マルチメディア通信と分散処理 69−17
（1995. 3. 3）

共有メモリ型マルチプロセッサを用いた通信プロトコルの並列処理方式

加藤 聰彦　　鈴木 健二
国際電信電話(株) 研究所

近年、伝送路の高速化に伴い、通信プロトコル処理の高性能化が重要な課題となっている。また、最近のワークステーションでは、メモリを共有するマルチプロセッサ構成が広く採用されている。このため、共有メモリ型マルチプロセッサ上での通信プロトコルの並列処理方式が重要となっている。そこで筆者らは、オペレーティングシステムのサポートするスレッドを用いて、1つのレイヤのプロトコル処理を個別のプロセッサを割り当てる方式を検討している。本稿では、筆者らのプロトコルの並列処理方式に関して、スレッド間での同期を極力避けるためのレイヤ間のインタフェース方式やバッファ管理方式などの詳細設計について述べる。

A Parallel Processing Method for Communication Protocols using Shared Memory Multiprocessor

Toshihiko KATO and Kenji SUZUKI
KDD R & D Laboratories

As the network transmission bandwidth increases, the high performance implementation of communication protocols is strongly required. We have been studying a per-layer based parallel processing method for protocols using widely available shared memory multiprocessors. Our method executes the processing of a protocol on a kernel supported thread executed on separate processors. It has adopted a layer interface and buffer management which avoid the synchronization between threads in order to reduce the overhead of parallel processing. This paper describes the detailed design of our method.

1. はじめに

近年、100Mbpsを越える超高速ネットワークの導入に伴い、それらに接続されるコンピュータにおいて、伝送速度に対応する高性能な通信プロトコル処理の実現が求められている。一方、最近のワークステーションでは、処理の高速化を目的として、メモリを共有するマルチプロセッサ構成が広く採用されている。そこで、市販のワークステーション上で高速な通信プロトコル処理を行なうためには、共有メモリ型マルチプロセッサ上での通信プロトコルの並列処理方式の検討が重要であると考えられる。

このような研究は従来からも行なわれており、1つのメッセージに対する複数レイヤの処理を、1つのプロセッサに割り当てるProcesser-per-messageと呼ばれる方式が用いられている[1,2]。これらの研究では、並列化に伴うオーバヘッドを考慮すると複数レイヤの処理が並列化の単位として適当であること、プロセッサ数に適応した負荷の配分が容易であることなどの理由で、本方式を

採用している。しかし本方式では、コネクション管理テーブルなどのプロセッサ間で共有される情報にアクセスするために、プロセス間で同期をとる必要があり、そのオーバヘッドが無視できないという結果が報告されている[2]。

これに対し筆者らは、レイヤごとに並列処理を行なうProcesser-per-layer方式を採用し、オペレーティングシステムのサポートするスレッドを用いて、1つのレイヤのプロトコル処理を1つのプロセッサを割り当てる方法[3]を検討している。この方法は、レイヤ構造に対応した自然な並列化を行なうことが可能であるが、Processer-per-message方式に比べて並列化の単位が小さいため、その実装において、レイヤ間のインタフェースやバッファ管理でのスレッド間の同期など、並列化に伴うオーバヘッドを極力避けることが重要となる。本稿では、筆者らが採用した通信プロトコルの並列処理の実現方式の詳細について示す。以下、2.において本方式の設計方針を示し、3.においてその詳細設計を述べ、4.で筆者らの方式を考察する。

2. 設計方針

共有メモリ型マルチプロセッサ上で、通信プロトコルの並列処理を実現するために、以下の方針を採用した。

(1) 1つのレイヤのプロトコル処理を、オペレーティングシステムでサポートされているスレッドにより実現し、各スレッドを個別のプロセッサに割り当てる。

(2) プリミティブやPDU (Protocol Data Unit)は、ヒープ領域に確保されたバッファ領域上に作成し、複数レイヤのスレッドから参照可能とする。

(3) 各レイヤのスレッドが他のレイヤとできる限り独立に動作可能とするために、レイヤ間のインタフェース方式や、プリミティブ/PDU用バッファへのアクセスの制御、バッファ領域の管理方式などにおいて、できるだけレイヤ間で同期をとらない方法を採用する。

