解　説

2. 基　礎　技　術

2.2 分散処理システムにおける同時実行制御†

前川　守††　清水謙多郎††
濱野　純††

0. はじめに

分散処理システムはその本来的性質として，多数のプロセスが独立並行して処理を行い，その並列性によって高い処理能力を活用することができるものである．しかし，互いに関連のあるプロセス同士が並行して実行される場合には，それらの間になんらかの同時実行制御機構がなければ，矛盾した結果を生じてしまう．これには，時分割処理の時代からあった問題と，分散処理システムであるがゆえに生じた問題が含まれる．

同時実行の制御にはシステム内で生起する事象の順序関係が定義されなければならないという観点から時刻の管理が，共有される情報への同時アクセスを防ぐ手段として排他制御方式が必要であり，これらの上に並行性制御が行われる．本稿ではこうした問題と，デッドロックの検出と解除方式とについて解説を試みる．

1. 時刻の管理

システム内で生起した事象の生起順序を決定し，管理することは，同時実行制御を行う上での基本的な問題である．

この管理を単一のコントロール・ノードが集中的に行うのであれば，分散システム特有の問題の多くを回避することができる．しかし，そのようにした場合には，コントロール・ノードの能力がシステム全体のボトル・ネックとなり，またコントロール・ノードの故障がシステム全体の機能停止につながるために不都合である．そのため，分散処理システムにおいては，同時実行制御管理のための機構もまた分散されて実現される必要がある．

分散処理システムはネットワーク上に構築されるシステムであって，プロセスに対する要求は一般にメッセージによって伝達される．メッセージが発信されてから受信されるまでにかかる時間はまちまちであって，そのために，実時刻では後に発せられた要求のほうが先に発せられた要求よりも先に（追いこして）到着することも起こり得る．

たとえば，図-1 の例では，アプリケーション AP_1 と AP_2 とが，それぞれサービス・プロセスPに要求を出しているが，論理上はまず AP_1 の要求があり，AP_1 から AP_2 への通信があり，その後で AP_2 の要求が生じているのであるから，Pにおいてはこれらの要求はメッセージの到着順（すなわち 5, 6 の順）ではなく，要求メッセージの発生順（1, 4 の順）に処理されるべきである．

このためには，AP_1 と AP_2 とが共通な時計を持って，それぞれのメッセージ中にその発信時刻を記入できるようにしておく必要がある．

2. 共通の時計

システム内での事象の生起順序を決定する規則には，一つのプロセス内で，プロセスが実行するアルゴリズムによって決まる順序と，プロセス間のメッセージの発信・着信の因果関係によって決まる順序との2種類がある．

1. 一つのプロセス内で起こる事象 a と b とがあるとき，a が b よりも先に実行されれば $a \rightarrow b$ である．

2. a があるプロセスから別のプロセスへのメッ

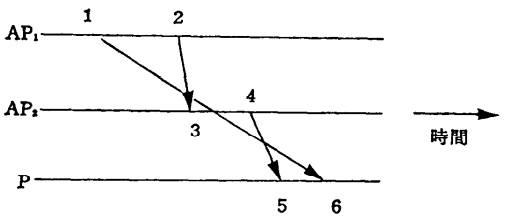

図-1　事象の順序の逆転

† Clocks, Mutual Exclusions, Concurrency Control and Deadlocks in Distributed Systems by Mamoru MAEKAWA, Kentaro SHIMIZU and Jun HAMANO (Dept. Information Science, Faculty of Science, Univ. of Tokyo).

