コンピュータシステムII（10）

情報基盤センター
天野　浩文

1

前回のおさらい（1）

- 排他制御（相互排除）
  - 複数のプロセス（スレッド）のプログラムコードの中に，共通の資源（例：共有変数など）にアクセスする部分があるとき，その部分を**同時に実行できるプロセス（スレッド）がただ一つとなるようにする**こと．
    - 最初のプロセス（スレッド）が上記のような**危険区間**（critical section）に到達すると，他のプロセス（スレッド）は危険区間の中に入れないように制御する．
- 同期処理
  - プロセス（スレッド）が，**何らかの条件が成り立つまで処理を中断して待つ**こと．
    - ここでいう「条件」とは，自分自身以外のものによって変更される種類のものを指す．

2

前回のおさらい（2）

- セマフォ（semaphore）
  - もともとは相互排除のための機構
  - 少し工夫すると，ある種の同期にも使える．
  - 最も簡単な相互排除機構
- 以下のような性質を持つ整数型変数
  - 初期化を別にすれば，次のような P および V と呼ばれる2種類の操作によってのみアクセス可能
    - P: 危険区間の開始を宣言する．**他のプロセス（スレッド）が危険区間に入っているときには，先に進まずにここで待つ**．
    - V: 危険区間の終了を宣言する．**同じセマフォに対するP命令で待っている他のプロセス（スレッド）が先に進めるようになる**．

3

前回のおさらい（3）

- セマフォを排他制御に使うとき
  - 初期状態は，危険区間に入れる状態
    - セマフォの初期値を1または正の整数にセット
  - 各プロセス（スレッド）は，危険区間の前後をP操作・V操作ではさんでおく．
- セマフォを同期処理に使うとき
  - 初期状態は，P操作が先に進めない状態
    - セマフォの初期値は 0 または負の整数
  - 先行させたいほうにV操作を，後追いさせたいほうにP操作を実行させる．

4

前回のおさらい（4）

- バリア同期
  - 待ち状態のプロセス（スレッド）を1つだけ覚醒させるPV操作とは異なり，*N* 番目にそれを実行したものが他のすべてを覚醒させる．
  - バリア同期すべきプロセス（スレッド）の数 *N* を初期値とする共有変数を用いる

```
Barrier(S): Sを1減らす
            if S>0 then {
               自分自身を休眠状態にする
            } else {
               他のすべてのプロセス（スレッド）を覚醒させる
               Sを再初期化
            }
```

5

前回のおさらい（5）

- P操作の実装に関する注意
  - 「先に進んでよいかどうか」の条件判断と，それに続くセマフォの変更や休眠処理までの間に，他のP操作の条件判断がはさまると，不具合が発生する．
  - 複数のプロセッサが共有メモリ上の変数にアクセスできる場合には，割り込みを禁止するだけでは不可
    - 単一の機械語命令で実装するとよい
- `Test-and-Set` 命令（機械語命令）を使ったP操作

```
while Test-and-Set(lock) do {}
```

  - `Test-and-Set(lock)`
    - 変数`lock`の値を返す
    - `lock`には *true*を書き込む

`lock`の値が *true* なら，**TS命令の繰り返し**．
`lock`の値が *false* なら，何も実行せずに終了して**次に進む**．

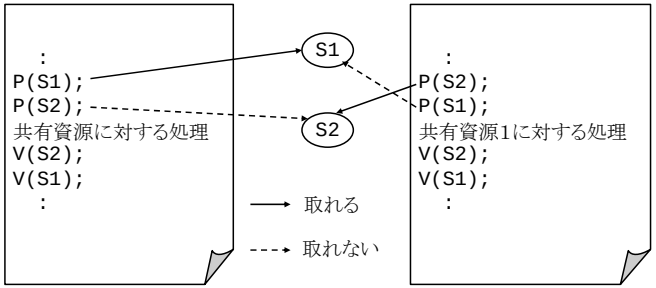
6

排他制御機構と同期機構（つづき）

7

デッドロック（deadlock）

- 「すでに確保した資源がありながら，他のプロセス（スレッド）が先に確保してしまったために待たされている資源がある」という状態にある一群のプロセス（スレッド）が，いずれも先に進めなくなって膠着状態に陥ること．



