トランザクション管理

参考文献

Jeffrey D. Ullman: *Principles of Database and Knowledge-base Systems*
Volume I, Computer Science Press, 1988. ISBN 0-7167-8158-1

Chapter 9　Transaction Management

トランザクション

データの読み出しや書き込みからなる処理の単位
並列に実行される

トランザクション管理の重要性

更新が必要なデータベース
- 座席予約システム
- ダブルブッキングの回避
- 高速な処理時間よりもデータ更新の安全性を確保したい

読み出すことが中心のデータベース
- 統計データベース
- WWWデータベース
- 並列な読み出し要求を数多く処理したい

例題　普通預金口座　A

T1: 1000円を振り込むトランザクション

```
READ A;
A := A + 1000;
WRITE A;
```

T2: 1000円を引き出すトランザクション

```
READ A;
A := A − 1000;
WRITE A;
```

スケジュール：　時間軸に沿ってステートメントを並べた列

	A
	4000
T1: READ A	
T1: A := A+1000	
T1: WRITE A	5000
T2: READ A	
T2: A := A−1000	
T2: WRITE A	4000

順番に実行するスケジュール

	A
	4000
T1: READ A	
T1: A := A+1000	
T2: READ A	
T2: A := A−1000	
T1: WRITE A	5000
T2: WRITE A	3000

おかしな結果！

例題
普通預金口座　A, B

T1:　AからBへ
2000円送金

```
READ A;
IF A >= 2000
    A := A - 2000;
WRITE A;

READ B;
B := B + 2000;
WRITE B;
```

T2:　BからAへ
5000円送金

```
READ A;
A := A + 5000;
WRITE A;

READ B;
IF B >= 5000
    B := B - 5000;
WRITE B;
```

T1, T2 を実行する過程で A と B の値が変化する様子

	A	B
	4000	4000
T1	2000	
T1		6000
T2	7000	
T2		1000

	A	B
	4000	4000
T2	9000	
T1	7000	
T1		6000
T2		1000

	A	B
	4000	4000
T2	9000	
T1	7000	
T2		abort

abort: 実行不能でトランザクションを消滅させる操作
T2の影響を元に戻す必要あり

トランザクションの原子性（Atomicity）の保証

整列スケジュールでは各トランザクションを
原子のような塊として実行する。

原子性を保証するとは、
各トランザクションを並列に実行するとき、最終結果が、
ある整列スケジュールの
結果と一致することを保証すること。

例

	A	B
	4000	4000
T2	9000	
T1	7000	
T2		abort
T2復帰	7000−5000	
	=2000	
T1		6000

	A	B
	4000	4000
T1	2000	
T1		6000

整列スケジュール

問題

T1: 1000円を振り込むトランザクション

```
READ A;
A := A + 1000;
WRITE A;
```

T2: 1000円を引き出すトランザクション

```
READ A;
A := A − 1000;
WRITE A;
```

A の値が変化する様子を示せ

```
                    A
                    4000
T2: READ A
T2: A := A−1000
T1: READ A
T1: A := A+1000
T2: WRITE A
T1: WRITE A
```

```
                    A
                    4000
T2: READ A
T2: A := A−1000
T1: READ A
T1: A := A+1000
T1: WRITE A
T2: WRITE A
```

T1: AからBへ
2000円送金

```
READ A;
IF A >= 2000
    A := A - 2000;
WRITE A;

READ B;
B := B + 2000;
WRITE B;
```

T2: BからAへ
5000円送金

```
READ A;
A := A + 5000;
WRITE A;

READ B;
IF B >= 5000
    B := B - 5000;
WRITE B;
```

問題　A, B の値が変化する様子を示せ

	A	B
	4000	4000

```
T1: READ A;
T1: IF A >= 2000
T1:     A := A - 2000;
T2: READ A;
T2: A := A + 5000;
T2: WRITE A;
T1: WRITE A;

T1: READ B;
T1: B := B + 2000;
READ B;
T2: IF B >= 5000
T2:     B := B - 5000;
T2: WRITE B;
T1: WRITE B;
```

Lock

データベースの更新単位をアイテム、
アイテムの大きさを粒度（granularity）と呼ぶ

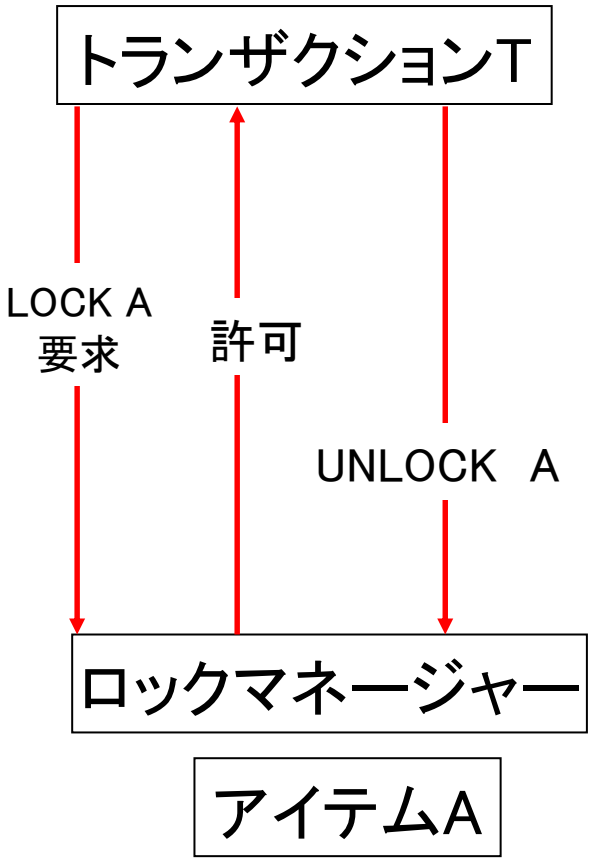
アイテムへのアクセスはロックにより管理
排他的なロックのかかっているアイテムには
他のトランザクションのアクセスを禁止

アイテムの粒度が大きい場合、
ロック管理システムの負担は軽くなるが、
同時に実行できるトランザクション数は減る

粒度の選択は典型的トランザクションを考えてきめる
預金口座のように小さい単位でトランザクションが
実行される場合は粒度は小さくする

ロックは様々な種類があるが当面は一種類とし、
アイテムAに対するロックを　LOCK　A　と記述

- トランザクション T はアイテム A に READ/WRITEする前に　LOCK　A　をロックマネージャーに要求
- ロックマネージャーはアイテム A にロックがかかっていない場合に T が LOCK Aすることを許可
- Aがロックされている間は他のトランザクションはAにアクセスできない
- TはAに対する処理を終了したら、ロックマネージャーに　UNLOCK　A　を要求し、ロックマネージャーはアイテムAに対するロックを解除



例題　普通預金口座　A

T1: 1000円を振り込むトランザクション

```
LOCK A;
READ A;
A := A + 1000;
WRITE A;
UNLOCK A;
```