†† 東京大学理学部情報科学科

セージ発信，b がその着信であるとき，$a \to b$ である．

この二つの規則に，推移律（$a \to b$, $b \to c$ ならば $a \to c$）を加えることで，システム内での事象に（半）順序関係を導入することができる．

この順序関係をみたす（$a \to b$ ならば clock $(a) <$ clock (b) である）時計をつくるには，下のようにすればよい．

1. 各プロセス（実際には計算機ごとに一つでよい）はカウンタをもち，新しい事象が起こるたびにそのカウンタを増加させる．

2. メッセージには，その発信時のカウンタの値がタイムスタンプとして記録されており，受信したプロセスはそのタイムスタンプが自分のもつカウンタよりも大きければ，自分のカウンタをタイムスタンプよりも大きくなるように変更する．

システム内のすべての事象は，これによって全順序をつけることができる．同じタイムスタンプをもつ事象については，プロセス（計算機）番号を使うことでどちらが大きいかを決めることにすればよい．

このように構成された時計は，システム内部の事象だけに基づく論理上のものである．そのため，システム外での時間関係との間に矛盾が生ずることもある．たとえば2人の人A，Bが別々の計算機で仕事をしているとき，Aが要求を投入し，そのことを電話でBに伝え，その後にBが要求を投入したとしても，システム内ではBがした要求のほうに小さいタイムスタンプがつけられることが起こりうる．これはシステム外で行われた通信に基づく順序関係がシステムの考慮に入っていないためである．

用いられるタイムスタンプを実時刻に近いものにすれば，この問題を解決することができる．そのためには，システム内の各計算機は十分に正確な時計をカウンタとして持ち，メッセージが到着したときには，その発信時刻と伝送遅れ時間（の最小）の見積りを用いて自分の時計を合わせなおすようにすればよい．Lamport[2] の結果によると，時計の誤差 k，ネットワークの直径 d（d 個の辺を通らないと通信できないノードの組が存在する），メッセージ発信の最大間隔 τ，予測不能なメッセージ遅れ時間 ξ に対し，$\varepsilon \approx d \times (2k\tau + \xi)$ 以下の誤差でシステム内のすべての時計を合わせることができる．

3. 排他制御問題

分散処理システムでの排他制御を，集中管理方式を用いて行うことはもちろん可能である．この場合，1回の排他制御を行うためには，要求するプロセスから集中管理プロセスへの要求メッセージと，それに対する回答メッセージとの二つのメッセージだけで十分である．しかし，前に述べたのと同じ理由から，排他制御管理も分散化できることが望ましい．分散化された制御方式では，

1. 各ノードは同量の管理情報をもつ．
2. 各ノードは同じアルゴリズムで決定を行う．
3. 各ノードは等しく負荷を分担する．
4. 各ノードから制御を起動するコストが同じ．
5. 単一ノードの故障が全システムの停止につながらない．

という性質が必要である．

はじめにこの問題に対して Lamport が示した解決はさきに述べたタイムスタンプを用いるものである[2]．この方式は，ネットワーク中でメッセージの到着する順が発信した順になることを仮定している．

1. クリティカル・セクション (critical section) に入ろうとするプロセスはその要求をキューに入れ，ほかのすべてのプロセスに対して Request メッセージを送る．

2. Request メッセージを受けたプロセスはその要求をキューに入れ，Acknowledge メッセージを返す．

3. クリティカル・セクションを出るプロセスはその要求をキューからはずし，ほかのすべてのプロセスに対し Release メッセージを送る．

4. Release メッセージを受けたプロセスは，その要求をキューからはずす．

Request メッセージを出したプロセスは，キューの中で自分の要求が最も古いものであり，かつその要求よりも新しいメッセージをほかのすべてのプロセスから受けとっていればクリティカル・セクションに入ることができる．この方式では，1回の排他制御につき（全プロセス数を N として）$3 \times (N-1)$ 個のメッセージが必要である．

Ricart と Agrawala の方法3では，$2 \times (N-1)$ のメッセージで排他制御が行われる．

1. クリティカル・セクションに入ろうとするプロセスは，他のすべてのプロセスに対して要求メッセージを送る．

2. 要求メッセージを受けたプロセスは，自分が要求を出していないか，自分の要求よりもこのメッセージのほうが古いとき，回答メッセージを返す．そうで

なければこの要求メッセージに対する回答メッセージの送信を延期する.

