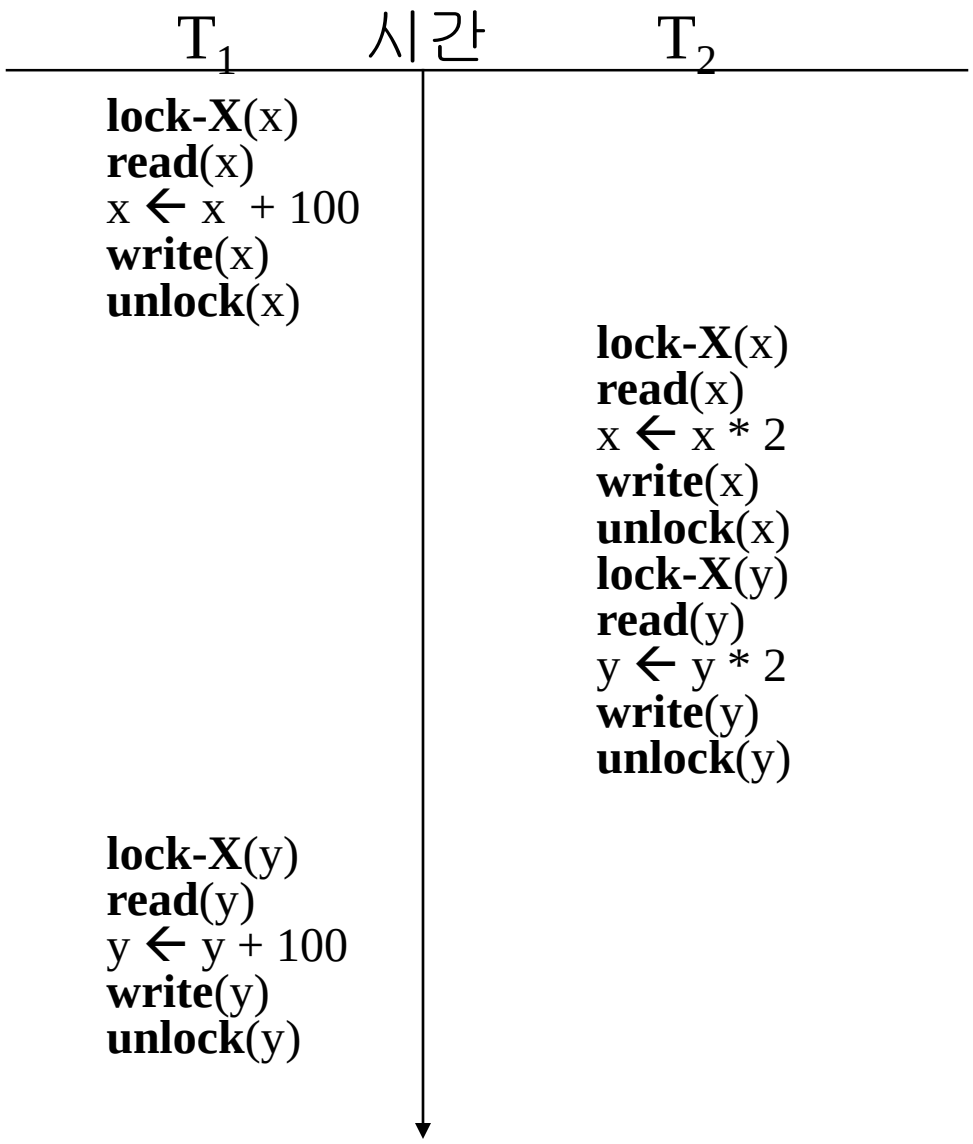
② 트랜잭션 T가 데이타 아이템 x에 대해 **write**(x) 연산을 실행하려면 먼저 **lock-X**(x) 연산을 실행해야 함

③ 트랜잭션 T가 **lock-S**(x)나 **lock-X**(x) 연산을 하려 할 때 x가 이미 다른 트랜잭션에 의해 양립될 수 없는 타입으로 **lock**이 걸려 있다면 그것이 모두 풀릴 때까지 기다려야 함