T2: 1000円を引き出すトランザクション

```
LOCK A;
READ A;
A := A − 1000;
WRITE A;
UNLOCK A;
```

	A
	4000
T1: LOCK A	
T1: READ A	
T1: WRITE A	5000
T1: UNLOCK A	
T2: LOCK A	
T2: READ A	
T2: WRITE A	4000
T2: UNLOCK A	

	A
	4000
T2: LOCK A	
T2: READ A	
T2: WRITE A	3000
T2: UNLOCK A	
T1: LOCK A	
T1: READ A	
T1: WRITE A	4000
T1: UNLOCK A	

	A
	4000
T1: LOCK A	
T1: READ A	
~~T2: LOCK A~~	
~~T2: READ A~~	

問題

T1: 1000円を振り込むトランザクション

```
LOCK A;
READ A;
A := A + 1000;
WRITE A;
UNLOCK A;
```

T2: 1000円を引き出すトランザクション

```
LOCK A;
READ A;
A := A − 1000;
WRITE A;
UNLOCK A;
```

LOCK の利用により下のスケジュールが生み出す問題点が回避されることを示せ

	A
	4000
T2: READ A	
T2: A := A−1000	
T1: READ A	
T1: A := A+1000	
T2: WRITE A	
T1: WRITE A	

各トランザクションは
一つのアイテムに高々一回だけLOCKをかける

T
LOCK A;
A := A+1;
UNLOCK A;
LOCK B;
LOCK A;
B := B+A;
UNLOCK A;
UNLOCK B;

T
LOCK A;
A := A+1;
LOCK B;
B := B+A;
UNLOCK A;
UNLOCK B;

Livelock

トランザクション T_i

```
LOCK A;
READ A;
A := A + i;
WRITE A;
UNLOCK A;
```

T1: LOCK A;	T1が最初にAをロック
T1: READ A;	
T1: A := A+1;	
T1: WRITE A;	T2, T3 が　LOCK A　を要求
T1: UNLOCK A;	
T3: LOCK A;	ロックマネージャーがT3にロックを許可
T3: READ A;	
T3: A := A+1;	T4 が　LOCK A　を要求 T2 は待ち続けている
T3: WRITE A;	
T3: UNLOCK A;	
T4: LOCK A;	ロックマネージャーが T4 にロックを許可
T4: READ A;	
T4: A := A+1;	
T4: WRITE A;	
T4: UNLOCK A;	

簡単な回避方法（First-come-first-served strategy）
LOCK　A　を要求するトランザクションに要求順にロックを許可

Deadlock

```
T1:  AからBへ
2000円送金

LOCK A;
LOCK B;

READ A;
IF A >= 2000
    A := A - 2000;
WRITE A;
READ B;
B := B + 2000;
WRITE B;

UNLOCK A;
UNLOCK B;
```

```
T2:  BからAへ
5000円送金

LOCK B;
LOCK A;

READ A;
A := A + 5000;
WRITE B;
READ B;
IF B >= 5000
    B := B - 5000;
WRITE B;

UNLOCK B;
UNLOCK A;
```

T1: LOCK A;　許可
T2: LOCK B;　許可

T1: LOCK B;　不許可
T2: LOCK A;　不許可

T1とT2は永遠に
待ちつづける

プロトコルを工夫してDeadlock を起こさない手法1

各トランザクションは同時にすべてのロック要求を出す
ロックマネージャーは、この全要求を許可するか、全てを不許可

```
T1:  AからBへ
2000円送金

LOCK A;
LOCK B;
READ A;
IF A >= 2000
     A := A - 2000;
WRITE A;
READ B;
B := B + 2000;
WRITE B;
UNLOCK A;
UNLOCK B;
```

```
T2:  BからAへ
5000円送金

LOCK B;
LOCK A;
READ A;
A := A + 5000;
WRITE B;
READ B;
IF B >= 5000
     B := B - 5000;
WRITE B;
UNLOCK B;
UNLOCK A;
```

T1: LOCK A　許可
T1: LOCK B　許可

T1: UNLOCK A
T1: UNLOCK B

T2: LOCK B　許可
T2: LOCK A　許可

プロトコルを工夫してDeadlock を起こさない手法2

- アイテムに順序を導入

A ＞ B

- 各トランザクションは
  順序の大きい順に
  アイテムをロック

T1: AからBへ
2000円送金

LOCK A;
LOCK B;
:

T2: BからAへ
5000円送金

LOCK B;
LOCK A;
:

T2: BからAへ
5000円送金

LOCK A;
LOCK B;
:

T1の実行後にT2が処理される

アイテムに順序を入れる方法がDeadlock を起こさない理由

背理法： Deadlock 状態のトランザクション集合があると仮定

前処理： アイテムをロックしていないトランザクションを
除いても、集合は依然として Deadlock状態

観察： 各トランザクション T_i は他のトランザクション T_{i+1} が
ロックしているアイテム A_i の開放を待っている

$T_i \xrightarrow[A_i]{} T_{i+1}$

注意
T_i は A_i より順序の
大きいアイテムをロック

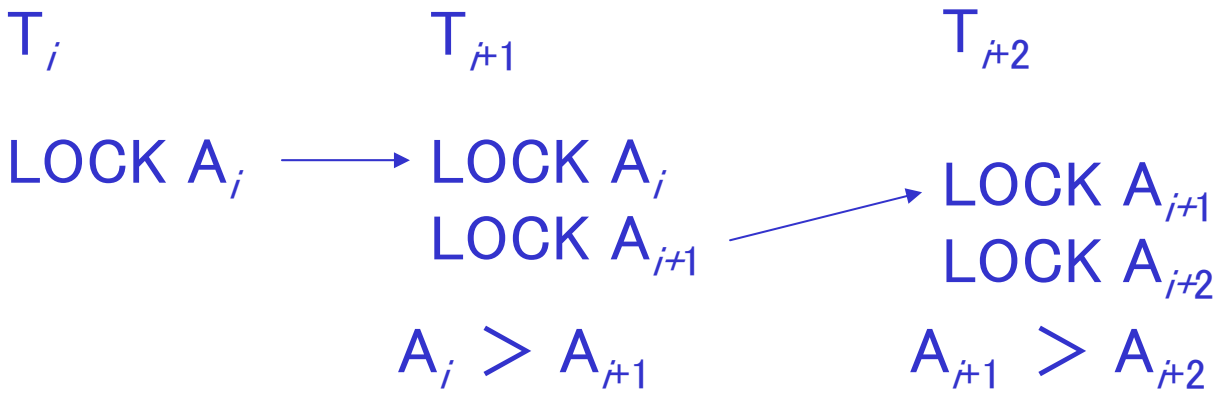
T_i	T_{i+1}	T_{i+2}
LOCK A_i →	LOCK A_i	
	LOCK A_{i+1} →	LOCK A_{i+1}
		LOCK A_{i+2}
	$A_i > A_{i+1}$	$A_{i+1} > A_{i+2}$