8

デッドロックが起きるための必要条件

- デッドロックが起こるためには，**以下の条件がすべて成立していなければならない**．
  1. **相互排除**（mutual exclusion）
     少なくとも1つの資源は，同時にひとつのプロセス（スレッド）にしか利用できない．
  2. **確保と待機**（hold and wait）
     少なくとも1つの資源を確保しているが，他のプロセス（スレッド）が先に確保してしまった資源が利用可能になるのを待っているプロセス（スレッド）が少なくとも1つある．
  3. **非プリエンプション**（no preemption）
     いったん確保された資源は，そのプロセス（スレッド）の処理が終了して解放されるまでは利用可能にならない．
  4. **巡回待機**（circular wait）
     「プロセス（スレッド）i はプロセス（スレッド）j の確保している資源が解放されるのを待っている」という関係が，$p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n \rightarrow p_1$という形式で閉路を形成している．

9

デッドロックの抑止（prevention）

- デッドロックの必要条件4つのうちの**いずれかが成立しないように**，プロセス（スレッド）からの資源要求の順序・タイミングに一定の制約を課す．
  - 条件1：「相互排除」を取り除くのは，一般には不可能．
  - 条件2：「確保と待機」を取り除くには，各プロセス（スレッド）に，たとえば
    - 実行に先立ってすべての資源を確保するようにさせる．
    - 共有資源を何も確保していないときにのみ，資源を要求するようにさせる．
  - 条件3：「非プリエンプション」を取り除くには，たとえば：
    - すでに何かの資源を確保したプロセス（スレッド）がさらに別の資源を要求したとき，それが確保できなければ，先に確保した資源を取り上げてしまう．
  - 条件4：「巡回待機」を取り除くには，たとえば
    - 資源の間にあらかじめ順番を付け，その順にしか資源を要求できないようにさせる．

10

デッドロックの回避（avoidance）

- デッドロックの抑止の問題点
  - （確かに，デッドロックが起きていないか，起きるおそれがないかを監視しておく必要はないのだが…）
  - しかし，資源の利用効率，ひいてはシステム全体の処理効率が低下してしまう．
- **デッドロックの回避**（avoidance）
  - システム全体の資源の割り当て状況を監視しながら，デッドロックが発生する可能性のある資源割り当て要求が出されたら，それの処理を遅らせる．
  - 資源の割り当て状況やプロセス相互の待機状況を常に監視しておかなければならない．

11

デッドロックが起こってしまったら

- 不幸にもデッドロックが起こってしまったら
  - **そのままにしておいても自然に解消されることは絶対にない**．
- **デッドロックからの回復**（recovery from a deadlock）
  - デッドロックに関与しているプロセス（スレッド）のいずれかを強制終了させる．
    - 終了させられたプロセス（スレッド）が確保していた資源は解放され，巡回待機の条件が（一時的に）消滅する．
    - あるいは，デッドロックが解消されるまで，順にプロセス（スレッド）を強制終了して，資源を解放させていく．
  - デッドロックに関与しているプロセス（スレッド）のいずれかを，まだ実行開始していなかったことにする．
    - やりかけた操作をすべて元の状態に戻して，最初から実行しなおす．

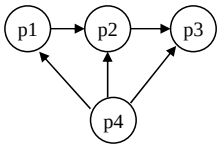
12

デッドロックの検出（detection）

- デッドロックの回避やデッドロックからの回復を行うためには，システムがデッドロックに陥っていっているかどうか，あるいは，デッドロックに陥る危険がないかどうか，を知る方法が必要．
- **待ちグラフ**（wait-forグラフ）
  - 「プロセス（スレッド）i はプロセス（スレッド）j の確保している資源が解放されるのを待っている」という関係を，$p_i \rightarrow p_j$という枝（edge）で表したグラフ
  - これに閉路があることは，デッドロック状態にあることの必要十分条件である．
  - 待ちグラフを常時管理しておき，定期的に閉路検出アルゴリズムを走らせる．
- **時間切れ**（timeout）方式
  - ある程度の時間が経過しても終了しない処理は，デッドロックに陥っていると推定する．
  - 検知は非常に簡単だが，単に時間がかかっているだけのものをデッドロックと誤判定する危険もある．