④ 트랜잭션 T가 모든 실행을 종료하기 전에는 T가 실행한 모든 **lock**(x)에 대해 반드시 **unlock**(x)를 실행해야 함

⑤ 트랜잭션 T는 자기가 **lock**을 걸지 않은 데이타 아이템에 대해 **unlock**을 실행할 수 없음

로킹 규약으로 직렬가능이 아닌 스케줄



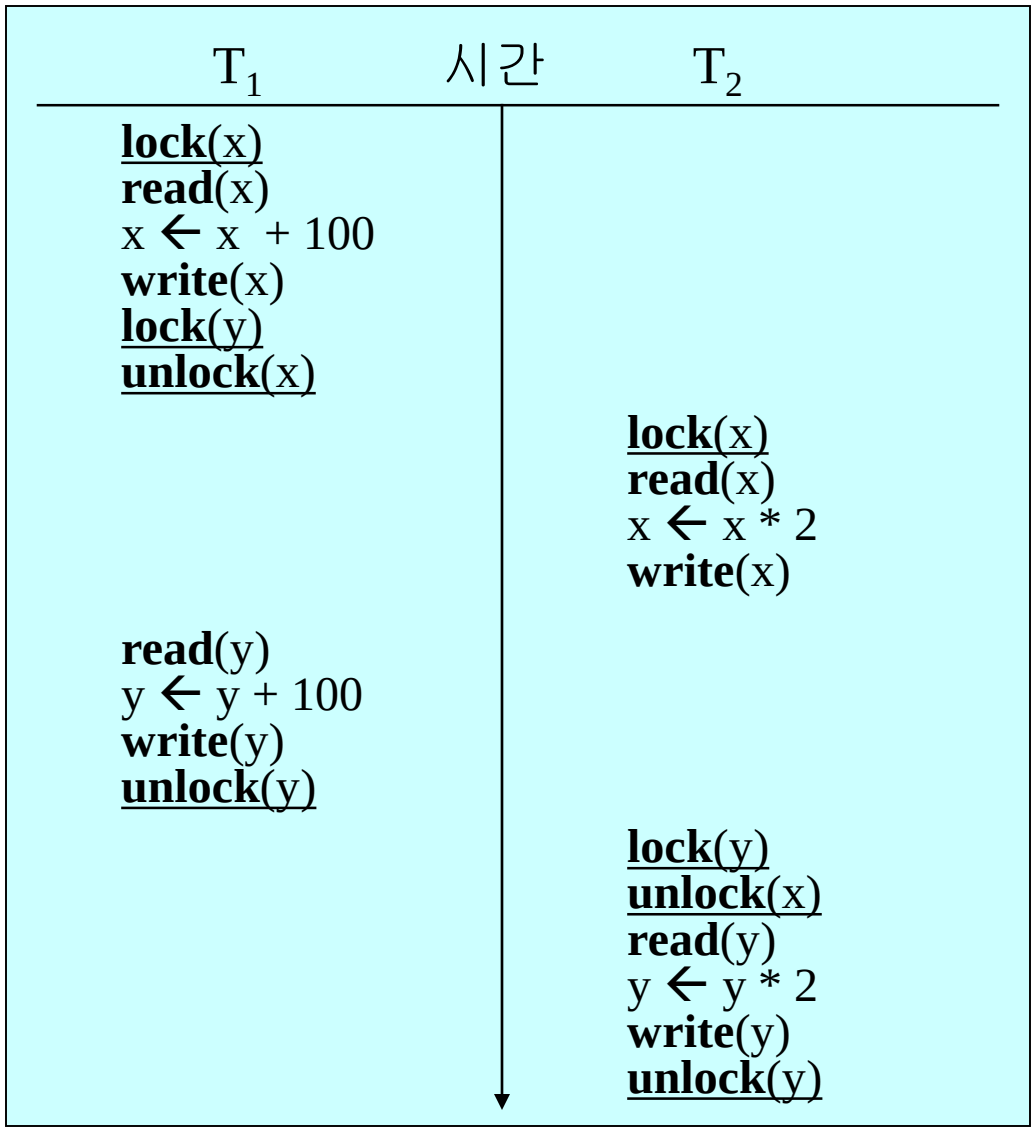
❖ 2단계 로킹 규약

- 2단계 로킹 규약(2PL)
  - ① 확장 단계(growing phase)
    트랜잭션은 **lock**만 수행하고 **unlock**은 수행할 수 없는 단계
  - ② 축소 단계(shrinking phase)
    트랜잭션은 **unlock**만 수행하고 **lock**은 수행할 수 없는 단계
- 스케줄 내의 모든 트랜잭션들이 2단계 로킹 규약을 준수한다면 그 스케줄은 직렬 가능

☞ Notes

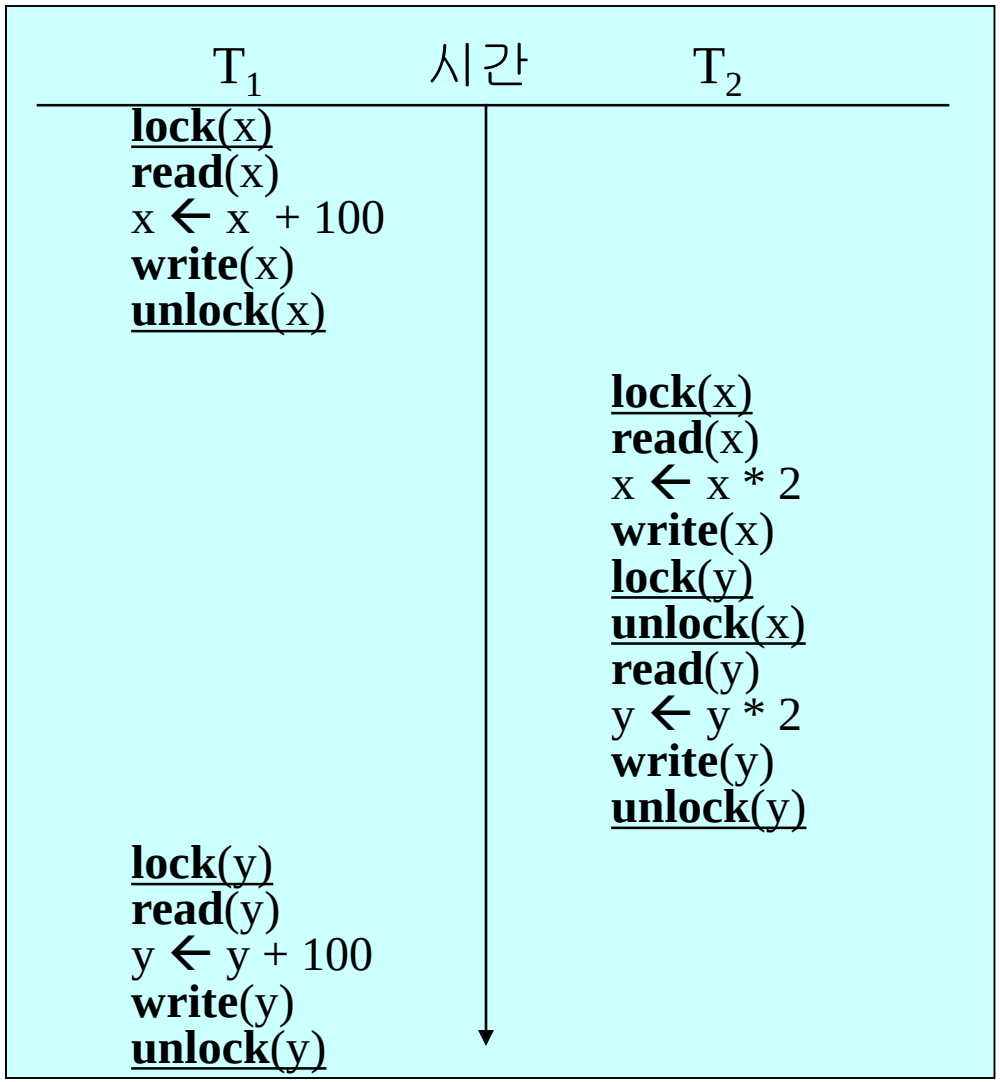
- 2단계 → 직렬 가능을 보장
- 2단계는 직렬 가능성의 충분조건이며 필요조건은 아님