ある T_1 を選択して、鎖を構成

$A_1 > A_2 > \cdots\cdots > A_{n-1}$

ループしない有限の下降鎖

T_n は,いつか A_{n-1} を UNLOCK.
アイテムをロック
していない T_n が必ず
存在し、矛盾



スケジューラーがDeadlock を検知して abort 等で対処する方法

Waits-for Graph

トランザクションがノード

T1がロックしているアイテムA
のUNLOCKをT2が待っている
とき、「T1の次はT2」を表現する
ためT1からT2へ有向辺を引き
Aをラベルとしてつける

閉路の存在は Deadlockを意味

たとえば　T2 を abort して B への
ロックを開放し、T1 へ渡す

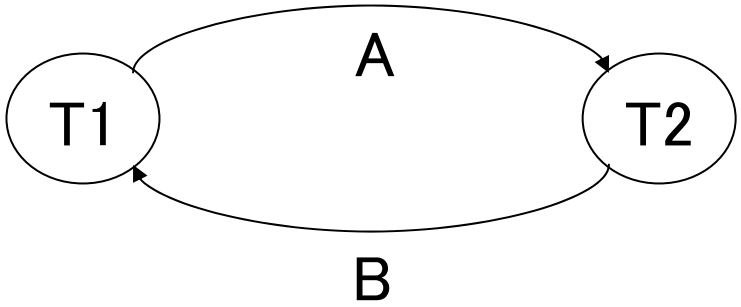
```
T1:  AからBへ
2000円送金

LOCK A;
LOCK B;
:
UNLOCK A;
UNLOCK B;
```

```
T2:  BからAへ
5000円送金

LOCK B;
LOCK A;
:
UNLOCK B;
UNLOCK A;
```

T1: LOCK A;　許可
T2: LOCK B;　許可



問題

T1	T2	T3
LOCK A	LOCK B	LOCK C
LOCK B	LOCK C	LOCK A
A := A+B;	B := B+C;	C := C+A;
UNLOCK A	UNLOCK B	UNLOCK C
UNLOCK B	UNLOCK C	UNLOCK A

1. LOCK文をアイテムのアルファベッド順で並べる方式により、deadlock を回避することを考える。T1, T2, T3 から同時にロック要求があり、最初に T3 が実行が認められたとする。2番目、3番目に実行されるトランザクションとスケジュールを述べよ。

2. Waits-for graph により deadlock を解消することを考える。T1, T2, T3 から同時にロック要求があり、wait-for graph に閉路ができたとき、T3 を abort したとする。1番目、2番目に実行されるトランザクションを述べよ。

整列化可能性
(Serializability)

トランザクションの集合をある順序で整列化した列を

$T_1\ T_2\ \ldots\ldots\ T_k$

i=1,2,…,k の順番で T_i のステートメントを実行する
スケジュールを整列スケジュール (serial schedule) と呼ぶ

トランザクションをプログラムしたときのユーザの心境は
各トランザクションの実行を他のトランザクションが
邪魔しないことを仮定している

つまりトランザクション全体がある整列スケジュールとして
実行されることを思ってプログラムする

しかしシステムは効率を重視してトランザクションの
順番を入れ替えて実行するかもしれない

入れ替えをしても、その実行結果は
ある整列スケジュールの実行結果と一致してほしい

スケジュールが整列化可能とは、実行結果が
ある整列スケジュールの実行結果と一致することと定義する

この概念をどのように定式化するか？

T1: Aのx倍をBへ振り込む

```
LOCK A;
LOCK B;
READ A;
READ B;
B := B+A * x;
WRITE B;
UNLOCK A;
UNLOCK B;
```

T2: Bのx倍をAへ振り込む

```
LOCK B;
READ B;
UNLOCK B;
LOCK A;
READ A;
A := A+B * x;
WRITE A;
UNLOCK A;
```

T2: LOCK B;
T2: READ B;
T2: UNLOCK B;

T1: LOCK A; LOCK B;
T1: READ A; READ B;
T1: B := B+A * x; WRITE B;
T1: UNLOCK A; UNLOCK B;

T2: LOCK A;
T2: READ A;
T2: A := A+B * x;
T2: WRITE A;
T2: UNLOCK A;

整列スケジュール

T1 B:=B+A*x
T2 A:=A+B*x

T2 A:=A+B*x
T1 B:=B+A*x

A=B=1000
x=2

T1: B=3000
T2: A=7000

T2: A=3000
T1: B=7000

T2: B=1000
T1: B=3000
T2: A=3000

整列化不可能

A=0
B=1000
x=2

T1: B=1000
T2: A=2000

T2: A=2000
T1: B=5000

T2: B=1000
T1: B=1000
T2: A=2000

整列化可能？

スケジュール S が整列化可能を、「どのような初期状態に対しても、
S と実行結果が一致する整列スケジュールが存在すること」
と定義すると困る…

初期状態の候補は無限にあり、有限ステップでチェックできるか？
帰納的関数がある場合、2つの帰納的関数の等価性は一般に
決定不能

スケジュール S が整列化可能とは、
どのような初期状態に対しても、どのような実行文に対しても、
S と実行結果が一致する整列スケジュールは存在すること。

どのような実行文に対しても等価となる整列スケジュールは存在するか？

T1: Aのx倍をBへ振り込む

LOCK A;
LOCK B;

UNLOCK A;
UNLOCK B;

T2: Bのx倍をAへ振り込む

LOCK B;

UNLOCK B;
LOCK A;

UNLOCK A;

整列スケジュール

T2: LOCK B;
T2: UNLOCK B;

T1: LOCK A;
T1: LOCK B;
T1: UNLOCK A;
T1: UNLOCK B;

T2: LOCK A;
T2: UNLOCK A;

T1: LOCK A;
T1: LOCK B;
T1: UNLOCK A;
T1: UNLOCK B;

T2: LOCK B;
T2: UNLOCK B;
T2: LOCK A;
T2: UNLOCK A;

T2: LOCK B;
T2: UNLOCK B;
T2: LOCK A;
T2: UNLOCK A;

