並行オブジェクト指向言語における再帰にともなうデッドロックの回避機構

柳川 和久　佐藤 直人　大澤 範高　弓場 敏嗣
電気通信大学大学院情報システム学研究科

あらまし:

並行オブジェクト指向言語でオブジェクト内で複数のメソッド起動を許す場合, 一般にメソッド間の相互干渉のためオブジェクトの一貫性が保証されない. この場合, オブジェクトの一貫性を保つためには相互排除, マルチバージョニングなどの手法が用いられる. 相互排除を行なう場合, 排他的なメソッドの実行の結果, そのメソッドが属するオブジェクト自身にメッセージが送られる場合にデッドロックが生じる.

本論文では並行オブジェクト指向言語における再帰にともなうデッドロックを排除する機構を提案する. アクセス制御におけるキー/ロック法を適用することにより, 従来の手法よりも単純で, 効率の良い機構が実現されることを示す.

キーワード: 並行オブジェクト, 再帰, デッドロック

Avoidance of Recursive Deadlock in Concurrent Object-Oriented Programming

K. Yanagawa　N. Sato　N. Osawa　T. Yuba
Graduate School of Information Systems, the University of Electro-Communications

Abstract:

In concurrent object-oriented languages, intra-object concurrency sometimes breaks object's consistency. For consistency control, a mutual exclusion method is generally used, causes deadlock when recursive message is sent to the object originates that message.

In this paper, we propose a method to avoid deadlock in a recursive program, based on key/lock method in access control. This method is simpe and efficient.

Keywords: Concurrent Object, Recursion, Deadlock

1 はじめに

分散メモリ型の並列計算機に適合した計算モデルの一つに並行オブジェクト指向モデルがある. このモデルでは, 独立して並行動作するオブジェクト間のメッセージ交換により計算が進行する.

オブジェクト内の並列性を持つモデルでは, 一度に複数のメッセージが同時に処理され得る. この場合, オブジェクトの一貫性を保つために何らかの処理を行う必要がある. マルチバージョニング等の手法も用いられてはいるが [6], 多くは相互排除を行う方法を採用しているか [2,4,8], もしくはモニタ型の一度に一つのメッセージを処理する方法を採用している [5,10].

相互排除を用いる場合やモニタ型の動作をする場合, 起動されたメソッドのメッセージ送信により, メソッドを起動したオブジェクトにメッセージが送られてくる場合にデッドロックをひき起こす場合がある. 例えば隣接しているオブジェクトに問い合わせ

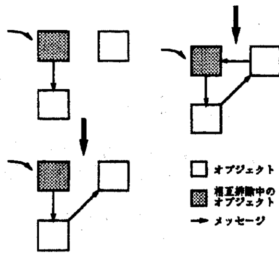

図 1. 再帰にともなうデッドロック

を行い, その返事を受けて計算結果を返すグラフアルゴリズムでは 図 1 のように再帰的なメッセージが生じ得る. このような再帰的なメッセージによるデッドロックを回避する手法として, [1,2] 等で提案された手法が知られている.

本研究ではこの問題に対処する別の手法を提案する. この手法は従来の手法と比べ非常に単純であり, より効率的な実装ができることがわかっている. 以下ではまずデッドロック回避手法を説明し, 次に計算量の削減法を述べる. その後, 関連研究との比較を行い, まとめと今後の課題を述べる.

2 キー/ロック法によるデッドロック回避

ここではオブジェクトは一度に一つのメッセージを受け取り, 一つのメッセージに対して一つのメソッドが起動されるモデルを考える. ただし起動されたメソッドが相互排除を必要としない場合, メソッドが終了しなくても次のメッセージを受け取ることができる.

一般に, 再帰にともなうデッドロックを回避するには, 実行中の排他メソッドに由来するメッセージに何らかの鍵を添付し, 相互排除中も鍵を持つメッセージは処理するようにすることが行われる.

提案する手法 (キー/ロック法と呼ぶ) ではこのアルゴリズムを直接利用する. 以下で「排他メソッド」とは相互排除を必要とするメソッドのことである:

1. 排他メソッド実行時にはオブジェクトをロック, ロックに対応した鍵が作られる.
2. 送信メッセージには鍵を添付する.
3. 鍵付きメッセージを処理する場合, 送信メッセージには鍵を添付する.
4. ロックに対応した鍵を持つメッセージはロックを「通り抜けて」処理される.

図 2 はこのアルゴリズムを模式的に示したものである. ここでコールとは返事を待つ同期型のメッセージ送信のことである.