▶ 예 - 2PLP로 직렬 가능 스케줄



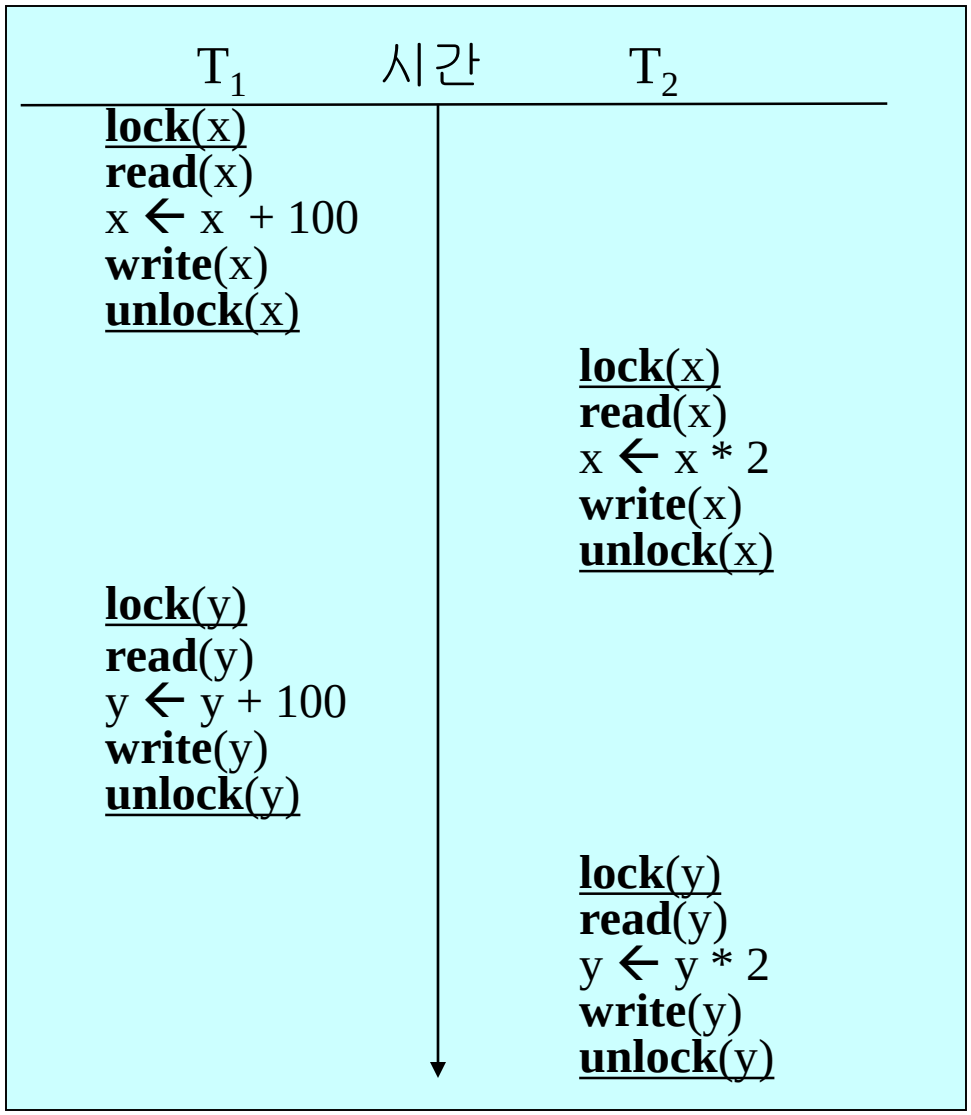
T_1, T_2 : 2단계
⇒ 직렬 가능

▶ 예 - 2PLP가 아닌 스케줄



T_1: 2단계가 아님
T_2: 2단계
⇒ 직렬 가능성을 보장하지 못함
⇒ 직렬 가능 스케줄이 아님

▶ 예 - 2PLP는 아니지만 직렬 가능 스케줄



T_1, T_2 : 2단계가 아님
⇒ 직렬 가능성을 보장하지 않음
⇒ 실제, 직렬 가능 스케줄
- 2단계는 충분조건이고 필요조건이 아님
- 2단계로는 생성되지 않는 직렬 가능 스케줄