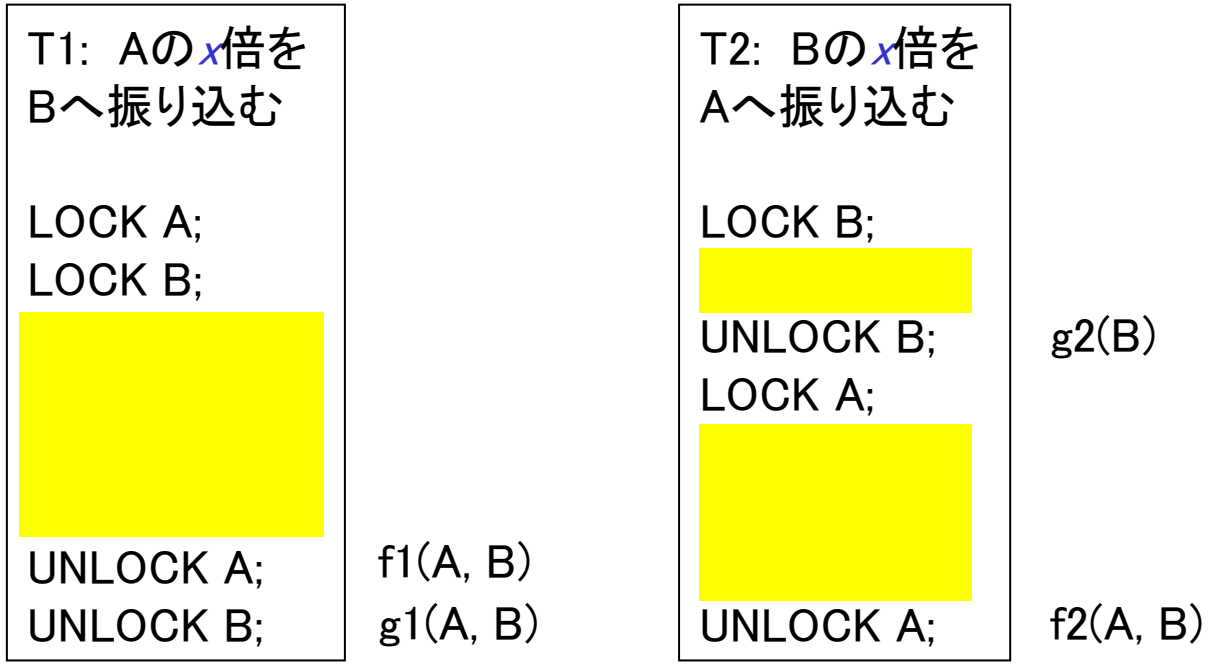
T1: LOCK A;
T1: LOCK B;
T1: UNLOCK A;
T1: UNLOCK B;

LOCK A と UNLOCK A
のペアに対して
新しい関数 f を割当てる

f の引数は UNLOCK A
の前にトランザクション中で
ロックされる全ての
アイテムの値

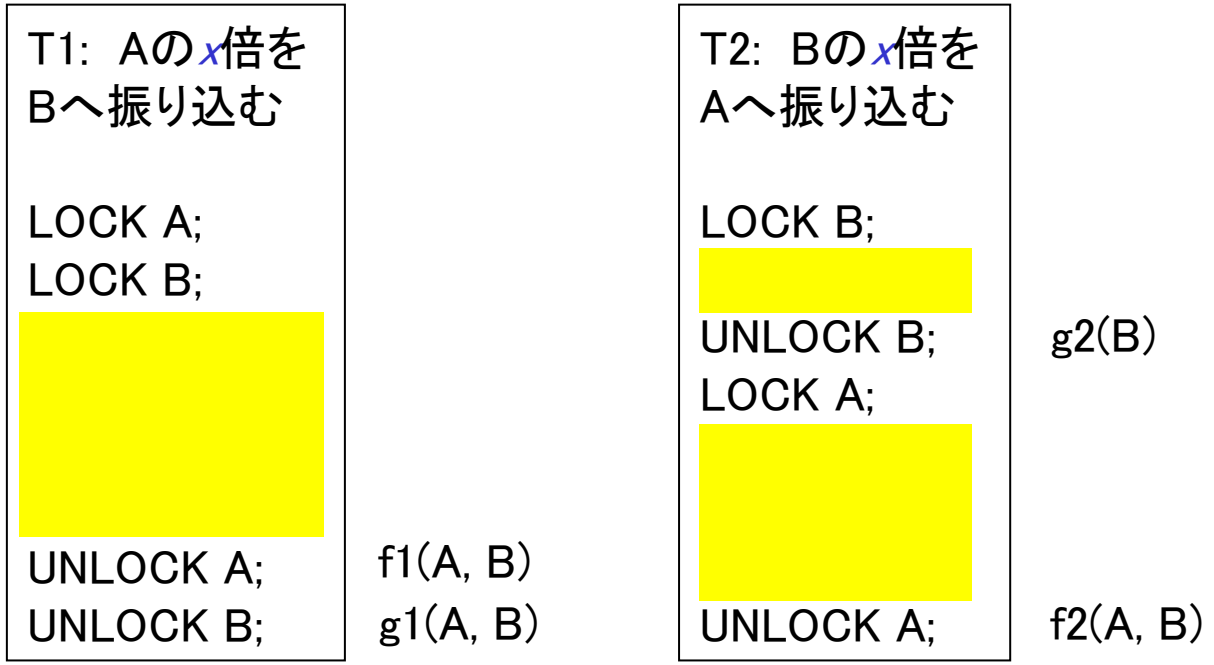
T1: Aの*x*倍をBへ振り込む		T2: Bの*x*倍をAへ振り込む	
LOCK A;		LOCK B;	
LOCK B;			
		UNLOCK B;	g2(B)
		LOCK A;	
UNLOCK A;	f1(A, B)		
UNLOCK B;	g1(A, B)	UNLOCK A;	f2(A, B)

		A	B
T2: LOCK B;		a	b
T2: UNLOCK B;	g2(B)		g2(b)
T1: LOCK A;			
T1: LOCK B;			
T1: UNLOCK A;	f1(A,B)	f1(a, g2(b))	
T1: UNLOCK B;	g1(A,B)		g1(f1(a, g2(b)), g2(b))
T2: LOCK A;		f2(f1(a,g2(b)), g2(b))	
T2: UNLOCK A;	f2(A,B)	~~f2(f1(a,g2(b)),~~	
		~~g1(f1(a, g2(b)), g2(b)))~~	



整列スケジュール1

	A	B
T1: LOCK A;	a	b
T1: LOCK B;		
T1: UNLOCK A;	f1(a,b)	
T1: UNLOCK B;		g1(f1(a,b), b)
T2: LOCK B;		
T2: UNLOCK B;		g2(g1(f1(a,b), b))
T2: LOCK A;		
T2: UNLOCK A;	f2(f1(a,b), g2(g1(f1(a,b), b)))	



整列スケジュール2

	A	B
T2: LOCK B;	a	b
T2: UNLOCK B;		g2(b)
T2: LOCK A;		
T2: UNLOCK A;	f2(a, g2(b))	
T1: LOCK A;		
T1: LOCK B;		
T1: UNLOCK A;	f1(f2(a,g2(b)), g2(b))	
T1: UNLOCK B;		g1(f1(f2(a,g2(b))), g2(b))

3つのスケジュールの最終結果

	A	B
	~~f2(f1(a,g2(b)),~~ ~~g1(f1(a, g2(b)), g2(b)))~~	
	f2(f1(a,g2(b)), g2(b))	g1(f1(a, g2(b)), g2(b))
整列スケジュール1	f2(f1(a,b), g2(g1(f1(a,b), b)))	g2(g1(f1(a,b), b))
整列スケジュール2	f1(f2(a,g2(b)), g2(b))	g1(f1(f2(a,g2(b))), g2(b))