3. 要求メッセージを出したプロセスは,それに対する回答メッセージがほかのすべてのプロセスから得られたとき,クリティカル・セクションに入ることができる.

4. クリティカル・セクションからプロセスが出るとき,2.で延期してある回答メッセージがあれば,それを発信する.

以上あげた二つの方法では,プロセスはクリティカル・セクションに入るためにほかのすべてのプロセスから「満場一致」の許可を得なければならない.

Thomas[12] の方法 (Majority Consensus) では,二つの競合する更新要求があればどちらかは拒否されるという立場からこの「満場一致」の条件をゆるめ,プロセスの過半数から許可を得ればよいようにしている.これによって,排他制御1回あたりにかかわるプロセス数やメッセージ数を半減することができる.

さらに,Gifford[11] の weighted voting 方式では,プロセスごとに持ち票 (重み) をつけることによって,プロセスの過半数ではなくプロセスのもつ総票数の過半数を得ればよいようにしている.この方式である特定のプロセスに票のすべてを持たせ,他のプロセスの投票権を0としたものが集中管理方式であり,各プロセスに均等に票を与えたものが Thomas の方式である.その意味で,Gifford の方式は集中管理方式と分散管理方式の間のさまざまなアスペクトを,票の分散ぐあいによって記述することが可能である.

前川[4] の方式は分散型アルゴリズムの中でさらに少数のメッセージしか必要としない.

いま,プロセス i, j $(1\leq i,j\leq N)$ がクリティカル・セクションに入るために要求メッセージを送り,それに対する回答メッセージを返される相手プロセスの集合をそれぞれ Si, Sj とかくことにすると,

1. $\forall i,j$ について $Si\cap Sj\neq\phi$
2. $|S_1|=|S_2|=\cdots=|S_N|=K$ (ある定数)
3. $\forall i \quad |\{j|Sj\ni i\}|=D$ (ある定数)

であることが必要である.さらに,

4. $\forall i \quad i\in Si$

とすれば,実際に送られるメッセージ数を少なくできる.これらの条件から少し計算をすると,ほぼ $K=\sqrt{N}$ が得られる.すなわち,分散型アルゴリズムでは,最小限約 $\sqrt{N}$ 個の他のプロセスと通信しなければ排他制御は実現できず,また,上の 1.〜4. をみたすように Si をとれば $O(\sqrt{N})$ のメッセージで実現できることになる.このような条件をみたす Si の集合を求めることは,N 点の finite projective plane を見つけることと等価な問題である.$(K-1)$ が素数幅のとき,こうした Si の集合をとることができ[20],それ以外の場合にも,条件2,3を少しゆるめれば可能である.詳細は文献[4]を参照されたい.図-2 に,比較的小さな K について,Si の集合を構成した例を示す.

$S_1=\{1, 2, 3\}$
$S_4=\{1, 4, 5\}$
$S_6=\{1, 6, 7\}$
$S_2=\{2, 4, 6\}$
$S_5=\{2, 5, 7\}$
$S_7=\{3, 4, 7\}$
$S_3=\{3, 5, 6\}$

(a) $K=3$

$S_1=\{1, 2, 3, 4\}$
$S_3=\{1, 5, 6, 7\}$
$S_8=\{1, 8, 9, 10\}$
$S_{11}=\{1, 11, 12, 13\}$
$S_2=\{2, 5, 8, 11\}$
$S_6=\{2, 6, 9, 12\}$
$S_7=\{2, 7, 10, 13\}$
$S_{10}=\{3, 5, 10, 12\}$
$S_9=\{3, 6, 8, 13\}$
$S_5=\{3, 7, 9, 11\}$
$S_{13}=\{4, 5, 9, 13\}$
$S_4=\{4, 6, 10, 11\}$
$S_{12}=\{4, 7, 8, 12\}$

(b) $K=4$

図-2 $\{Si\}$ を構成した例

1. クリティカル・セクションに入ろうとするプロセス i は,Si のすべてのメンバに要求メッセージを送る.

