同時実行制御

同時実行制御の役割

- 非直列スケジュールが直列化可能性を満足するように
  トランザクションの実行を制御する事
- いくつかの方式がある
  - スケジュール方式
  - 2相ロック方式　(教科書で扱われている方式)
  - 時刻印方式　(参考図書で紹介されている方式)
  - 楽観的方式
  - . . .

2つのトランザクションを仮定する

T_1: A さんの口座から B さんの口座に 10,000 円振り込む　$r_1(A)\ w_1(A)\ r_1(B)\ w_1(B)$

T_2: C さんの口座から B さんの口座に 5,000 円振り込む　$r_2(C)\ w_2(C)\ r_2(B)\ w_2(B)$

T_1 と T_2 の操作が並行して実行される系列を考える

- 系列1:　$r_1(A)\ r_2(C)\ w_1(A)\ r_1(B)\ w_2(C)\ w_1(B)\ r_2(B)\ w_2(B)$
  - B さんの口座は +15,000円となり，正しい結果になった
- 系列2:　$r_1(A)\ r_2(C)\ w_1(A)\ r_1(B)\ w_2(C)\ r_2(B)\ w_2(B)\ w_1(B)$
  - B さんの口座は ＋10,000円となり，正しくない結果になった
  - **T_2の操作が反映されていない !!!**
- 系列3:　$r_1(A)\ r_2(C)\ w_1(A)\ r_1(B)\ w_2(C)\ r_2(B)\ w_1(B)\ w_2(B)$
  - B さんの口座には +5,000円となり，正しくない結果になった
  - **T_1の操作が反映されていない !!!**

前スライドの不具合が出ないようにするには

- 前のスライドの例で正しくない結果が発生した原因
  - 系列2と系列3の結果は異なるが，いずれもトランザクションT_1でBさんの口座の更新途中でトランザクションT_2のBさんの口座に対する読み出し操作が入ったこと

↓

- あるトランザクションが操作しているデータに対する他のトランザクションの操作を禁止することにより，このようなことが起きないようにできるのではないか
  - データを操作する前に他のトランザクションがそのデータを操作できないように設定し，操作終了後に他のトランザクションがそのデータを操作できるように設定する

ロック機構

- ロックをするための操作 (*lock*)
  - 表記: *l*(A)　データ A のロックを要求する
  - 動作: データAがロックされていないときは，データA
    をロックする
    ロックされていなければ，アンロックされるまで
    待ち，ロックする
- アンロックするための操作 (*unlock*)
  - 表記: *u*(A)　データ A をアンロックする
  - 動作: データAのロックを解除し，ロック待ちの
    トランザクションがあればロックさせる

ロッキングプロトコル

- データの操作 (読み書き) 前にロック操作，操作後にアンロック操作を行う
- しかし，これだけでは相反直列化可能性を保証できない

  例えば，不整合検索異状を防げない

2相ロック法

- 通常のロッキングプロトコルに以下のプロトコルを加えた2相ロッキングプロトコルを用いる
  - ロックフェーズ:
    - ✓ 読み書きするデータをロックする
    - ✓ このフェーズが終わるまでアンロックしない
  - アンロックフェーズ:
    - ✓ ロックしているデータをアンロックする
    - ✓ このフェーズに入ったらロックしない

- トランザクション群 $\boldsymbol{T} = \{T_1, T_2, \ldots, T_n\}$ 内の全てのトランザクションが2相ロッキングプロトコルに従うなら，$\boldsymbol{T}$の非直列スケジュールが直列化可能であることを保証できる
- トランザクションの終了時にまとめてアンロックする方法を厳格な2相ロッキングプロトコルと言う

2相ロッキングプロトコルに従って 前の例にロック操作とアンロック操作を挿入してみる

- T_1および T_2にロック／アンロック操作を挿入した結果
  (この例ではデータの操作開始直前にロック，終了直後にアンロックを挿入)

  T_1: l_1(A) r_1(A) w_1(A) l_1(B) u_1(A) r_1(B) w_1(B) u_1(B)

  T_2: l_2(C) r_2(C) w_2(C) l_2(B) u_2(C) r_2(B) w_2(B) u_2(B)

- 正しくない結果を生じた系列2に，ロックおよびアンロック操作の挿入を試みる

  r_1(A)　r_2(C) w_1(A)　r_1(B) w_2(C)　r_2(B) w_2(B)　w_1(B)