記号列として異なる

記号列はUNLOCKの実行履歴を表現している

- 与えられたスケジュールと等価な
  整列スケジュールが見つかれば整列可能

- 全ての整列スケジュールを生成して検査する
  のは計算コストが大きい

  n個のトランザクションからは整列スケジュールが n! 個

問題

T1: LOCK B
T1: UNLOCK B

T2: LOCK A
T2: UNLOCK A
T2: LOCK B
T2: UNLOCK B

T3: LOCK A
T3: UNLOCK A
T3: LOCK B
T3: UNLOCK B

上の3つのトランザクションからなる以下の2つのスケジュールが整列可能か否か、UNLOCK に関数を割り当てる手法により調べよ

T2: LOCK A
T2: UNLOCK A
T3: LOCK A
T3: UNLOCK A
T1: LOCK B
T1: UNLOCK B
T2: LOCK B
T2: UNLOCK B
T3: LOCK B
T3: UNLOCK B

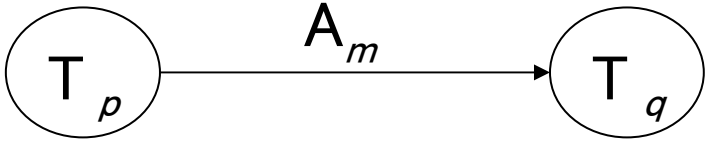
T1: LOCK B
T1: UNLOCK B
T2: LOCK A
T2: UNLOCK A
T2: LOCK B
T2: UNLOCK B
T3: LOCK A
T3: UNLOCK A
T3: LOCK B
T3: UNLOCK B

整列化可能性を判定するアルゴリズム

スケジュールの整列化可能性を判定するアルゴリズム

スケジュールは　T: LOCK A または T: UNLOCK A
の形をしたステートメントの列 $a_1, a_2, \ldots, a_n$ とする

整列化可能性判定グラフの構成
各 $a_i = T_p$: UNLOCK A_m について $a_j = T_q$: LOCK A_m （$p \neq q$）
となる $j > i$ が存在する場合、最小の j を選択し有向辺を張る