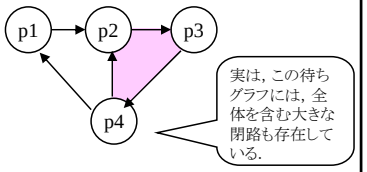
13

待ちグラフの例



デッドロックではない状態

- p3が終了すれば，p2も終了できる．
- p2が終了すれば，p1が終了できる．
- p1とp2とp3が終了すれば，p4が終了できる．



デッドロックが発生している状態

- p2とp3とp4は巡回待機の状態にあるため，このままでは終了できない．
- p1は，p2が終了しない限り終了できない．
- p4は，p1とp2とp3が終了しない限り終了できない．

14

単一プロセッサ上のデッドロック検出との比較

- 並列処理・分散処理におけるデッドロック検出
  - 実は，単一プロセッサ上で複数のプロセス（スレッド）を時分割実行している場合と，ほぼ同様の議論となっている．
    - 単一プロセッサ上でもデッドロックは発生しうる．
    - それを抑止・回避・検出することが必要．
- ただし，並列処理・分散処理においては，単一プロセッサの場合よりも，より複雑で困難な問題になる．
  - システム全体の状況を把握しているスケジューラがどこで動作しているか？
    - プロセススケジューリングの項で，システム全体のREADYプロセスキューやスケジューラの配置を議論したときと同様の状況

15

デッドロックからの回復の有効性（1）

- 並列処理・分散処理には，次の2つの理由で行われるものがある．
  1. 複数の演算ノードを同時に必要とするような大規模計算や，広域に分散したデータをまとめて使用するような大規模データ処理のため
  2. 個々の計算・処理は単一または少数の演算ノード上で実行可能であるが，たくさんの計算・処理を複数の演算ノード上にばらまいて同時に実行するため（結果として，全体が並列処理・分散処理になっている）
- **デッドロックからの回復は，上記のいずれの場合も有効なのだろうか？**

16

デッドロックからの回復の有効性（2）

- 複数の演算ノードを同時に使用するような大規模計算・処理のとき
  - デッドロックを起こしたプロセス（スレッド）も，そうでないプロセス（スレッド）も，共通の目的のための大きな仕事の一部分を分担している．
  - 内部でバリア同期を行っていることもある．
- この中でデッドロックが発生する場合
  - プログラム全体を再度先頭から実行しなおしても，また同じプロセス（スレッド）群が同じ場所でデッドロックに陥る可能性が高い．
  - デッドロックを発生させたプロセス（スレッド）だけ強制終了させたり再実行させたりすると，正しい結果にならない．
    - 前回示した行列積の例を思い出して欲しい．
- このような状況では，**デッドロックからの回復はあまり有効ではない**．
  - むしろ，デッドロックを回避または抑止できるように，プログラムのほうを修正すべき

17

デッドロックからの回復の有効性（3）

- 複数の演算ノード上にばらまかれて同時に実行される，多数の小さな計算・処理のとき
  - それらの計算・処理は，全体でひとつの計算・処理を行っているわけではない．全体でバリア同期を取るようなこともない．
  - 単体でデッドロックを起こす可能性は低いが，複数集まると，デッドロックに陥ることがある．
    - 後で述べる**トランザクション処理**は，このような処理の好例
- この種のデッドロックが発生した場合
  - デッドロックを発生させてしまったプロセス（スレッド）だけ強制終了させたり再実行させても，他のプロセス（スレッド）は正しい結果を生成できる．
  - 再実行されたプロセス（スレッド）は，再びデッドロックが起こらなければ，正しい結果を生成できる．
- このような状況では，**デッドロックからの回復も有効である**．
  - 実際に，データベースでは，デッドロックに陥ったトランザクション（この用語の意味は後述）のアボートや再実行を行うことがある．

18

並行処理制御（Concurrency Control）

教科書では，これを「同時性制御」と呼んでいる．しかし，concurrent は従来から「並行（の）」と訳されることが多かったし，「同時」という訳語は simultaneous との混同を招く恐れもある．この講義では，あえて「並行処理制御」という用語を用いる．
（なお，「並行制御」，「並行性制御」，「同時実行制御」などと呼ぶ専門家もいる．）