  ↓

l_1(A) r_1(A) l_2(C) r_2(C) w_1(A) l_1(B) r_1(B) w_2(C) l_2(B) r_2(B) w_2(B) u_2(B) w_1(B) u_1(B)

Bはロックされているので，**l_2(B)** を実行できない

→ このスケジュールは発生しない

2相ロッキングプロトコルで生じるスケジュールが相反直列化可能性を満足することの証明

1. ***T*のログ *L* が相反直列化可能性を満たさないと仮定する**
2. 1.により，***L* の相反グラフに巡回閉路が含まれる**ことになる
3. 閉路が

   $T_1 \rightarrow T_2$, $T_2 \rightarrow T_3$, …, $T_{n\text{-}1} \rightarrow T_n$, $T_n \rightarrow T_1$

   で構成されていると仮定するなら，

   T_1 がアンロックフェーズに入った後 T_2 のロックフェーズが終る
   T_2 がアンロックフェーズに入った後 T_3 のロックフェーズが終る
   …
   $T_{n\text{-}1}$ がアンロックフェーズに入った後 T_n のロックフェーズが終る
   T_n がアンロックフェーズに入った後 T_1 のロックフェーズが終る

   となっていなければならない
4. 3.は，T_1 がアンロックフェーズに入った後に，T_1 のロックフェーズが終ることを意味しており，2相ロッキングプロトコルではこのような状態にはならない

   → 仮定 (相反直列化可能性を満たさない) が間違っていた

よって，2相ロッキングプロトコルに従ったトランザクション集合のスケジュールは

相反直列化可能性を満足する

デッドロック

- 複数のトランザクションが，他のトランザクションのロック解除を，互いに待っている状態
  - 以下の2つのトランザクションで，それぞれ最初のロック操作が実行されると，T_1は T_2 が B を解放するのも待ち，T_2 は T_1 が A を解放するのを待つ (永久に)

    T_1: l_1(A) l_1(B) ...

    T_2: l_2(B) l_2(A) ...
- デッドロックが生じた場合は，その原因になっているトランザクションのいくつか (生贄，victim) をロールバック (rollback) することで解除できる

  ロールバック：トランザクションが実行されなかった状態に戻す

デッドロックの発生条件

(プロセスはトランザクション，資源はデータと読み替えてください)

デッドロックが生じる条件として，以下の4つがあり，どれか1つでも満足しなければデッドロックは生じない

- ✓ 相互排除条件　プロセスは，必要とする資源を占有する
- ✓ 待ち条件　プロセスは，自分に割り付けられた資源を保持しながら，さらに必要とする資源を待つ
- ✓ 横取り不可能条件　資源を解放するまで，それらを保持しているプロセスから取り上げられない
- ✓ 循環待ち条件　資源待ちの循環が存在する

デッドロックの検出

- 厳密に調べる方法 (相反直列化可能性の判定と同程度の手間)
  - ➢ 待ちグラフを用いる
    - ✓ トランザクションを節点とする
    - ✓ トランザクション T_2 がロックしているデータのアンロックを T_1 が待っているときに，T_1 から T_2 に有向辺を $(T_1 \rightarrow T_2)$ を設ける
  - ➢ 待ちグラフに巡回閉路が含まれていればデッドロックが生じている
- タイムアウトを用いる方法 (正確ではない)
  - ➢ 指定した時間内にロックできない場合にデッドロックと判定する
    - ✓ 利点：デッドロック**検出の手間がかからない**
    - ✓ 欠点：**誤判定**（実際にはデッドロックではないが，デッドロックと判定）することがある

時刻印方式

- トランザクションに予め順番（時刻印）をつけ，その順序で逐次実行した結果と等価になるように制御
  - 順序を乱す操作を行ったトランザクションをロールバック
- 3 種類の時刻印を使う
  - トランザクションの時刻印 (ts(T))
    - ✓トランザクションの投入時刻などを用いる
  - 書込み時刻印 (wts(A))
    - ✓各データに持たせるもので，そのデータに書込んだトランザクションの時刻印の最大のもの
  - 読出し時刻印 (rts(A))
    - ✓各データに持たせるもので，そのデータを読出したトランザクションの時刻印の最大のもの