整列化可能な場合に対応する
整列スケジュール中で T_p は T_q の前にある

T_p を始点としA_mをラベルにもつ有向辺は高々1つ

定理　グラフに閉路がなければ整列化可能、あれば不可能

整列化可能性判定グラフ
閉路がない場合

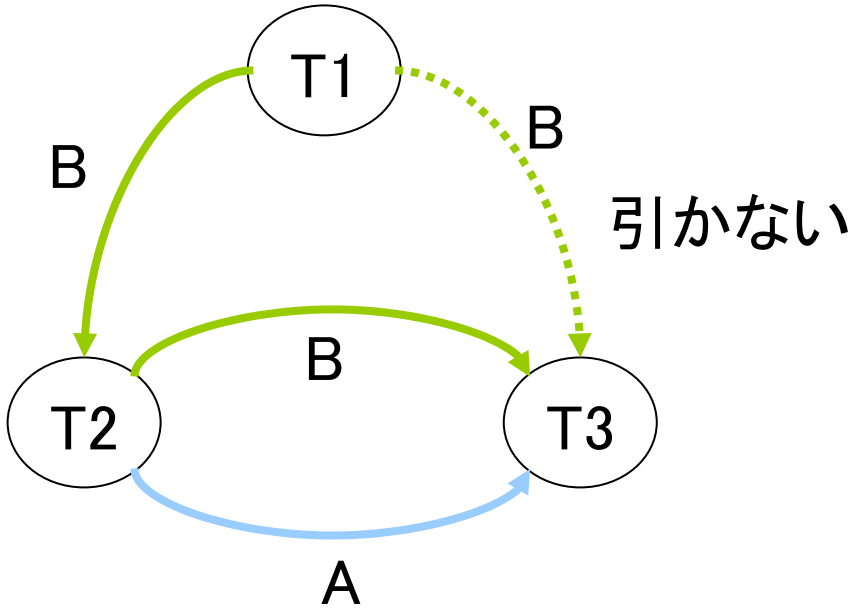
T2: LOCK A
T2: UNLOCK A

T3: LOCK A
T3: UNLOCK A

T1: LOCK B
T1: UNLOCK B

T2: LOCK B
T2: UNLOCK B

T3: LOCK B
T3: UNLOCK B



この関係は
考慮しない

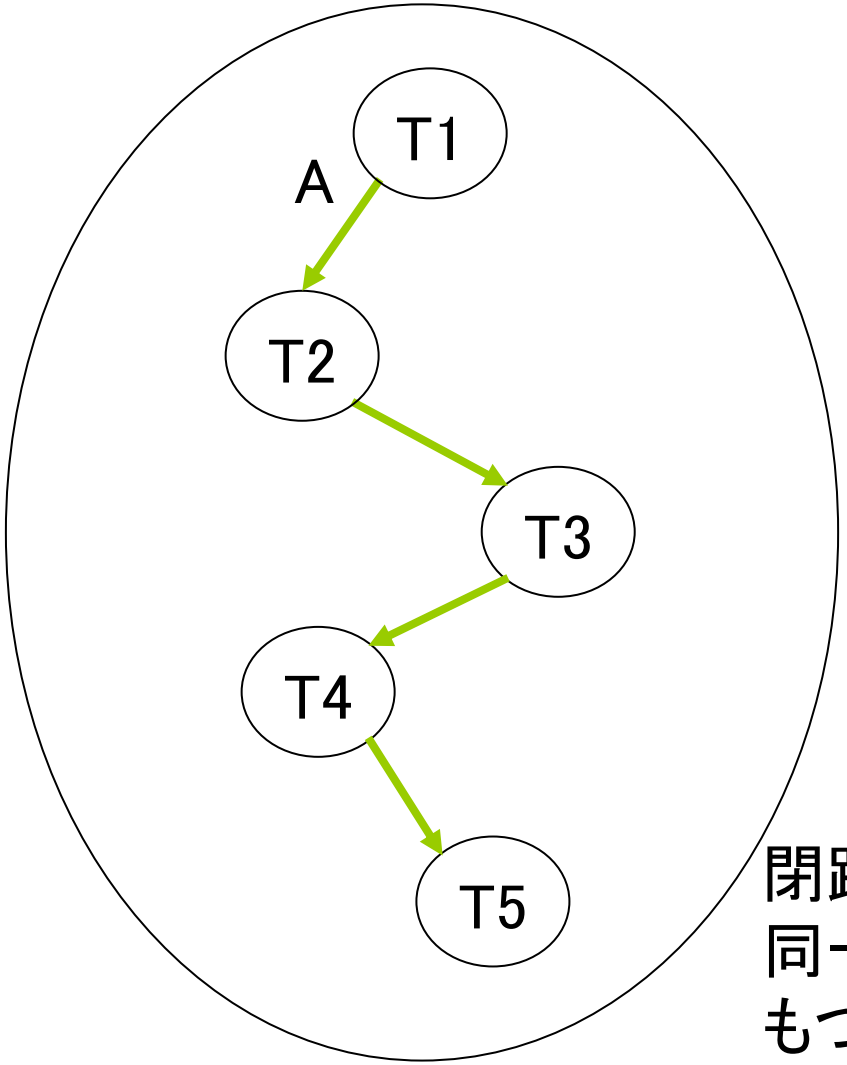
任意の始点から出る有向辺で
同一ラベルをもつ辺は高々一つ

各トランザクションは一つのアイテムに高々
一回だけLOCKをかける

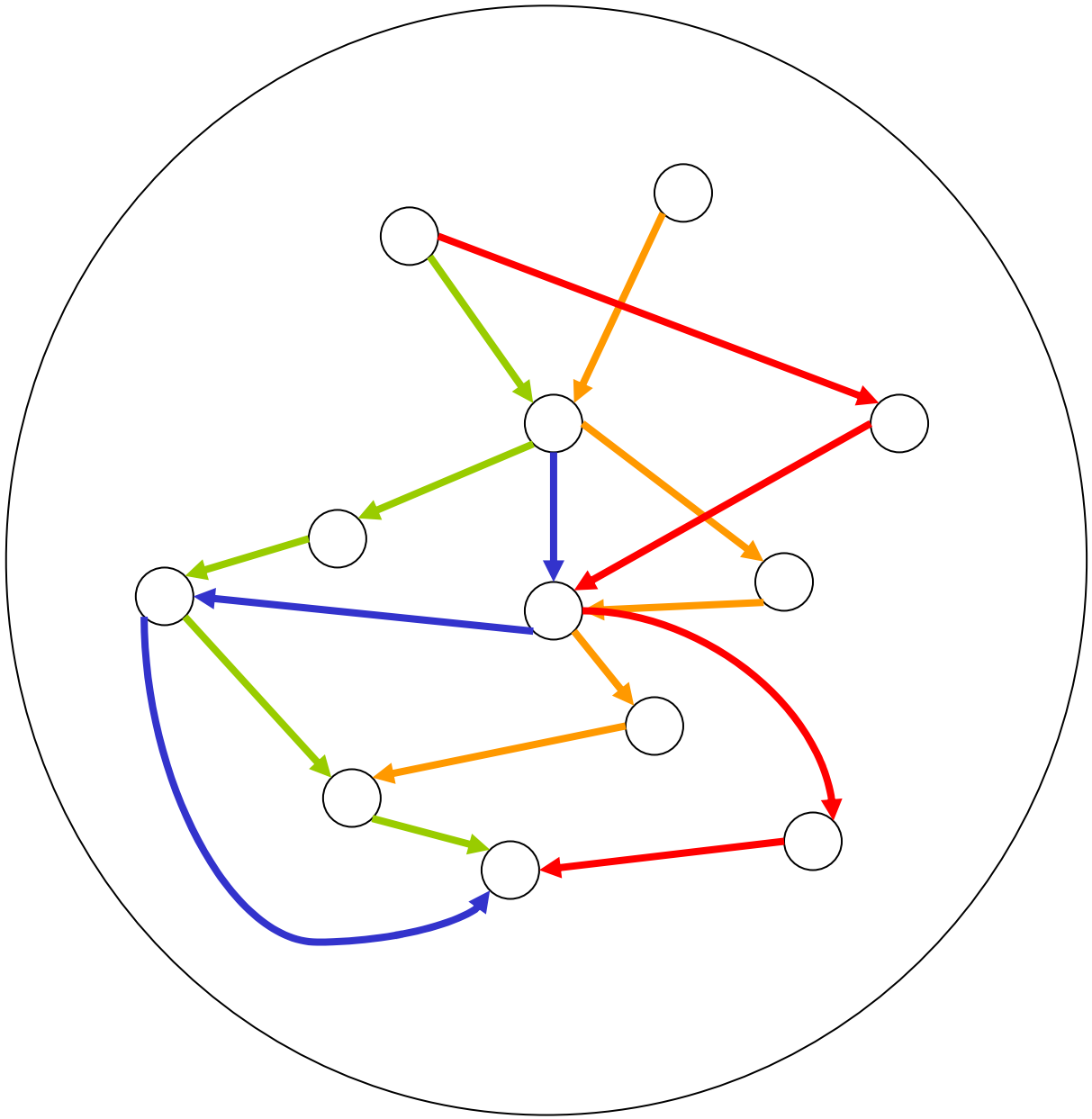
整列化可能性判定グラフ
閉路がない場合

与えられた
スケジュール

T1: LOCK A
T1: UNLOCK A
:
T2: LOCK A
T2: UNLOCK A
:
T3: LOCK A
T3: UNLOCK A
:
T4: LOCK A
T4: UNLOCK A
:
T5: LOCK A
T5: UNLOCK A