▶ 2단계 로킹 규약의 변형

- 엄밀 2단계 로킹 규약(strict 2PLP)
  - 모든 독점 로크(**lock-X**)는 그 트랜잭션이 완료할 때 까지 **unlock**하지 않고 그대로 유지해야 함
  - 완료하지 않은 어떤 트랜잭션에 의해 기록된 모든 데이타는 그 트랜잭션이 완료할 때 까지 독점 모드로 로킹
    - 다른 트랜잭션이 그 데이타를 판독할 수 없도록 함

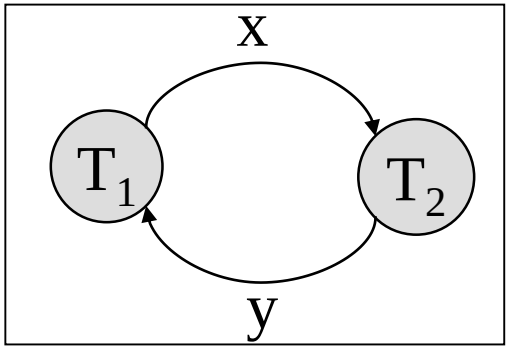
  ⇒연쇄 복귀 문제가 일어나지 않음
- 엄격 2단계 로킹 규약(rigorous 2PLP)
  - 엄밀 2PLP 보다 더 제한적
  - 모든 로크는 그 트랜잭션이 완료할 때 까지 **unlock**되지 않고 로크된 상태로 유지해야만 함
  - 트랜잭션들이 완료하는 순서로 직렬화 가능

Note!

대부분의 상용 데이타베이스 시스템 : 엄밀 2PLP나 엄격 2PLP 사용

❖ 교착 상태(Deadlock)

- 모든 트랜잭션들이 실행을 전혀 진전시키지 못하고 무한정 기다리고 있는 상태
  - T_1은 T_2가 데이타 아이템 x를 **unlock**하기 기다림
  - T_2는 T_1이 데이타 아이템 y를 **unlock**하기 기다림



- 교착상태가 발생하는 필요충분조건
  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

▶ 교착상태 예방

- 트랜잭션 스케줄링 기법
  - 트랜잭션 실행 전 필요한 데이타 아이템들을 모두 로크
  - 충돌되는 데이타를 필요로 하는 트랜잭션은 병행 실행이 아예 불가
  - 데이타 요구에 대한 사전 지식이 필요
    ⇒ 현실적으로 사용하기 어려움
  - 데이타 아이템 활용도 감소
    - 데이타 아이템이 한꺼번에 로크되기 때문
  - 기아(starvation) 문제
    - 자주 사용되는 데이타 아이템을 필요로 하는 트랜잭션은 다른 트랜잭션이 그 데이타 아이템을 쓰는 동안 무한정 기다림

- 데이타베이스에 있는 모든 데이타 아이템들에 일정한 순서를 정하는 방법
  - 모든 트랜잭션들이 이 순서에 따라 데이타 아이템을 로크하게 함
  - 프로그래머로 하여금 데이타 아이템의 순서를 숙지하도록 해야 함
    ⇒현실성 없음

▶ 교착상태의 회피(Avoidance)(1)

- 타임스탬프 이용
  - 트랜잭션의 시작 순서에 기초하는 식별자(identifier)
  - 트랜잭션이 기다려야 할지 복귀해야 할지 결정하는 데 사용
  - 트랜잭션 재시도 : T_2에 의해 로크된 x를 T_1이 요구할 때

i. **wait-die 기법** : 트랜잭션 T_i가 이미 T_j가 로크한 데이타 아이템을 요청할 때 만일 T_i의 타임스탬프가 T_j의 것보다 작은 경우(즉 **TS**(T_i) < **TS**(T_j)가 되어 T_i가 고참인 경우)에는 T_i는 기다린다. 그렇지 않으면 T_i는 복귀(즉 die)하고 다시 시작한다

ii. **wound-wait 기법** : 트랜잭션 T_i가 이미 트랜잭션 T_j가 로크한 데이타 아이템을 요청할 때 T_i의 타임스탬프가 T_j의 것보다 클 경우 (즉 **TS**(T_i) > **TS**(T_j)가 되어 T_j가 고참인 경우)에는 기다린다. 그렇지 않으면 T_j는 복귀해서 (즉 T_i는 T_j를 상처 입힌다) 다시 시작한다

☞ Note: 두 기법에서 같은 타임스탬프를 유지

▶ 교착상태의 회피(Avoidance)(2)

- **wait-die** 기법과 **wound-wait** 기법의 차이점
  - **wait-die**
    - 고참 트랜잭션이 신참 트랜잭션을 기다림
    - 고참 트랜잭션이 가지고 있는 데이타를 신참 트랜잭션이 요구하면 복귀 후 재실행
    - 재실행 시 똑같은 순서로 데이타 요구

      ⇒ 불필요한 복귀가 자주 일어날 가능성
  - **wound-wait**
    - 고참 트랜잭션은 신참 트랜잭션을 기다리지 않음
    - 신참 트랜잭션이 가지고 있는 데이타를 고참 트랜잭션이 요구하면 신참 트랜잭션은 복귀 후 재 실행
    - 신참 트랜잭션은 기다리기만 하면 됨