19

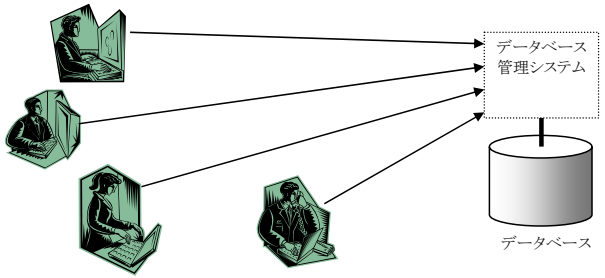
トランザクション（Transaction）

- 一つの意味的にまとまった仕事
  - 「意味的にまとまった」とは，途中まで実行されて中断・中止された状態では，意味のある（正しい）結果を残せない，ということ．
  - データベース（database, DB）の分野で使われることが多く，「ひとまとまりの処理を構成するデータベース操作の集まり」などと定義されることもある．
  - ここでは，データベースに議論を限ることにする．
- トランザクションの例
  - 預金口座Aに，指定された金額を入金する
    - データベース中の預金口座Aのデータに対する参照（読み出し）と更新（書き込み）
  - 預金口座Aから，指定された金額を引き出す
    - データベース中の預金口座Aのデータに対する参照と更新
  - 預金口座Aから預金口座Bに，指定された金額を送金する．
    - データベース中の預金口座Aのデータに対する参照と更新
    - データベース中の預金口座Bのデータに対する参照と更新

20

トランザクションとデータベース管理システム

- **データベース管理システム**（database management system, DBMS）
  - データベースへの操作（読み出し，書き込み）のすべてを代行する
- トランザクションは，すべてこのDBMSによって処理される．



21

古典的なトランザクション処理

- CPUに比べるとディスク装置ははるかに低速なので，あるトランザクションの処理のためにディスクアクセスが始まると，CPUが遊んでしまう．
  - CPUが一つしかない場合でも，その間に他のトランザクションのための演算処理を行えばよいではないか！
- 時間軸の上では，複数のトランザクションのための処理が同時に進行しているように見える．
  - このように，一つのCPUの上で複数の処理が（実行権を奪い合いながら／譲り合いながら）同時に進行している状態を**並行**（concurrent）**である**という．

22

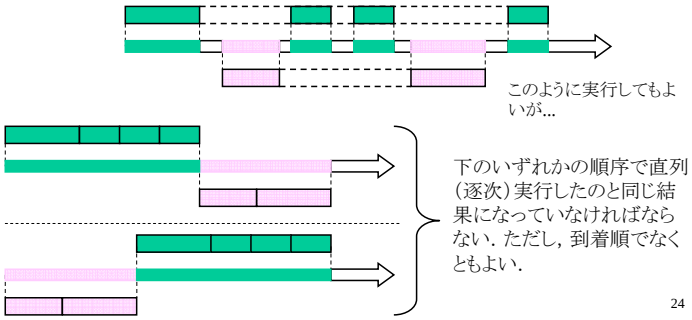
並行処理制御（Concurrency Control）の必要性

- 複数のトランザクションを並行に処理する場合に，何の制御もせずに野放図に並行処理すると:
  - トランザクションの要求した処理が正しく実行されないおそれがある．
  - データベースの一貫性が損なわれるおそれがある．
    - データベースのデータは長期間保存されることが多いので，誤りが長期間他の処理に影響を与え続けたり，誤りがたくさん蓄積してデータベースの価値が損なわれる．
- 並行処理を行う場合には何らかの制御が必要だ！
  ⇒**並行処理制御**
  - （性能向上のために）並行処理したとしても，あえてそうしなかった場合と同じ結果になっている必要がある．

23

並行処理制御の目的

- **直列可能性**（serializability）
  - 複数のトランザクションをあるスケジュールで切り替えながら実行した場合には，それと等価な結果を生む何らかの直列スケジュールが存在しなければならない．