はじめに O がメッセージを受け取り排他メソッド o を起動する. このとき O は鍵を生成する (図 2 の 1). o は P にコールを行う. P はメッセージを受け取りメソッド p を起動する. このときメッセージには鍵が添付されている (図 2 の 2). p が O にコールを行う. このときメッセージには o から受け取った鍵が含まれている (図 2 の 3). O は p からのメッセージが鍵を持っているのでメソッド o' を起動する (図 2 の 4). その後は o', p, o が順次実行を返事を返して終了し, 最後には全てのメソッドが終了する (図 2 の 5).

排他メソッドの起動 (図 2 の 1) 以後に鍵を持たないメッセージを O が受け取った場合, 排他メソッドが終了するまで O はそのメッセージに対するメソッドを起動しない. この手法をモデル化した例を付録 A に示してある.

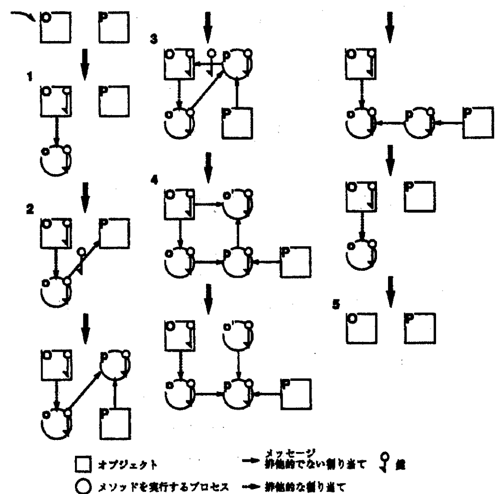

図 2. キー/ロック法

3 計算量の改善

生成順の鍵のリストの性質からメッセージ中の鍵の位置が特定できる. また非同期メッセージはプロセスをブロックしないため, 鍵を与える必要がないことからオブジェクト側の情報量を削減できる. 以下に計算量削減の概要を示す.

素朴に実装すると排他メソッド呼出しのたびに新たな鍵が作られ, オブジェクトと送信メッセージに記録されるので, オブジェクトとメッセージの双方

が $O(n)$ の情報を持つ必要がある．またメッセージを受信したオブジェクトは $O(n)$ の鍵のリストに対して照合を行わなければならない．

生成された順に鍵を付け足していったリストを考える．任意の二つの鍵のリスト $L1 = x_1x_2 \ldots x_n$ と $L2 = y_1y_2 \ldots y_n$ に対し，

$$x_n = y_n \rightarrow \forall k \in [1, n-1][x_k = y_k]. \qquad (1)$$

がいえる．すなわち任意の二つの鍵のリストについて，ある部分が等しいならばその部分までの prefix はすべて等しい．この証明は付録 B に示す．

これを利用すると以下のことが可能になる:

- オブジェクト側が自分が付けた鍵の位置を覚えておくことにより照合を定数時間で行えるようにできる．
- 同じオブジェクトが生成した鍵が同じ位置に来るように書き換えることで，メッセージに付ける鍵のリストの長さを減らすことができる．リストの書き換え自体の手間は，新たに生成した鍵を以前に自分が生成して付加した鍵の位置に付加するようにするだけですむ．

非同期メッセージには鍵を付けないことにすると，オブジェクトが持つ情報をカウンタにできる．これは同期メッセージ送信が返答を受け取るまでブロックすることから次のことがいえるからである:

- 実行されるべき再帰的なメッセージがブロックされることがない．ブロックされる場合カウンタが動いていることになるが，
  1. 増えた場合には，既に別の再帰的なメッセージのによる排他メソッドが起動されているのでブロックされなければならない．
  2. 減るのは再帰的なメッセージの原因となるメッセージを発したメソッドが終了した時であるが，これは再帰的なメッセージが処理されない限り終了しない．
- 実行されてはいけないメッセージはブロックされる．

図 2 の例で説明する．はじめは O, P ともに カウンタが 0 であり，[Q1] なる鍵のリストを持つメッセージを O が受け取ったとする．1 の段階で O はカウンタを 1 にし，鍵 O1 を生成する．2 の段階で送信されるメッセージには [Q1,O1] なる鍵のリストが付けられ，O は鍵をリストの 2 番目に付けたことを記録する．P に送られたメッセージが排他メソッドのコールでないなら 3 の段階で O に送られるメッセージには [Q1,O1] という鍵のリストが付けられている．O はリストの 2 番目を見て鍵 O1 を見つけ，自分が作ったカウンタの値 1 に対する鍵であることを確認してこのメッセージを処理する．

4 従来の手法との計算量の比較

以下の指標を用いる:

n: コールチェインの長さ

k: コールチェイン中の並行コールの回数

ただしコールチェインとはコールの連鎖のことであり，A が B をコールし，その処理中に B が C をコールした場合 $n = 2$ となる．並行コールとは複数のコールを同時に行い，全ての返答が返ってくるまで待つという動作である．

同じ言語非依存であり，プログラマに対して透過なフレームワークである [1] で提案された手法と比較する．同論文で提案された二つの手法では，時間計算量は以下で与えられている:

Multi-ported Objects
送信: $O(n)$, 受信: $O(1)$.