      ⇒ 불필요한 복귀가 비교적 덜 일어남

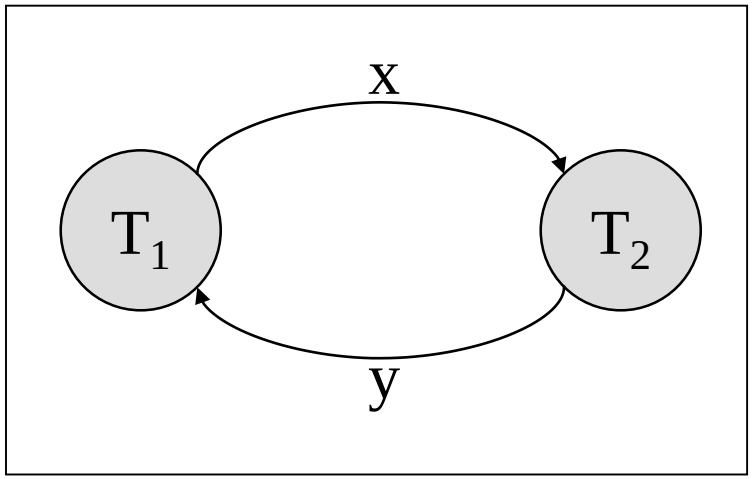
▶ 교착상태 탐지(detection)(1)

- 교착상태 방지가 보장되지 않을 시 탐지 기법 필요

- 필요 사항
  - 시스템의 정보 유지
    - 현재 로크된 데이타
    - 로크 요청이 대기 중인 데이타(pending data)
  - 시스템 검사 알고리즘
    - 교착상태 탐지 위해 주기적 가동
  - 교착상태 회복 기법

▶ 교착상태 탐지(detection)(2)

- 대기 그래프(V,E)
  - V : 트랜잭션
  - E : $(T_i \rightarrow T_j)$ T_i 가 T_j를 대기 중
  - 사이클 ⇔ 교착상태



- 교착 상태 검사 시기 기준
  - 실행 중인 트랜잭션 수 또는 데이타를 로크하기 위해 트랜잭션들이 대기하는 기간

▶ 교착상태 탐지(detection)(3)

- 회복
  - 취소할 트랜잭션 선택 : 최소 비용
  - 복귀(**rollback**) : 취소, 재시작
  - 기아(starvation) : 같은 트랜잭션이 계속 취소
    - 완료되지 못함
    - FCFS가 도움이 됨

❖ 타임스탬프 순서 기법

- 트랜잭션을 인터리브로 실행

  → 타임스탬프 순서로 직렬 가능
- 타임스탬프 순서로 각 객체를 접근하는 것을 보장
- 접근한 트랜잭션이 가장 최근에 접근한 트랜잭션보다 더 오래되었다면 새로운 타임스탬프를 재시작
- 타임스탬프 (**TS**)
  - 시스템의 클록(clock) 값 또는 논리적 카운터

▶ 타임스탬프 순서 기법

- 트랜잭션 T_i의 타임스탬프 : **TS**(T_i)
  - 시스템이 생성한 유일한 식별자
  - 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

- 데이타 아이템 x의 타임스탬프
  - **read_TS**(x)
    데이타 아이템 x의 판독시간 스탬프로서 **read**(x)를 성공적으로 수행한 트랜잭션의 타임스탬프 중에서 제일 큰 타임스탬프
  - **write_TS**(x)
    데이타 아이템 x의 기록시간 스탬프로서 **write**(x)를 성공적으로 수행한 트랜잭션의 타임스탬프 중에서 제일 큰 타임스탬프

타임스탬프 순서 규약

①T_i가 **read**(x)를 수행하려 할 때

(ㄱ) **TS**(T_i) < **write_TS**(x)이면 거부하고 T_i를 복귀

⇒ **TS**(T_i)보다 타임스탬프가 큰 어떤 트랜잭션이 T_i가 접근하기 전에 이미 x의 값을 먼저 변경시켰기 때문

(ㄴ) **TS**(T_i) ≥ **write_TS**(x)이면 실행하고

read_TS(x) ← max{ **read_TS**(x), **TS**(T_i) }

②T_i가 **write**(x)를 수행하려 할 때

(ㄱ) **TS**(T_i) < **read_TS**(x)이면 거부하고 T_i를 복귀

⇒ **TS**(T_i)보다 타임스탬프가 큰 어떤 트랜잭션이 x의 값을 먼저 판독했기 때문

(ㄴ) **TS**(T_i) < **write_TS**(x)이면 거부하고 T_i를 복귀

⇒ **TS**(T_i)보다 타임스탬프가 큰 어떤 트랜잭션이 x의 값을 먼저 기록했기 때문

(ㄷ) **TS**(T_i) ≥ **read_TS**(x)이고 **TS**(Ti) ≥ **write_TS**(x)이면 실행하고

write_TS(x) ← **TS**(T_i)