24

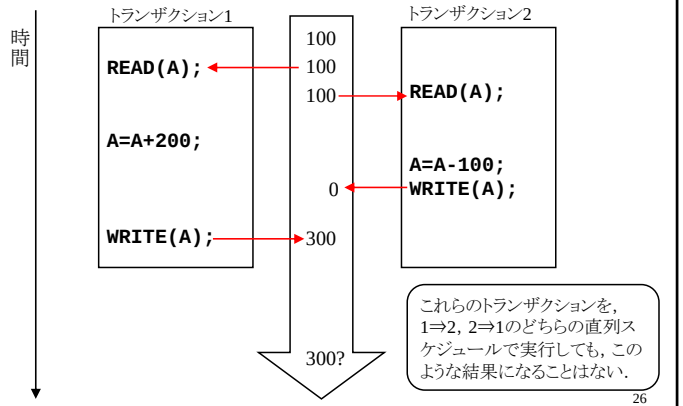
トランザクション処理に関する最近の議論

- 複数のトランザクションを同時に実行してもよいが，その処理内容について，DBMSは以下のような性質を保証しなければならない．
  - ACID特性（ACID property）
    - **原子性**（Atomicity）
      - トランザクションが行う操作はすべてデータベースに反映されるか，あるいは，すべて取り消されるか，のどちらかでなければならない．
    - **一貫性**（Consistency）
      - 一貫性の取れたデータベースに対してトランザクションを実行した結果は，再び一貫性が取れていなければならない．
    - **隔離性**（Isolation）…直列可能性と同じことを言っている
      - 複数のトランザクションを並行処理（同時実行）した場合でも，その結果は，それらのトランザクションを何らかの順序で逐次処理した場合と一致しなければならない．
    - **耐久性**（Durability）
      - 一旦実行終了したトランザクションの行ったデータ操作は，その後のシステム障害などで消失してはならない．

25

トランザクションが隔離性を満たさない例（1）

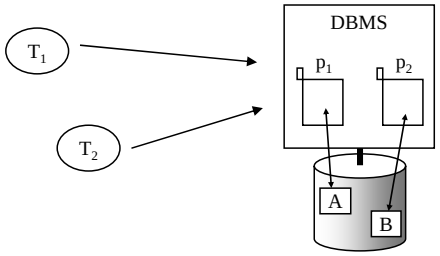
- 前回の講義でも取り上げた，口座からの出金の例



26

トランザクションが隔離性を満たさない例（2）

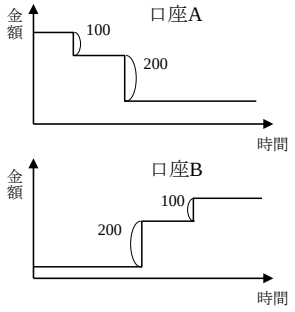
- 口座Aから口座Bに送金するトランザクションを考える．
  - 口座Aのデータと口座Bのデータをそれぞれプロセス p_1とプロセス p_2が管理している．
  - トランザクションT_1とトランザクションT_2は，口座Aからの出金処理・口座Bへの入金処理を行う．それぞれの金額は100，200とする．



27

トランザクションが隔離性を満たさない例（2）（つづき）

- T_1・T_2はいずれも，出金を行ってから入金を行うが，以下のようなスケジュール S で実行されたとする．
  1. p_1：T_1のために口座Aの出金処理
  2. p_2：T_2のために口座Aの出金処理
  3. p_1：T_2のために口座Bの入金処理
  4. p_2：T_1のために口座Bの入金処理
- T_1・T_2を直列（逐次）に実行する場合の順序
  - $T_1 \Rightarrow T_2$
  - $T_2 \Rightarrow T_1$
- スケジュール S で実行された場合の口座Aと口座Bに残る**履歴の組み合わせ**（右の2枚の図）は，上記のいずれの直列スケジュールとも一致しない．



28

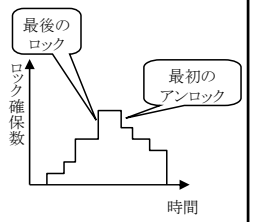
隔離性に関するより厳密な定義

- 複数のトランザクションを並行処理(同時実行)した場合でも，その結果は
  - データベースの，可能なすべての初期状態に対して，
  - アクセスされるすべてのデータ項目について，
  - それらのトランザクションを何らかの順序で逐次処理した場合の結果(**変化の履歴を含む**)と一致しなければならない．
- 特定の初期状態のときだけ等価な直列スケジュールが存在する，というのではダメ
  - 先ほどの例で，残高不足でどちらのトランザクションも送金できないような場合には，どんなスケジュールで実行しても直列実行の場合と(たまたま)等価な結果になる．
- データ項目Aの上で観測されるトランザクション実行順とデータ項目Bの上で観測されるトランザクション実行順が異なるようなことも，あってはならない．

29

隔離性を満たすための方策の例

- **二相ロック**(two-phase locking)プロトコル
  - 各トランザクションは，データベース内のデータ項目に対し，ロック・アンロックによる排他的なアクセスを行う．
    - データ項目に対するロック・アンロックは，セマフォに対するP・V操作のような働きをする．
    - 共有(読み出し)ロックと専有(書き込み)ロック
    - 複数の資源をロックする場合，いったんアンロック操作が始まったら，二度とロック操作を行わない．
  - 直列可能性(隔離性)を保証できることが理論的に証明されている．
  - しかし，デッドロックに陥る可能性がある．
    - やりかけた仕事を最後まで実行できない可能性がある，ということ．