2. i の要求メッセージを受けたプロセスは,ほかのプロセスによってすでにロックされていないならば i によってロックされ,Locked メッセージを i に返す.他のプロセスによってロックされている場合,この要求メッセージはキューに入れられるが,現在このプロセスをロックしている要求メッセージか,キューにある他の要求メッセージのほうが i の要求メッセージより古いものであれば,失敗メッセージを i に返す.そうでなければ,現在このプロセスをロックしているプロセスに対し問い合わせメッセージを送り,そのプロセスが必要なすべてのロックを得たかどうか問い合わせる.

3. 問い合わせメッセージを受けたプロセスは,すでに失敗メッセージを受けとっていれば放棄メッセージを返す.

4. 放棄メッセージを受けたプロセスは現在のロックから解放され,この要求メッセージは再びキューに入れられる.次にこのプロセスはキューの中で最も古い要求メッセージにロックされ,要求メッセージを出したプロセスに Locked メッセージが返される.

5. Si のすべてのメンバから Locked メッセージを

受けとったら，i はクリティカル・セクションに入ることができる．

6. クリティカル・セクションから i が出るときには，Si のすべてのメンバに解放メッセージを送る．

7. 解放メッセージを受けたプロセスは現在のロックから解放され，キューの中で最も古い要求メッセージにロックされる．その要求メッセージを出したプロセスに対して Locked メッセージが返される．

この方式では，$C\sqrt{N}$ (C は3から5) のメッセージで排他制御を実現することができる．

以上あげたアルゴリズムでは，クリティカル・セクションにプロセスが一つも存在しない場合にどのプロセスもきわどい部分に対する専有権(Privilege)をもたないという意味で，真に分散化されたアルゴリズムである．この性質をゆるめ，クリティカル・セクションに対する専有権をやりとりすることで，よりメッセージ量が少ないアルゴリズムをつくることができる．

Suzuki and Kasami の方法[5]では，N 個のメッセージで排他制御を実現する．

1. クリティカル・セクションに入ろうとするプロセスは，専有権トークンを持っていないならば，ほかのすべてのプロセスに Request メッセージを送る．そして専有権トークンを受けとるまで待つ．

2. Request を受けたプロセスは，専有権トークンを持っていて，しかもクリティカル・セクションに入っていない場合，専有権トークンを返す．クリティカル・セクションにいる場合には，Request を専有権トークン中に記録する．

3. 専有権トークンを受けとったプロセスは，クリティカル・セクションに入ることができる．

4. クリティカル・セクションを出るプロセスは，専有権トークンを見て，もし Request がきていれば，その中で最も古いものに対して専有権トークンを渡す．

このアルゴリズムでは，他のプロセスにはあらかじめ専有権トークンがどこにあるかがわからないためにすべてのプロセスに対して Request を送らなくてはならない．

Raymond の方法[7]，Trehel and Naimi の方法[6]では，ともに，プロセス間の通信のトポロジを unrooted tree につくることで，log N のメッセージで排他制御を実現する．

これらのアルゴリズムでは，

1. 各プロセスは自分に隣接するプロセスを知っている（そのほかのプロセスは知らなくてもよい）．

2. 各プロセスは，現在自分が専有権トークンを持っているか，あるいは，自分に隣接するどのプロセスを含む部分木（「方向」）に専有権トークンがあるかを知っている．

このとき，専有権トークンを要求するプロセスは，それが存在する方向に隣接するプロセスに Request を送る．Request は専有権トークンの存在する方向に次々に伝送され，専有権トークンをもつプロセスに到達する．

専有権トークンをもつプロセスは，Request メッセージが送られた経路を逆向きにたどってこれを伝送する．この経路上にあるプロセスは，これが自分の上を通過する際に，新たに存在する方向を記憶する．