閉路のないグラフでは、
同一アイテムをラベルに
もつ有向辺は、連続した
列をつくる



一般には複数の
アイテムに対する
列が交差する

整列化可能性判定グラフ G の位相ソート

閉路がない場合には
Gにはいかなる有向辺の終点でないノード v が存在する

存在しないと仮定すれば、あるノードから有向辺を逆に辿る列をつくると、ノード数が有限なので、いつかは自分に戻る閉路ができる

v と v を始点とする有向辺を G から除く

このステップを繰返し、除かれたノード（トランザクション）列
T_1 T_2 …… T_k を位相ソート列と呼ぶ

位相ソート列を使って整列スケジュールを作る

与えられたスケジュール

T2: LOCK A
T2: UNLOCK A

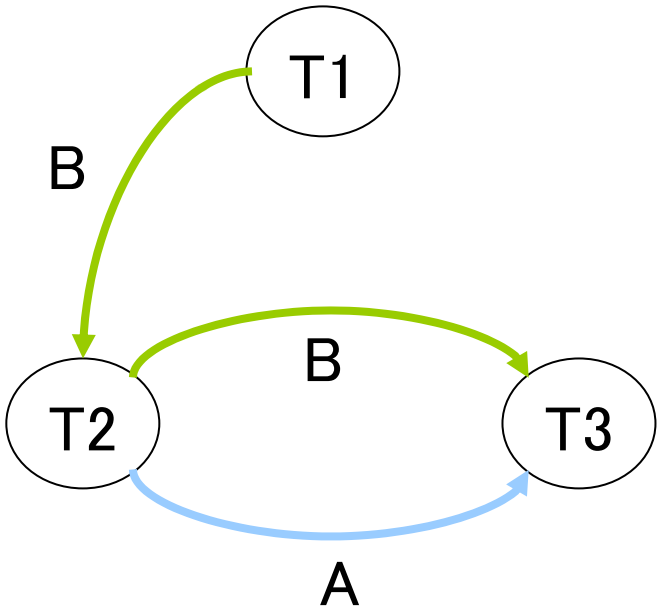
T3: LOCK A
T3: UNLOCK A

T1: LOCK B
T1: UNLOCK B

T2: LOCK B
T2: UNLOCK B

T3: LOCK B
T3: UNLOCK B

整列化可能性判定グラフ



位相ソート列　T1, T2, T3

整列スケジュール

T1: LOCK B
T1: UNLOCK B

T2: LOCK A
T2: UNLOCK A
T2: LOCK B
T2: UNLOCK B

T3: LOCK A
T3: UNLOCK A
T3: LOCK B
T3: UNLOCK B

問題

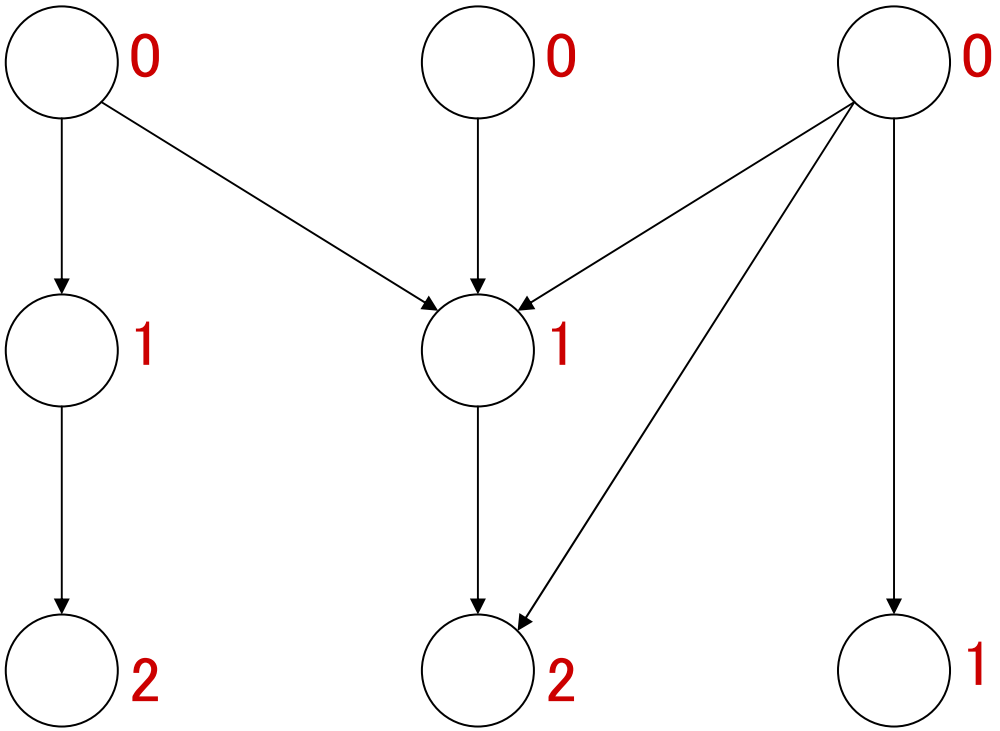
整列化可能性判定グラフを用いて以下の
スケジュールが整列化可能か否か求めよ

T2: LOCK B;
T2: UNLOCK B;
T1: LOCK A;
T1: LOCK B;
T1: UNLOCK A;
T1: UNLOCK B;
T3: LOCK A;
T3: UNLOCK A;
T2: LOCK A
T2: UNLOCK A

- 与えられたスケジュールの整列可能性を判定することは容易になった
- 現実にはトランザクションは動的に追加され、スケジュールは単調増加する
- 増加に応じて整列化可能性判定グラフを生成して閉路の存在検査をするべきか？　閉路が発見されたらそれまでの計算をやり直すのか？
- 整列化可能となるスケジュールを必ず生成できるトランザクションのプロトコル（生成規則）は考えられないか？

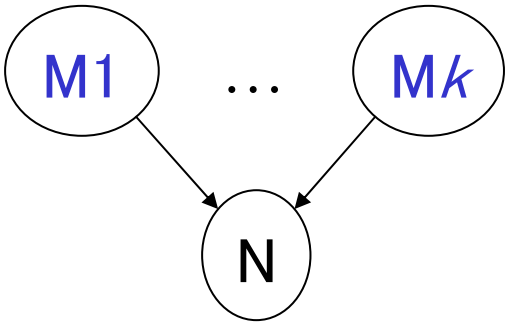
定理前半
スケジュール S の整列化可能性判定グラフ G が閉路を含まない場合、位相ソートでつくった整列スケジュール R は S と等価

準備



ノード N の深さ d(N)