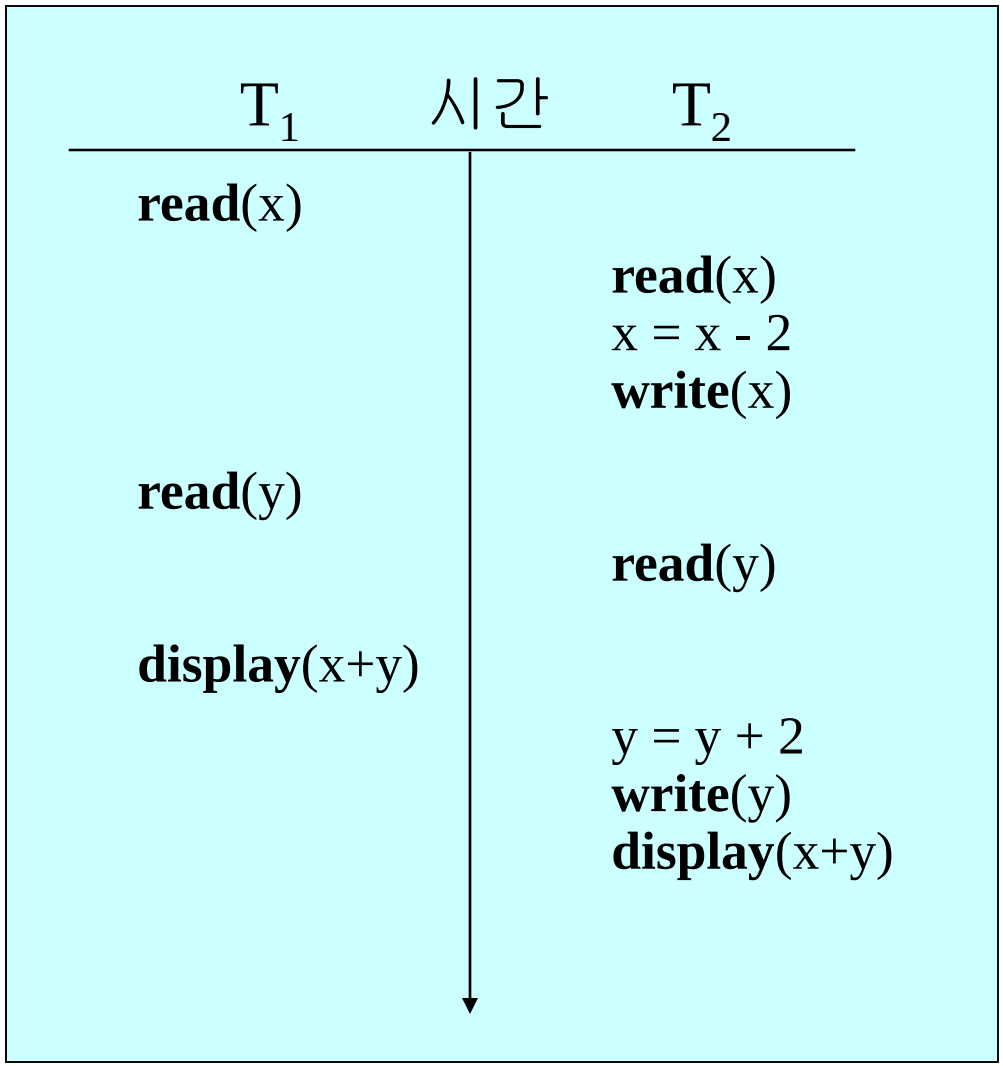
장점 vs 단점

- 교착상태가 없음 : 대기가 없기 때문
- 연쇄 복귀

  T_i의 복귀가 T_j의 복귀를 유발
- 순환적 재시작 : 기아(starvation)

  연속적인 복귀와 재시작

▶ 예제



- ◆ 타임스탬프 순서 스케줄

☞ Notes

- 2단계 로킹으로 생성 가능

▶ Thomas의 기록 규칙(revised TS protocol) (1)

- 잠재적 병행성의 증대

- **Write** 규칙의 수정(no change in read rule)

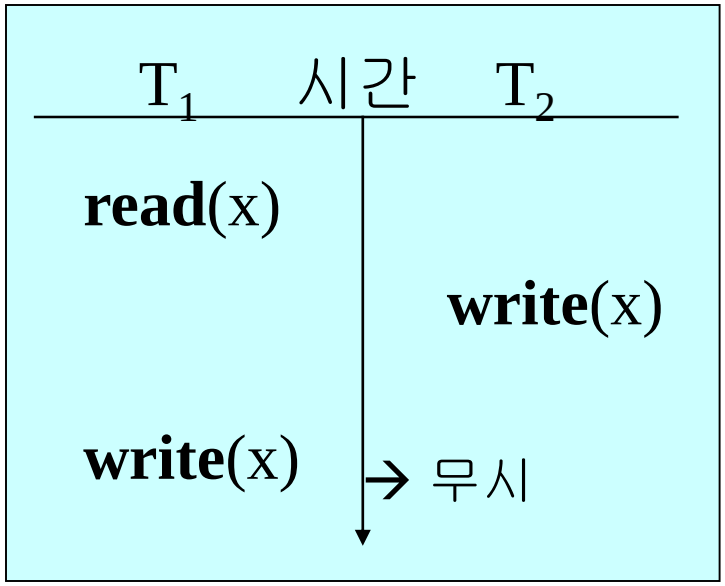
  T_i가 **write**(x)를 수행하려 할 때 :

  만일 **TS**(T_i) < **read_TS**(x) 이면 **write**(x)
  를 거부하고 T_i를 취소시켜 복귀시킨다

  만일 **TS**(T_i) ≥ **read_TS**(x) 이고
  TS(T_i) < **write_TS**(x) 이면
  write(x)를 단순히 무시한다

  그 이외의 경우는 **write**(x)를 허용하고
  write_TS(x) ← **TS**(T_i)

▶ Thomas의 기록 규칙 (2)



TS(T_1) >=**read_TS**(x) and
TS(T_1) < **write_TS**(x)

- 기존의 타임스탬프 규약에서는 T_1의 **write** 연산이 거부되면 T_1이 복귀
- Thomas의 기록 규칙은 무용의 **write** 연산을 무시하고 스케줄이 직렬 가능

▶ 다중 버전 병행 제어 (1)

- 각 데이타 아이템 x에 대해 여러 버전 $< x_1, x_2, \ldots, x_m >$ 이 시스템에 의해 유지

- 각 버전 x_k의 값은 다음과 같은 두개의 타임스탬프와 함께 저장
  - **write_TS(x_k)**: 버전 x_k를 생성한 트랜잭션의 타임스탬프
  - **read_TS(x_k)**: 버전 x_k를 성공적으로 판독한 트랜잭션 중에서 제일 큰 타임스탬프