Named-Threads
送信: $O(1)$, 受信: $O(k)$.

また空間計算量は，multi-ported object ではオブジェクトとメッセージの双方で $O(n)$, named-thread では $O(k)$ である．このため [1] では少数のオブジェクトの緊密で並行コールが多い環境では multi-ported object が，逆に多数のオブジェクトで並行コールが少ない環境では named-thread が向いていると結論している．

一方われわれの与えたキー/ロック法での時間計算量は，3 節で示した改善を施すと，

送信: $O(1)$, 受信: $O(1)$

となる．また空間計算量は，

オブジェクトが持つ情報量: $O(1)$

メッセージが持つ情報量: $O(n)$

とすることができる．結果として [1] のどちらの手法と比べても，時間計算量は減少する．

空間計算量ではメッセージについては multi-ported object とは同等であり，named-thread と比較すると少数のオブジェクト間の並行コールが多い環境ではオーバヘッドが小さくなり，その逆の場合には大きくなる．ただし multi-ported object とキー/ロック法ではオーバヘッドが最大でもコール

チェインに含まれるオブジェクトの数に抑えられるのに対し, named-thread では上限がない.

オブジェクト側でのオーバヘッドについては, 情報量はキー/ロック法ではカウンタを用いることができるので定数にできる. 一方 [1] の手法ではどちらも新たな排他メソッドを起動するたびに現在の情報をスタックにプッシュし, 新たな情報を記録する必要がある.

表 1. 計算量の比較のまとめ

	送信	受信	情報量 (mes/obj)
MPO	$O(n)$	$O(1)$	$O(n)/O(n)$
NT	$O(1)$	$O(k)$	$O(k)/O(k)$
K/L	$O(1)$	$O(1)$	$O(n)/O(1)$

MPO: Multi-Ported Objects
NT: Named-Threads
K/L: Key/Lock method

5 関連研究

いくつかの並行オブジェクト指向言語では, 直接的な再帰に対しこれを回避する手法を実装している. 例えば ABCL/1 [10] では, オブジェクトはメソッドでないルーチンを持つ. cooC [8] では, 直接的な自分自身へのメッセージ送信を関数呼び出しに置き換えることによりデッドロックを避けている. Obliq [2] では, メソッドの引数 self が自分と等しいメソッドからのメッセージではロックを要求しないようにすることによってこれを実現している. しかしこれらの言語では, 複数のオブジェクト間での相互再帰によるデッドロックは避けられない.

COOL [4] ではコールの際には送信側が相互排除を解除し, 返答が得られるまで中断するという手法をとっている. この場合, 再帰が起こり得るコールについて, プログラマが相互排除の解除を明示する必要がある.

メッセージの受信とディスパッチをボディ手続きとして記述できる言語 [3, 7] や選択的なメッセージの受信を明示できる言語 [10] では, プログラマが再帰的なメッセージを識別し, 特別な処理の記述を行うことで再帰にともなうデッドロックを避けられる. しかしこれを実現するには全てのメッセージ送信パターンを知る必要がある.

一方 [1] では本論文のモデルと同様の動作をするオブジェクトについて, 言語に依存しないプログラマに対して透過なデッドロック回避手法として, multi-ported object という手法と named-thread という手法を提案している. どちらの手法も複数のオブジェクト間の相互再帰を扱える.

6 おわりに

並行オブジェクト指向言語における再帰にともなうデッドロックの回避手法を提案した. 提案した手法ではロックに対応した鍵を用いて, 実行中の排他メソッドに由来するメッセージを識別する. 生成された順に沿った二つの鍵のリストはある部分が等しいならばその部分までの prefix がすべて等しいという性質を利用することで, 鍵の照合にかかる時間を $O(1)$ とし, また非同期メッセージに鍵を付けないことでオブジェクト側の情報量も $O(1)$ とすることができた.

今後の課題としては, 適当な言語へのキー/ロック法の実装と, 有効性の検証を行うことがあげられる. 特にオーバヘッドが実用の範囲内にあることの確認は不可欠であると考えている.