N を終点とする有向辺の始点を N の隣接点

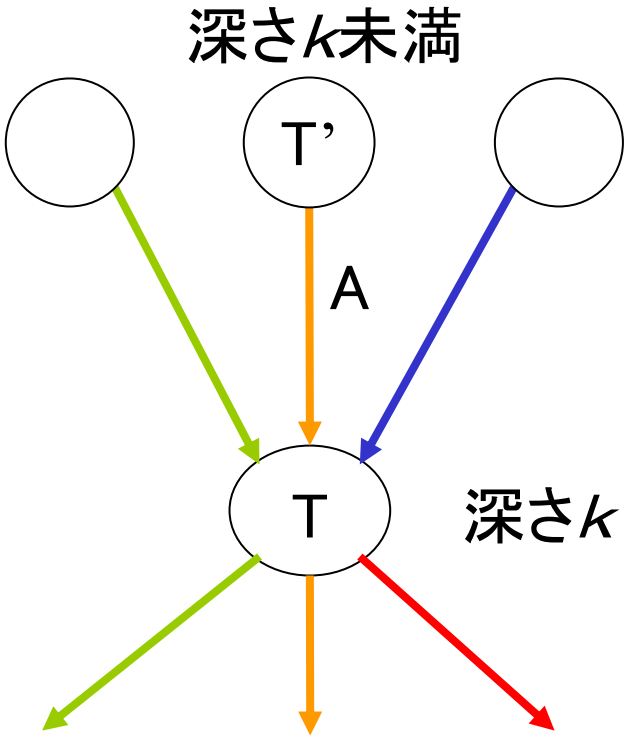


d(N) = 0　　Nに隣接点がない
= max{ d(M) | M は隣接点} + 1

スケジュール S と R で、同じトランザクションは
同一のアイテムに関して同じ値を読むことを証明

深さに関する帰納法

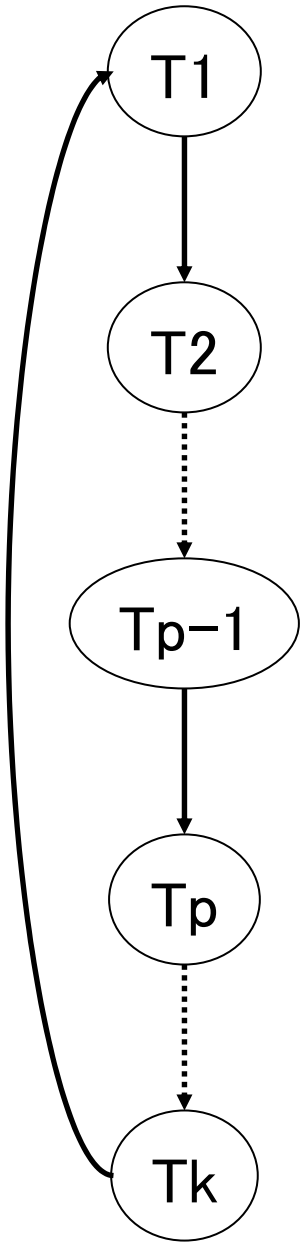
深さ0のとき、各アイテムの値は初期状態だから、OK



S, R で T がアイテム A を
LOCKする直前に A を UNLOCK
するトランザクションをT'とする

帰納法の仮定でS と R は
T' で A に同一の値を読み込む

S と R は T でも A に
同一の値を読み込む



定理後半　スケジュール S の整列化可能性判定
グラフが閉路を含む場合、Sは整列化不可能

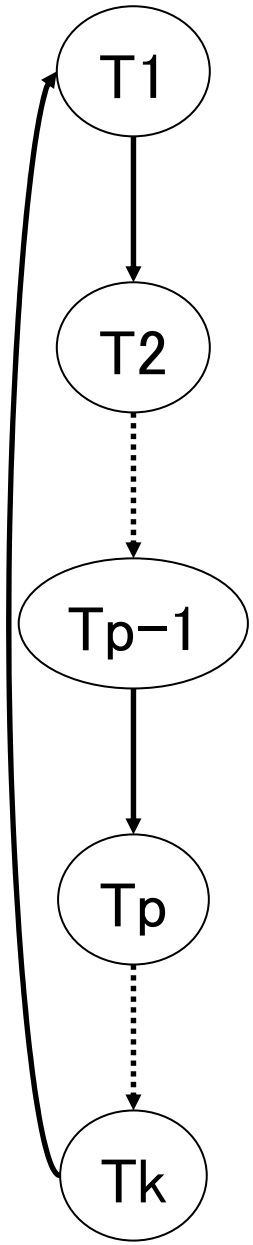
仮に等価な整列スケジュール R があるとする

Ti (i=1,…k)の中で Tp が R に最初に現れるとする

S	R
Tp−1: UNLOCK A　g(…)	
Tp: LOCK A	Tp: LOCK A
:	:
Tp: UNLOCK A　f(…)	Tp: UNLOCK A　f(…)

R では f(…) に g(…) は出現しない．Sでは出現.

R と S は等価でないので矛盾



もう一つ例

S		R	
Tp: LOCK A			
Tp−1: LOCK B			
Tp−1: UNLOCK B	g(B)		
		Tp: LOCK A	
Tp: LOCK B		Tp: LOCK B	
:		:	
Tp: UNLOCK A	f(A,B)	Tp: UNLOCK A	f(A,B)

R では f(…) に g(…) は出現しない．Sでは出現．

R と S は等価でないので矛盾

2相ロックプロトコル
(Two-phase Locking Protocol)

整列化可能となるスケジュールを必ず生成するトランザクションの生成規則(プロトコル)

２相ロックプロトコル

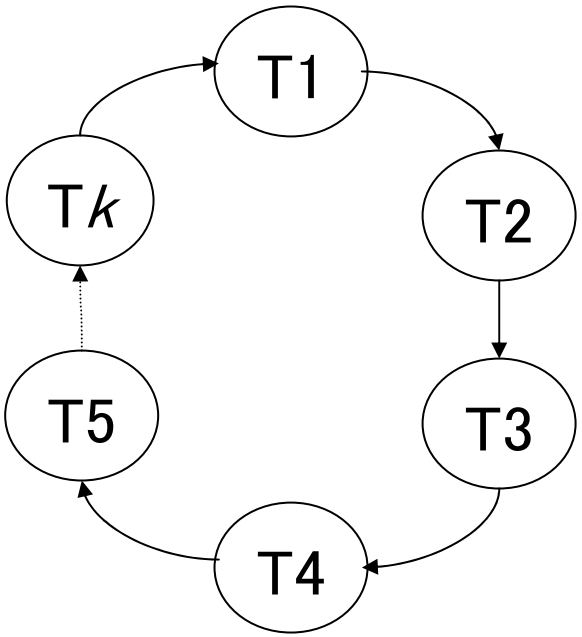
各トランザクションにおいて、すべてのLOCK文は、
すべてのUNLOCK文の前に実行する

定理
２相ロックプロトコルに従ってできた
スケジュール S は、必ず整列化可能

2相ロックの正当性の証明

背理法

整列化不可能な場合、整列化可能性判定グラフには閉路あり



T1 の UNLOCK 後に T2 の LOCK

T2 の UNLOCK 後に T3 の LOCK

:

T*k* の UNLOCK 後に T1 の LOCK

T1はUNLOCK後にLOCKしており
2相ロックプロトコルに従うことに矛盾

2相ロックプロトコルの最適性
条件は少しでも緩めると整列化可能性は失われる

T1 は2相ロックプロトコルに従わないとする

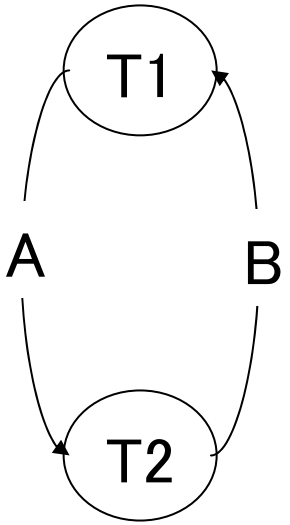
T1
LOCK A
:
UNLOCK A
:
LOCK B
:
UNLOCK B
:

T2
LOCK A
LOCK B
UNLOCK A
UNLOCK B

T1: LOCK A
:
T1: UNLOCK A

T2: LOCK A
T2: LOCK B
T2: UNLOCK A
T2: UNLOCK B

T1: LOCK B
:
T1: UNLOCK B



注意　2相ロックプロトコルで Deadlock を回避できるわけではない

```
T1:  AからBへ
2000円送金

LOCK A;
LOCK B;

READ A;
IF A >= 2000
     A := A − 2000;
WRITE A;
READ B;
B := B + 2000;
WRITE B;

UNLOCK A;
UNLOCK B;
```

```
T2:  BからAへ
5000円送金

LOCK B;
LOCK A;

READ A;
A := A + 5000;
WRITE B;
READ B;
IF B >= 5000
     B := B − 5000;
WRITE B;

UNLOCK B;
UNLOCK A;
```

T1: LOCK A;　許可
T2: LOCK B;　許可

T1: LOCK B;　不許可
T2: LOCK A;　不許可

T1とT2は永遠に
待ちつづける

まとめ

- 整列化可能性の概念を定義
- 整列化可能であることと
  整列化可能性判定グラフが閉路を含まないことは同値
- ２相ロックプロトコルは整列化可能の十分条件