これらの方法では，平均して木の高さ分のメッセージ伝送で排他制御が行われるため，メッセージ数は $O(\log N)$ となる．

4. 並行性制御問題

複数のプロセスが共通の資源に対して読み書きを行う場合には，その処理が実際には並列に行われていても，あたかもある順番で別々に処理が行われたような結果を生ずるべきである．このような結果を生ずるようにプロセスの進行を管理することをプロセスを直列化 (Serialize) するというが，そのための機構が並行性制御 (Concurrency Control) である．また，直列化すべき対象としてプロセスをみるとき，これを特にトランザクションということがある．

並行性制御のための機構は，ロックに基づく方法，タイムスタンプに基づく方法，オプティミスティックな方法の三つに大別することができる．

4.1 ロックに基づく方法

トランザクションが共用資源に対して一連のアクセスを行っている場合，その途中の矛盾した状態を用いてほかのトランザクションが仕事をすると矛盾が生ずる．そこで，途中の一貫性のない状態ではほかのトランザクションと資源を共用しないように，資源に関して相互排除を行えば不都合を回避できる．

すべてのトランザクションが整形 (well-formed) で2相 (two-phase) であれば，それらを直列化できることが知られている[18]．トランザクションが整形であるとは，

1. 資源にアクセスする前にはそれをロックする．

2. すでにロックされている資源はロックしない．

3. 処理を完了する前にすべてのロックを解除する．

の三つの性質をみたすことであり，2相であるとは，トランザクション中でひとたびロック解除を行ったらそれ以降はロックを要求しない，ということである．

ただし，整形かつ2相でロックを行っても，デッドロックを生じることにかわりはない．この問題に関しては本稿のおわりに触れる．

ロックに基づく方法は次にのべる二つの方法に比べてより効率的であることが知られており[16]，広く行われている[8]．

4.2 タイムスタンプに基づく方法

すべてのトランザクションにあらかじめタイムスタンプを与え，資源に対して競合するアクセスを要求する場合にはそのタイムスタンプの順にしかアクセスを許さないことでトランザクションを直列化しようとするのが，タイムスタンプに基づく方法である[9),10),13),14)]．基本タイムスタンプ方式 (Basic Timestamp Ordering) では，アルゴリズムは以下のとおりである．

1. トランザクションはその生成時にタイムスタンプを割りあてられる．資源に対するアクセス要求メッセージは，必ずこのタイムスタンプが添えられて送られる．

2. 各資源は，それに最後に書き込みを行ったトランザクションのタイムスタンプ W と，最後に読み出しを行ったトランザクションのタイムスタンプ R とを記憶する．

3. タイムスタンプ M のついた読み出し要求がきた場合，もしも $M<W$ すなわちそのトランザクションよりも新しいトランザクションがすでに書き込みを行っていれば，この要求を行ったトランザクションはアボートされる．そうでなければ読み出しが行われ，R は Max (R, M) に更新される．

4. タイムスタンプ M のついた書き込み要求がきた場合，もしも $M<$Max (R, W) すなわちそのトランザクションよりも新しいトランザクションがすでに読み出しか書き込みを行っていれば，この要求を行ったトランザクションはアボートされる．そうでなければ書き込みが行われ，W は Max (W, M) に更新される．

5. トランザクションがアボートされる場合，そのトランザクションが行った変更はすべて取り消され，トランザクションには新しいタイムスタンプがつけなおされて再起動される．

Thomas の方法 (Thomas Write Rule) では，上の4. を変更して効率化をはかっている．

4′. タイムスタンプ M のついた書き込み要求がきた場合，もしも $W<M<R$ すなわちそのトランザクションよりも新しいトランザクションが，そのトランザクションが書き換えようとするよりも古い結果をすでに読み出していれば，この要求を行ったトランザクションはアボートされる．もしも $M<W$ ならば何もしない．それ以外の場合には書き込みが行われ，W は Max(W, M) に更新される．