7 謝辞

この研究を行う上で, 数々の参考となる助言をいただいた弓場研究室の諸氏に感謝いたします.

参考文献

[1] E. A. Brewer and C. A. Waldspurger. Preventing Recursion Deadlock In Concurrent Object-Oriented Systems. Technical Report MIT-LCS//MIT/LCS/TR-526, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1992.

[2] L. Cardelli. A Language with Distributed Scope. http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers/Obliq.A4.ps, May 1995.

[3] D. Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, Vol. 36, No. 9, pp. 90–102, September 1993.

[4] R. Chandra, A. Gupta, and J. L. Hennesy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, Vol. 27, No. 8, pp. 13–26, August 1994.

[5] A. S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. In D. B. Skillicorn and D. Talia, edited, *Programming Languages for Parallel Processing*, Chapter 4, pp. 179–191. IEEE Computer Society Press, 1995.

[6] T. Hirotsu, H. Fujii, and M. Tokoro. A Multiversion Concurrent Object Model for Distributed and Multiuser Environments. *Proceedings of The 15th International Conference on Distributed Computing Systems*, pp. 271–278, 1995.

[7] M. Karaorman and J. Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM*, Vol. 36, No. 9, pp. 103–116, September 1993.

[8] R. Trehan, 澤島信介, 森下明, 友田一郎, 井上淳, 前田賢一. Concurrent Object Oriented 'C' (COOC). 情報処理学会研究会報告 92-PRG-7, pp. 17–24, 1992.

[9] 柳川和久, 大澤範高, 弓場敏嗣. 並行オブジェクト指向言語における再帰にともなうデッドロックの回避機構. 情報処理学会第 54 回全国大会講演論文集, Vol. 1, pp. 337–338, March 1997.

[10] A. Yonezawa, edited. *ABCL: An Object-Oriented Concurret System – Theory, Language, Programing, Implementation and Application.* The MIT Press, 1990.

付録 A　プロセスモデル

<table>
<tr><td>p!a</td><td>a の値を p に送信</td></tr>
<tr><td>p?a</td><td>p で受信した値を a に束縛</td></tr>
<tr><td>$p</td><td>ポート p を生成</td></tr>
<tr><td>P;Q</td><td>P と Q の逐次合成</td></tr>
<tr><td>P|Q</td><td>P と Q の並列合成</td></tr>
<tr><td>P#Q</td><td>P と Q の（ガード付き）選択</td></tr>
<tr><td>{list}</td><td>list 内の式を不可分に実行</td></tr>
<tr><td>name(args)</td><td>引数 args を持つプロセス</td></tr>
<tr><td>(g:P)</td><td>ガード g を持つプロセス</td></tr>
<tr><td>D@E</td><td>D もしくは E である定義</td></tr>
<tr><td><list></td><td>list 内の変数およびプロセスを持つドメイン</td></tr>
</table>

通信は非同期で，かつバッファリングされているものとする．よってメッセージの送信 `x!a` はいつでも実行することができ，対応する受信 `x?a` が実行されなくても終了する．同一ドメイン内のプロセスはドメインで定義された変数を共有する．プロセス内で導入された変数のスコープはプロセス内である．

オブジェクトは細部を無視すると，メッセージのディスパッチャプロセスとメソッドのプロセスの記述によりモデル化できる．デッドロックを起こす場合は以下のようになる：

```
object = <
  n = 0.

  disp(p)
  = p?(m,r);
    ((m is non exclusive:
      (({n:=n+1};meth(m,r);{n:=n-1})
       |disp(p)))
     # (m is exclusive:
        (wait(n=0);meth(m,r);disp(p)))).

  meth(m,r)
  = m':=pre(m);$r';t!(m',r');
    r'?a';a:=post(a');r!a.
  @ a:=proc(m);r!a.
>
```

ここで pre, post, proc などはメソッドの動作を抽象化したプロセスであり，`wait` は引数の条件が満たされるまでサスペンドするプロセスである．

```
O = new object.
P = new object

  O.disp(o)|P.disp(p)|o!(m,r)
= O.(wait(n=0);meth(m,r);disp(o))
  | P.disp(p)
= O.(meth(m,r);disp(o))|P.disp(p)
= O.((m':=pre(m);$r';p!(m',r');
      r'?a';a:=post(a');r!a);disp(o))
  | P.(p)
= O.((r'?a';a:=post(a');r!a);disp(o))
  | P.(wait(n=0);meth(m',r');disp(p))
= O.((r'?a';a:=post(a');r!a);disp(o))
  | P.((m'':=pre(m');$r'';o!(m'',r'');
        r''?a'';a':=post(a'');r'!a');
       disp(p))
= O.((r'?a';a:=post(a');r!a);disp(o))
  | P.((r''?a'';a':=post(a'');r'!a');
       disp(p)).
```

この場合ディスパッチャプロセスが動いていないのでデッドロックする.