- 기본 아이디어
  - **write**(x): 성공적이면 x의 새로운 버전을 생성
  - **read**(x): x의 버전 중 하나를 읽음, 직렬가능

▶ 다중 버전 병행 제어 (2)

- 다중 버전 타임스탬프 순서 규약
  - 트랜잭션 T_i가 **read**(x)나 **write**(x)연산을 요청하고, x_k에 대해서 **write_TS**(x_k)($\leq$ **TS**(T_i))가 제일 크다고 가정하자

$x = <x_1, x_2, \ldots, x_m>$

① T_i가 **read**(x)를 요청하면,
버전 x_k의 값을 판독,
read_TS(x_k)= MAX{**read_TS**(x_k), **TS**(T_i)}
② T_i가 **write**(x)를 요청하면,
if **TS**(T_i) < **read_TS**(x_k) then 복귀
else 새버전의 x_{m+1}를 생성
write_TS(x_{m+1}) = **read_TS**(x_{m+1})
= **TS**(T_i)로 설정

- 연쇄 복귀의 가능성
- **read** 연산 : 결코 실패하거나 기다리지 않음(never fail, never wait)

▶ 다중 버전 병행 제어 (3)

- 알고리즘 ①
  - 트랜잭션은 항상 가장 최신 버전의 데이타를 판독

- 알고리즘 ②
  - 트랜잭션이 너무 뒤늦게 기록하려고 하면 강제로 복귀
  - 트랜잭션이 어떤 버전의 데이타를 기록하려고 할 때 그 데이타를 다른 트랜잭션이 판독해 버린 경우 → 기록 불허

다중 버전 병행 제어 (4)

- 더 이상 필요가 없는 버전들의 삭제
  - 데이타 아이템 x에 대해 두 버전 x_j 와 x_k이 존재
  - 조건 : 이 두 버전의 **write_TS** < 시스템에서 가장 오래된 트랜잭션의 타임스탬프
  - 결과 : x_j 와 x_k 중에서 더 오래된 버전 삭제 가능

❖ 낙관적 병행 제어 (1)

- 대부분의 트랜잭션이 읽기 전용

  → 검사 시 오버헤드

- 트랜잭션의 실행을 3단계로 나눔

  1). 판독 단계(R)

  - 지역 변수만을 이용 읽기와 갱신 수행

  2). 확인 단계(V)

  - 실제 데이타베이스에 반영하기 전에 충돌 직렬 가능성 검사

  3). 기록 단계(W)

  - 확인 단계를 통과하면 트랜잭션의 실행결과는 실제로 데이타베이스에 반영. 그렇지 않으면 트랜잭션은 취소되고 재시작

❖ 낙관적 병행 제어 (2)

- 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) $\Rightarrow$ <T_i,T_j>

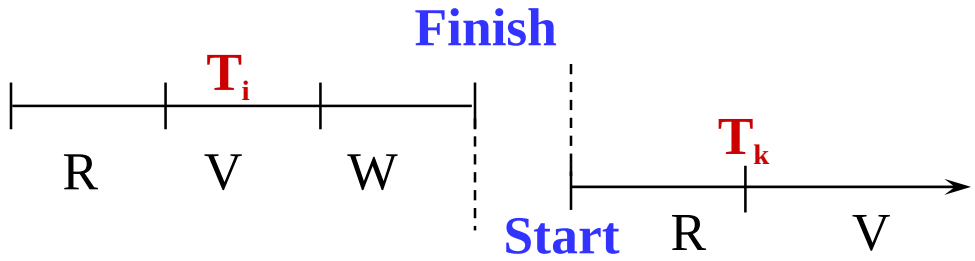
❖ 낙관적 병행 제어 (3)

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

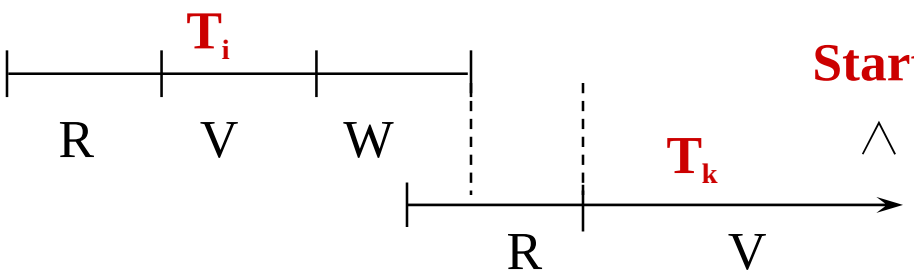
① **Finish**(T_i) < **Start**(T_k)
T_i가 T_k 시작 전에 완성
② **Start**(T_k) < **Finish**(T_i) < **Validation**(T_k)
and **Write-set**(T_i) ∩ **Read-set**(T_k) = ∅
③ **Validation**(T_i) < **Validation**(T_k)
and **Write-set**(T_i) ∩ **Read-set**(T_k) = ∅
and **Write-set**(T_i) ∩ **Write-set**(T_k) = ∅