この方法では，誰も読み出さないうちに重ね書きされて消されてしまう書き込み要求を無視することによって，書き込みどうしの競合によるアボートの可能性を減らしている．

さらに，読み出し時のアボートを回避するための方法として，多重版 (Multiversion) 方式がある．

この方式では，資源に対して書き込みを行うことは，もとの値を書き換えてしまうのではなく，書き込んだトランザクションのタイムスタンプ W と書き込んだ内容 V の組 (W, V) を新しいバージョンとして記憶させることである．

基本タイムスタンプ方式の 3. で読み出しトランザクションがアボートされるのは，自分が読むべき古い値がすでに書き換えられて読めなくなっているためであるが，多重版方式を用いればそうしたことはなくなる．

1. タイムスタンプ M のついた読み出し要求がきた場合，M よりも小さいタイムスタンプ W をもつバージョンのうち最新のバージョンの内容が読み出される．

2. タイムスタンプ M のついた書き込み要求がきた場合，M よりも大きなタイムスタンプをもつトランザクションが，$M>W$ である最新のバージョンをすでに読み出していれば，この書き込みは拒否され，トランザクションはアボートされる．それ以外の場合には，新しいバージョンが作成される．

保守的タイムスタンプ方式 (Conservative Timestamp Ordering) は，トランザクションをアボートして再起動しなくてもよいようにするための方式である．この方式では，すべてのトランザクションからの要求をまず集め，その上で，タイムスタンプ最小の要求を処理する．これによって，より古いタイムスタンプをもつトランザクションからの要求が，すでに実行してしまった要求よりもあとから到着することを防い

でいるために，トランザクションのアボートは起こらない．しかし，この方式には三つの問題点がある．

その一つは，もしも要求を出さないトランザクションがあると，それの要求を待って実行が止まってしまうことである．

二つ目は，1番目の問題を解決するために，要求しないことを示すための「空の要求」を陽に送る必要があることである．

三つ目は，この方法では，競合する要求同士だけでなく，競合しない要求も常にタイムスタンプの順にしか実行されないということである．この問題を解決するためには，競合しうるトランザクションをあらかじめ解析し，トランザクションを分類したのちに，競合するものにだけ保守的方式を適用することが考えられる．

4.3 オプティミスティックな方法

Kung と Robinson の方法[15]では，トランザクションは競合しないことを「楽観的に」期待して実行され，最後に書き込みや読み出しが行われた資源の集合を解析することで，実際に矛盾を生ずるような競合があったかどうかを確かめようとする．もしも矛盾を生じるような競合があればトランザクションはアボートされて再起動される．コミットが完了する以前の中間結果はトランザクションごとのプライベートなコピー上におかれ，ほかのトランザクションから見えることはないようになっている．

並行性制御の方式は，大別して以上3種類であるが，一般にロックに基づく方式が最も効率的であることが知られている．

5. デッドロックの検出と解除

複数のプロセスが互いにほかのプロセスのもつロックを要求して無限の待ち状態になることをデッドロックという[17]が，集中システムにおけるデッドロック問題に対する対処法はすでによく知られている．基本的には三つの方式がある．

1. デッドロックを検出し，それを解除する．
2. システムの状態表を管理し，それを解析することによってデッドロックを避ける．
3. システムの構造をデッドロックが生じないように構成する．

分散システムにおいてもこれらの方式が基本となるが，3の方式はシステムの柔軟性の面から考えると困難である．分散データベースにおいては，1の方式が一般にとられる[19]．この場合にはシステムにおける wait-for-relationship の状態を保持している必要がある．2の方式も適用可能ではあるが，デッドロックを防止するために資源利用に過度の制限が課されるため，あまり得策とはいえない．結局，1のデッドロック検出・解除方式が最も一般的と判断される．