デッドロック回避を含めると以下のように記述できる:

```
object = <
  n = 0.
  e = 0.
  keys = [].

  disp(p)
  = p?(k,m,r);
    ((e = 0:disp'(p,k,m,r))
     # (e = 1:
        ((top(keys) in k:disp'(p,k,m,r))
         # (top(keys) not in k:
            (p!(k,m,r)|disp(p)))))).

  disp'(p,k,m,r)
  = ((m is non exclusive:
      (({n:=n+1};meth(k,m,r);{n:=n-1})
       |disp(p)))
     # (m is exclusive:
        (wait(n=0);e:=1;k':=key();
         push(keys,k');push(k,k');
         ((meth(k,m,r);pop(keys);e:=0)
         |disp(p))))).

  meth(k,m,r)
  = m':=pre(m);$r';t!(k,m',r');
    r'?a';a:=post(a');r!a.
  @ a:=proc(m);r!a.
>
```

ここで [] は空のリストを, push, top, pop はリストに対するスタック操作をそれぞれ表す. また key はシステム全体で unique な鍵を生成する.

```
O = new object.
P = new object.

  O.disp(o)|P.disp(p)|o!(k,m,r)
= O.(disp'(o,k,m,r)) | P.disp(p)
= O.((wait(n=0);e:=1;k':=key();
      push(keys,k');push(k,k');
      ((meth(k,m,r);pop(keys);e:=0)
       |disp(o))))
  | P.disp(p)
= O.((m':=pre(m);$r';p!(k,m',r');
      r'?a';a:=post(a');r!a)
     |disp(o))
  | P.disp(p)
= O.(((r'?a';a:=post(a');r!a);
      pop(keys);e:=0)|disp(o))
  | P.((wait(n=0);e:=1;k'':=key();
          push(keys,k'');push(k',k'');
          ((meth(k',m,r);pop(keys);e:=0)
          |disp(p))))
= O.(((r'?a';a:=post(a');r!a);
       pop(keys);e:=0)|disp(o))
   | P.(((m'':=pre(m');$r'';o!(k'',m'',r'');
          r''?a'';a':=post(a'');r'!a');
          pop(keys);e:=0)|disp(p))
= O.(((r'?a';a:=post(a');r!a);
       pop(keys);e:=0)|disp'(o,k'',m'',r''))
   | P.(((r''?a'';a':=post(a'');r'!a')
          pop(keys);e:=0)|disp(p))
= O.(((r'?a';a:=post(a');r!a);
        pop(keys);e:=0)
       |(({n:=n+1};meth(k'',m'',r'');
          {n:=n-1})|disp(o)))
   | P.(((r''?a'';a':=post(a'');r'!a')
          pop(keys);e:=0)|disp(p))
= O.(((r'?a';a:=post(a');r!a);
        pop(keys);e:=0)
       |(((a''':=proc(m'');r''!a''');
           {n:=n-1})|disp(o)))
   | P.(((r''?a'';a':=post(a'');r'!a')
          pop(keys);e:=0)|disp(p))
= O.(((r'?a';a:=post(a');r!a);
        pop(keys);e:=0)|disp(o))
   | P.disp(p)
= O.disp(o)|P.disp(p).
```

この場合全てのメソッドプロセスが終了する.

付録 B　式 1 の証明

帰納法を用いて $x_n = y_n \rightarrow x_{n-k} = y_{n-k}$ を示す.

$k = 1$ の時に $x_n = y_n \wedge x_{n-1} \neq y_{n-1}$ とする. これは x_n と y_n を発したプロセスに対するメッセージが異なっていることを表す. 本論文ではオブジェクトが一つのメッセージに対し一つのメソッドを起動することを仮定している. このため次のことがいえる:

- 異なったメッセージに対し同じ鍵を生成することはない.
- 一つのプロセスが複数のメッセージを受け取って起動されることはない.

よってこのような場合に同じ鍵が与えられることはないので, これはあり得ない. よって $x_n = y_n \rightarrow x_{n-1} = y_{n-1}$.

ある k に対し $i \in [1, n-(k-1)]$ までで $x_{n-k} = y_{n-k}$ だったとする. $i = 1$ の場合 (上記) と同様にして k の時にも $x_{n-(k-1)} = y_{n-(k-1)} \rightarrow x_{n-k} = y_{n-k}$ がいえる.