上記はあくまでも一例．ロック・アンロックの形態が異なる方法や，ロック・アンロックを全く用いない方法などもあるが，それらの詳細はデータベースの講義に譲る．

30

トランザクション処理と並列・分散処理(1)

- トランザクション処理
  - DBMSに格納されたデータが大規模であれば，その処理負荷は非常に重い．
  - 非常にたくさんのトランザクションがあるときにも，全体の処理負荷が重くなる．
  - あるトランザクションが必要とするデータが広域に分散しているかもしれない．
- 複数のトランザクションを単一のプロセスで処理すると(データベース管理システムが単一プロセスで逐次動作すると)
  - 複数の演算ノードがあっても並列処理・分散処理が行えず，処理効率が上がらない．
  - 必要なデータが分散してる状態では処理ができない．

31

トランザクション処理と並列・分散処理(2)

- DBMS自体を，複数の演算ノード上で複数のプロセスまたはスレッドとして処理する．
  - トランザクションに含まれる複数の操作を，**並行ではなく並列に実行できるようになる**．
  - しかし，先に述べた隔離性の条件を満たすための制御は，より複雑になる．
    - 処理を実行する主体が複数あるため，直列に実行したときと同じ結果になるように保証するのは大変．

32

原子性を満たすために（1）

- 隔離性を満たすような並行処理制御が行われている場合でも，トランザクションの処理が最後まで完遂できないことがある．
  - デッドロックに陥った場合
  - （例えば残高不足などの理由で）それ以降の処理が許可されない場合
  - DBMSに障害があって，トランザクション処理全体が停止してしまうような場合
- トランザクションは，途中まで処理が終わった状態で放置されてはいけない．
  - 最後までやり遂げるか
  - 実行されていない状態（最初の状態）からやり直すか

33

原子性を満たすために（2）

- いくつものデータ項目にアクセスするトランザクションがデータベースへの書き込みをその都度行っていると...
  - 途中で止まったときに，すでに書き込まれた結果を元に戻すのは大変．
- そこで，通常は，以下のような方法が採られる．
  - トランザクション実行中の書き込みは，当該データ項目の作業用コピーに対して行う．元のデータベースにはただちに反映させない．
  - トランザクションが正常終了できるとわかったときに，初めて，すべての書き込み結果を，元のデータベースに反映させる
    ⇒**コミット**（commit）操作
  - トランザクションが正常に終了できないとわかったときは，当該トランザクションを強制終了し，作業用コピーを捨てる．
    （実行前の状態に戻ることができる）
    ⇒**アボート**（abort）操作

34

分散DBにおけるコミット操作の問題点

- DBMSが広域分散システム上で動作しているような場合
  - 単一の「管理プロセス」がシステム全体の状態を把握・管理するのは非常に困難
    - コミットしてもよいのか，アボートすべきなのかの判断を単一のプロセスで行うのが大変．
    - 分散システムの性質上，コミット操作（DBへの最終書き込み）そのものを管理プロセスが行うことはできない．
  - 各演算ノードでコミット操作が正常終了することを保証するのは非常に困難
    - 通信路に障害が起きて，通信が届かない可能性がある．
    - 演算ノードがシステム障害で動かなくなることすらあり得る．
    - こちらの問題を完全に解決することは不可能．

35

二相コミット（Two-Phase Commit）プロトコルの原理

- 仮定
  - 各演算ノードごとに，そこのデータを管理する「リソースマネージャ」が1つ存在する．
  - 個々のデータ項目の監視まではできないがリソースマネージャの「投票」の結果を管理できる「コーディネータ」が，システム全体で1つ存在する．
- **1つの分散トランザクションの作業用コピーの上での処理が終了した後のコミット操作を，次の2つの段階に分割する．**
  - 第一段階
    - 各演算ノードでコミットの準備ができたかどうかを確認する．
  - 第二段階
    - 各演算ノードのリソースマネージャからの「投票」を受けて，コミット／アボートの判断を行い，各ノードでコミットを行う．

「二相コミット」は，コミット操作そのものが2つの段階に分かれているためにそう呼ばれているのであり，トランザクションがコミット操作の前と後に分かれているためと考えるのは誤りである．