分散システムにおけるデッドロックの検出方式としては，集中方式，階層方式，分散方式がある．集中方式は一見簡単そうにみえるが，必ずしも得策ではなく，分散方式のほうが望ましい場合が多い．また，別の分類として，デッドロックの検出を定期的に行うか，デッドロックの可能性が生じた時点でのみ行うかの区別がある．これも一見定期的に行うのが簡単そうであるが，後者（連続的検出方式）のほうが優れている．

デッドロック・モデル

デッドロック解析のためのモデルは分散プロセスモデルである．システムはプロセスの網からなる．各プロセスはどのプロセスに要求メッセージを送ったかを知っており，どのプロセスからのメッセージを待っているかを知っている．この待ち条件によって，次の三つのモデルが存在しうる．

AND モデルは各プロセスの待ち条件がすべて AND の形で表現されるシステムのモデルである．この場合にはサイクルの存在がデッドロック存在の必要十分条件である．OR モデルは各プロセスの待ち条件がすべて OR の形で表現されるシステムのモデルである．この場合にはノットの存在がデッドロック存在の必要十分条件である．AND-OR モデルはこれらの混在するシステムであり，より一般的である．

AND-OR モデルでのデッドロック検出のための分散型連続検出方式を紹介する．

1. 各プロセスは要求メッセージを送る際，デッドロック検出メッセージも一つの対として相手方プロセスに送る．このとき，AND エッジに対しては，この要求メッセージの相手プロセスの名前を添えて送る．OR エッジに対しては，このプロセスのすべての OR エッジの相手先となっているプロセスの名前を添えて送る．

2. デッドロック検出メッセージを受け取ったプロセスは，そのプロセスが待ち状態にいなければ，検出メッセージをただちに消滅させる（この場合には，少なくともここで待ち関係が切れていることが明らかである）．もしもほかのプロセスのサービス完了を待っ

ている場合には，AND エッジに対しては，相手先プロセスの名前を添えて，この相手先プロセスに検出メッセージを転送する．OR エッジに対しては，このプロセスのすべての OR エッジの相手先プロセスの名前を添えて，すべての相手先プロセスに検出メッセージを転送する．

ただし，デッドロック検出メッセージが，どのような経路を経てきているかは検出メッセージに記録されているので，転送の際には，以下のような制限を行い，無駄な転送を防ぐ．

a．AND エッジの場合

デッドロック検出メッセージがその最初のプロセス（送り元）以外のプロセスに再び転送されてきた場合には，そのデッドロック検出メッセージをただちに消滅させる（送り元のプロセスがこのデッドロックを検出する）．

b．OR エッジの場合

デッドロック検出メッセージがある OR エッジをすでに通過している場合には，そのエッジに再びデッドロック検出メッセージを送る必要はない．

c．検出メッセージを送るべきエッジが一つも存在しない場合には，検出メッセージを消滅させる．

3. もしもデッドロック検出メッセージが最初のプロセス（送り元）に戻ってきた場合には，以下の処置をする．

a．もしもこの出発点のプロセスが他のプロセスのサービス完了を待つ待ち状態になければ，デッドロックの可能性は解消されているので，検出メッセージを消滅し，通常の実行を続ける．

b．デッドロック検出メッセージに記録されているメッセージの通過経路がグラフ上でノットを構成していれば，デッドロックが存在することは明らかである．すなわち，相手先プロセスの集合と，メッセージ（通常複数個ある）が通過したプロセスの集合が一致すれば，デッドロックが存在する．

ただし，AND エッジがあるため，いくつものノットが存在しうる．一つでもノットがあれば，それに含まれているプロセスがデッドロックに含まれていることになる．このことから，次のような処置をすればよいことがわかる．

b-1. 出発点のプロセスが待ち状態にあるときは，戻ってきたメッセージとすでに前に戻ってきて保持されているメッセージを合わせて調べ，ノットが存在している場合にはデッドロックであるので，デッドロックを解除するために必要なプロセスをアボートする．

b-2. ノットが検出されないときには，そのメッセージを保持しておく．

6. おわりに

本稿では，分散処理システム上でのプロセスの同時実行制御に関するさまざまな問題の解決に必要な基本的アルゴリズムを紹介したが，紙面の制約上，すべてをつくすということはできなかった．より詳しく知りたい読者は，参考書[1]などを参照されたい．