データ読出しの条件

- 読出しトランザクションの時刻印が，データの書込み時刻印より大きいか等しい

  $ts(T) \geqq wts(A)$

  - 自分より前に実行すべきトランザクションが書込みを行っていたデータからは，読み出せる

    逆を言えば，自分より後に実行すべきトランザクションが書込みを行ったデータからは，読み出せない

- 条件を満足するとき： データを読み出し，
  $rts(A) < ts(T)$ のときは $rts(A)$ を $ts(T)$ にする

  条件を満足しないとき： ロールバック

データ書込みの条件

- 書込みトランザクションの時刻印が，データの読出しとおよび書込み時刻印より大きいか等しい

  ts(T)≧rts(A)　かつ　ts(T)≧wts(A)

  - 自分より前に実行すべきトランザクションが読出しまたは書込みを行っていたデータには書き込める

    逆を言えば，自分より後に実行すべきトランザクションが読出しや書込みを行ったデータには書き込めない

- 条件を満足するとき：　データを書き込み，
  wts(A) を ts(T) にする

  条件を満足しないとき：　ロールバック

Thomas のルール

- 以下の理由で，書込み条件の2番目の条件 (ts(T)≧wts(A)) を外すことができる
  - ts(T)≧rts(A) かつ ts(T)＜wts(A) の場合，自分より後で実行すべきトランザクションが読み出しを行っていないので，T とAに最後に書き込みを行ったトランザクションの間に読出し操作が挟まれておらず，連続した書込みが行われたと解釈できる
- その解釈から，1番目の条件のみ満たされる場合は，T は何もせず，書込み操作が成功したことにすればよい

時刻印方式による制御例

- データの時刻印の初期値
  - rts(A)=wts(A)=0
  - rts(B)=wts(B)=0
  - rts(C)=wts(C)=0
- トランザクションの時刻印
  - ts(T_1)=1, ts(T_2)=2

成功例

$r_1(A)$ $r_2(C)$ $w_1(A)$ $r_1(B)$ $w_2(C)$ $w_1(B)$ $r_2(B)$ $w_2(B)$

操作	rts(A)	wts(A)	rts(B)	wts(B)	rts(C)	wts(C)
$r_1(A)$	1	0	0	0	0	0
$r_2(C)$	1	0	0	0	2	0
$w_1(A)$	1	1	0	0	2	0
$r_1(B)$	1	1	1	0	2	0
$w_2(C)$	1	1	1	0	2	2
$w_1(B)$	1	1	1	1	2	2
$r_2(B)$	1	1	2	1	2	2
$w_2(B)$	1	1	2	2	2	2

失敗例

r_1(A) r_2(C) w_1(A) r_1(B) w_2(C) r_2(B) w_2(B) w_1(B)

操作	rts(A)	wts(A)	rts(B)	wts(B)	rts(C)	wts(C)
r_1(A)	1	0	0	0	0	0
r_2(C)	1	0	0	0	2	0
w_1(A)	1	1	0	0	2	0
r_1(B)	1	1	1	0	2	0
w_2(C)	1	1	1	0	2	2
r_2(B)	1	1	2	0	2	2
w_2(B)	1	1	2	2	2	2
w_1(B)	ts(T_1) < rts(B) により拒否される					

時刻印方式の問題点

- ロールバックされたトランザクションが書き込んだデータを読み出したトランザクションもロールバックされる (ロールバックの連鎖)
- 書込みの割合が多くなるほど，ロールバックの可能性が上がる
  - ➢ 時刻印方式は書込みが多い場合には適さない

更新前の値を記録しておくことで読出し操作が原因となるロールバックをなくす多バージョン時刻印方式もある

- ➢ ただし，古い値を記録するための領域や管理が必要

2つの方式の比較

- 2相ロッキング方式
  - ロックの管理に手間がかかる
    - ✓ 誰がロック待ちか管理する必要がある
  - デッドロックの検出のオーバヘッドがある
    - ✓ タイムアウトを用いれば検出のオーバヘッドはなくなるが誤検知の問題がある
- 時刻印方式
  - 処理のオーバヘッドが小さい
    - ✓ 時刻印を追加するだけで，それぞれのデータで独立して処理できる
  - 書き込みの頻度が増えると，ロールバックが増える

一般に，書き込み頻度が少ない場合は時刻印方式が有利で，多くなると2相ロッキング方式が有利