36

二相コミットプロトコルの詳細（1）

- 第一段階（Phase 1）
  - コーディネータは，各リソースマネージャに「**準備要求**（PREPARE）」メッセージを送信
  - 各リソースマネージャは，以下の処理を行う
    - コミットの準備ができたら，「**準備完了**（READY）」を投票する．コミットできないときには，「**拒否**（REFUSE）」を投票する．
    - （処理がまだ終了していない演算ノードは）投票はいくらでも遅らせることができる．
      - （もちろん，一定の制限時間をおいて，「時間切れ」と判定することはあり得る．）

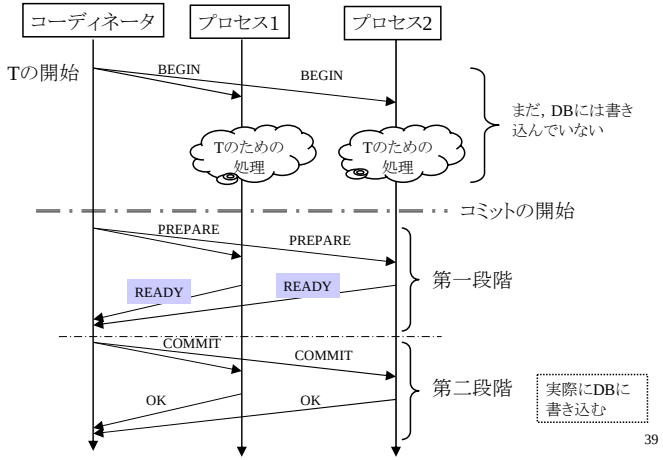
37

二相コミットプロトコルの詳細（2）

- 第二段階（Phase 2）
  - コーディネータは，すべてのリソースマネージャから「準備完了」のメッセージを受け取ったら，コミットすることを決定する．どれかひとつでも「拒否」が混ざっているか，あるいは，あまりに時間がかかり過ぎて「あきらめた」（＝時間切れになった）場合には，アボートすることを決定する．
  - コーディネータは，各リソースマネージャに，コミット／アボートの決定を送信する．
  - 各リソースマネージャは，実際のコミット操作に必要な書き込み処理またはアボートに必要な処理を行い，終了すれば「**完了**（OK）」を返答する．
  - コーディネータは，すべてのリソースマネージャから「完了」を受け取ると，初めて，分散トランザクションが正常に終了したか，「なかったことにできた」ことを確信できる．

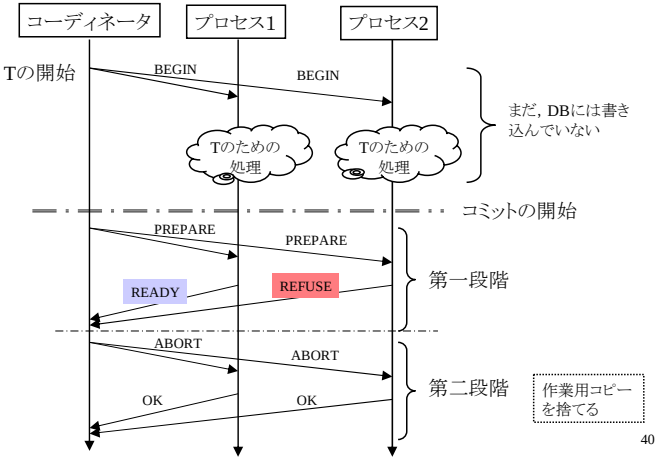
38

ある分散トランザクションTの『一生』（1）



39

ある分散トランザクションTの『一生』（2）



40

まとめ

- デッドロック
  - デッドロックの必要条件
  - デッドロックの抑止・回避，デッドロックからの回復
  - デッドロックの検出
- トランザクション処理
  - データベースに対する操作を，意味のあるまとまり（トランザクション）として処理すること．
  - 並行に処理しても，並列に処理してもよい．
  - ただし，ACID特性を保証しなければならない．
- 並行処理制御
  - ACID特性のうちのI（隔離性）を保証するための制御
  - 一例として，二相ロックがある．
- コミット
  - ACID特性のうちのA（原子性）を保証するための制御
  - 並列・分散でなくても，よく用いられる．
- 二相コミット
  - 分散システムにおいてトランザクションの原子性を保証するためのプロトコル
    - 二相ロックと混同しないように注意すること！

41