参考文献

1) Maekawa, M. et al.: Operating Systems: Advanced Concepts, Benjamin/Cummings Pub. Co., Feb. (1987).
2) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, Comm. ACM, Vol. 21, No. 7, pp. 558-565 (1978).
3) Ricart, G. and Agrawala, A. K.: An Optimal Algorithm for Mutual Exclusion in Computer Networks, Comm. ACM, Vol. 24, No. 1, pp. 9-17 (1981).
4) Maekawa, M.: A Square-root (N) Algorithm for Mutual Exclusion in Decentralized Systems, ACM Trans. Comput. Syst., Vol. 3, No. 2, pp. 145-159 (1985).
5) Suzuki, I. and Kasami, T.: A Distributed Mutual Exclusion Algorithm, ACM Trans. Comput. Syst., Vol. 3, No. 4, pp. 344-349 (1985).
6) Trehel, M. and Naimi, M.: A Log (n) Algorithm for Mutual Exclusion on a Distributed System, Technical Report, Centre de Recherche Micro Systems et Robotique, Universite de Franche-Comte-Besancon (1986).
7) Raymond, K.: A Tree-Based Algorithm for Distributed Mutual Exclusion, Dept. Computer Science, University of Melbourne (1986).
8) Menasce, D. A. et al.: A Locking Protocol for Resource Coordination in Distributed Databases, ACM Trans. Database Syst., Vol. 5, No. 2, pp. 103-138 (1980).
9) Bernstein, P. A. et al.: Concurrency Control in a System for Distributed Databases (SDD-1), ACM Trans. Database Syst., Vol. 5, No. 1, pp. 18-51 (1980).
10) Bernstein, P. A. and Shipman, D. W.: The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1), ACM Trans. Database Syst., Vol. 5, No. 1, pp. 52-68 (1980).
11) Gifford, D. K.: Weighted Voting for Repli-

cated Data, in, Proc. of the 7th Symp. on Operating Systems Principles, ACM, NY, pp. 150-162 (1979).
12) Thomas, R. H.: A Majority Consensus Approach to Concurrecy Control for Multiple Copy Databases, ACM Trans. Database Syst., Vol. 4, No. 2, pp. 180-209 (1979).
13) Bernstein, P. A. and Goodman, N.: Concurrency Control in Distributed Database Systems", ACM Comput. Surveys, Vol. 13, No. 2, pp. 185-221 (1981).
14) Bernstein, P. A. and Goodman, N.: Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems, Proc. 6th Int. Conf. Very Large Databases, pp. 285-300 (1980).
15) Kung, H. T. and Robinson, J. T.: On Optimistic Methods for Concurrency Control, ACM Trans. Database Syst., Vol. 6, No. 2, pp. 213-226 (1981).
16) Agrawal, R. and Dewitt, D. J.: Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation, ACM Trans. Database Syst., Vol. 10, No. 4, pp. 529-564 (1985).
17) Agrawal, R. et al.: Deadlock Detection is Cheap, ACM SIGMOD Record, Vol. 13, No. 2, pp. 19-34 (1983).
18) Eswaran, K. P. et al.: The Notions of Consistency and Predicate Locks in a Database System, CACM, Vol. 19, No. 11, pp. 624-633 (1976).
19) Menasce, D. A. and Muntz, R. R.: Locking and Deadlock Detection in Distributed Databases, IEEE Trans. Softw. Eng., Vol. SE-5, No. 3, pp. 195-201 (1979).
20) Albert, A. A. and Sandler, R.: An Introduction to Finite Projective Planes, Holt, Rinehart and Winston, NY (1968).

(昭和61年12月17日受付)