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

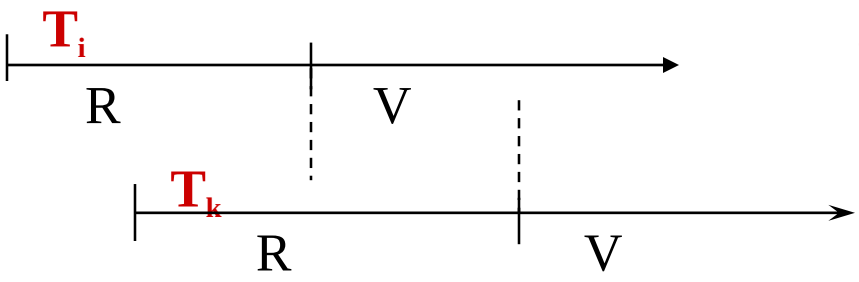
확인 검사 조건 : $TS(T_i) < TS(T_k)$

① 

$Finish(T_i) < Start(T_k)$

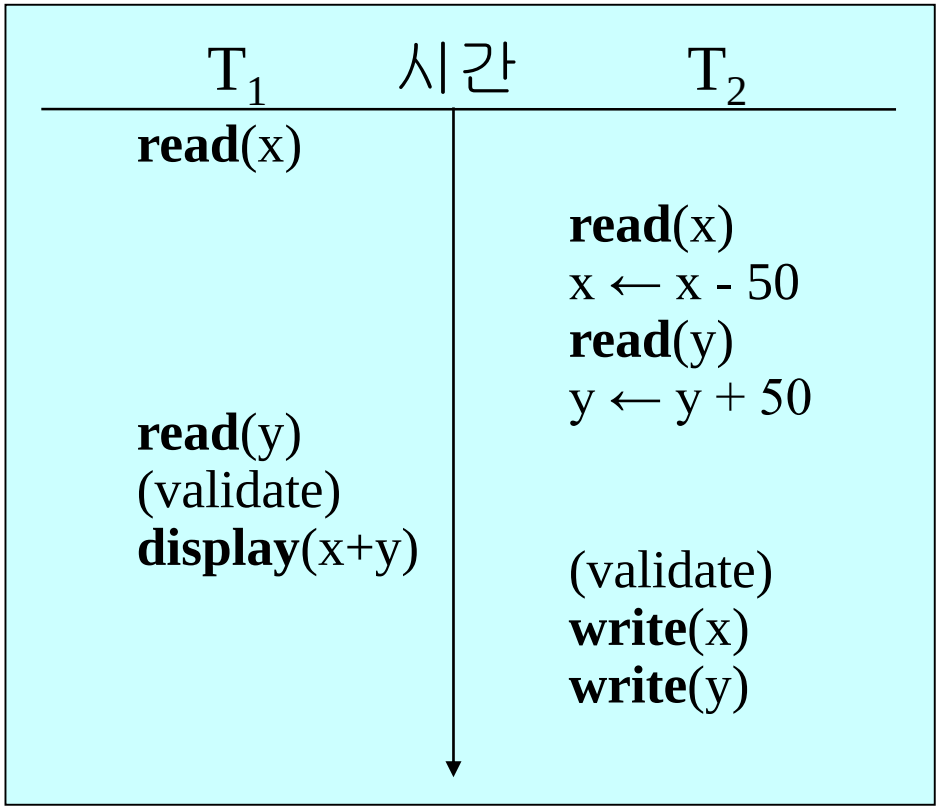
② 

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

③ 

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

예제



- 낙관적 기법에 의한 직렬 가능 스케줄

☞ Note: 2PL이나 타임스탬프 기법으로 생성 불가능

▶ 팬텀 충돌(Phantom Conflict)

T_1 : **SELECT SUM**(Sal)
FROM PROFESSOR
WHERE Dept = 'COMP ENG'
T_2 : **INSERT INTO** PROFESSOR(Pno, Pname, Dept, Sal)
VALUES('P123', 'LEE', 'COMP ENG', 200)

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 팬텀 투플에 대해 T_1과 T_2 가 서로 충돌되기 때문

☞ Note: 오직 투플 단위에서만 적용

해결책

- 해결책 : 팬텀이 아닌 실제 데이타의 충돌을 유도

i. 로킹 단위를 크게 하는 것 :

로킹의 대상이 되는 데이타의 단위를 투플이 아니라 릴레이션으로 한다

ii. 인덱스 로킹 기법

- 릴레이션과 그것의 인덱스를 갱신하는 것을 의미
- 단위 : index record 또는 index bucket

인덱스 로킹 기법

① 모든 릴레이션은 적어도 하나의 인덱스를 가지고 있어야 한다

② 트랜잭션 T_i는 접근하려는 릴레이션의 투플 t에 대한 포인터가 있는 인덱스 버켓에 **S**형 로크를 걸었을 때에만 그 투플 t에 대해 **S**형 로크를 걸 수 있다

③ 트랜잭션 T_i는 갱신하려는 릴레이션의 투플 t에 대한 포인터가 있는 인덱스 버켓에 **X**형 로크를 걸었을 때에나 그 투플 t에 대해 **X**형 로크를 걸 수 있다

④ 트랜잭션 T_i는 투플을 삽입하기 전에 릴레이션의 모든 인덱스를 갱신하여야 하고 갱신하려는 모든 인덱스 버켓에 **X**형 로크를 걸어야 한다

⑤ 로킹은 2단계 로킹 규약에 따라야 한다

▶ 삽입 / 삭제 연산과 병행 제어 (1)

insert(x) : x 새로 생성

delete(x) : x 이미 존재

◆ 양립성

T_i \ T_j	read	write	delete	insert
insert	X	X	X	X
delete	X	X	X	X

X : 충돌

▶ 삽입 / 삭제 연산과 병행 제어 (2)

{ **read**$_i$(x) **insert**$_j$(x)	: 논리적 오류
{ **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 **write**연산으로 취급
  - 2단계 로킹 규약에서는 전용 **lock**을 사용
  - 타임스탬프 순서 규약에서도 **write** 연산으로 취급