(4) レイヤのスレッド間で同期をとる場合も、カーネルのサポートする条件変数、スレッドをスリープさせるロック、1つのスレッドが繰り返しロックを要求するスピンロックなどを使い分け、処理のオーバヘッドをできるだけ小さくする。

(5) プロセッサ数の小さいワークステーションにも対応するために、1つのプロセッサに複数のスレッドを割り当てることも可能とする。

3. 詳細設計

上記の方針に基づいた通信プロトコルの並列処理の実現方式の概要と、レイヤ間のインタフェース方式、プリミティブ/PDU用バッファへのアクセスの制御方式、バッファ領域管理方式について示す。

3.1 概要

図1に並列処理を行なうプログラムの全体構成を示す。プログラム全体は応用プログラムとリンクされるライブラリとして実現され、その中で各レイヤがスレッドとして実行される。レイヤ間でやり取りされるプリミティブや各レイヤで処理されるPDUは、バッファ領域上に作成される。

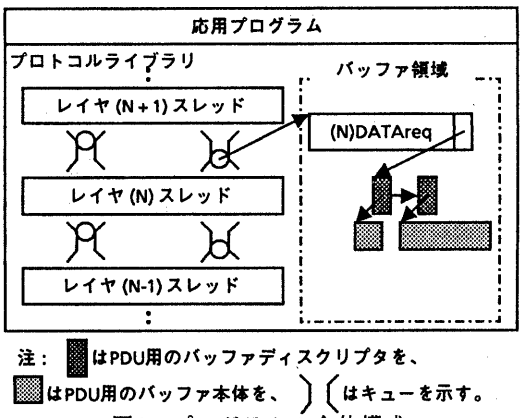

図1　プログラムの全体構成

各レイヤのスレッドの間には、バッファ領域上のプリミティブへのポインタを通知するキューが設けられる。各レイヤのスレッドは通常、キューにつながれたプリミティブを読み出し、それを処理し、必要に応じて出力のプリミティブをキューに書き込むという動作を、他のスレッドとは並列に行なう。キューが空で読み出せない場合またはキューがフルで書き込めない場合は、他のレイヤのスレッドが書き込み/読み込みを行なうのを待つために、レイヤ間の同期をとる。

3.2 レイヤ間のインタフェース方式

3.2.1 レイヤ間のキューの実現法

レイヤ間のキューについては、レイヤのスレッドがロックをかけることなしに、並行して

キューの書き込みと読み出しができるように、以下のような方法を用いる(図2参照)。

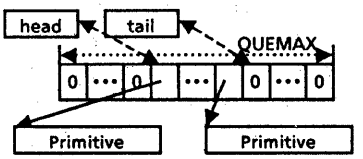

図2　キューの構成

(1)キューはプリミティブへのポインタを持つ固定長(QUEMAX)の配列と、読み込む要素と次に書き込む要素を示す変数(headとtail)から構成される。プリミティブがつながれていない配列の要素は0となっている。

(2)書き込む側のスレッドはtailのみにアクセスし、tailの次の要素((tail+1)%QUEMAX)が0である場合は、キューが書き込み可能であると判断し、tailの要素にプリミティブのポインタを代入し、tailを1進める。一方、tailの次の要素にすでにプリミティブのポインタが代入されている場合は、キューがフルであると判断し、書き込み失敗とする。なお、キューに書き込めなかったプリミティブはレイヤ内部で保持する必要がある。

(3)読み出す側のスレッドは、headの要素が0でなければそのポインタを読み出し、headの要素を0とし、headを1進める((head+1)%QUEMAXをheadに代入する)。

3.2.2 レイヤの同期の実現法

前述のように、各レイヤのスレッドは、空のキューへのプリミティブの到着またはフルとなったキューからの読み出しを待つ必要がある。さらに、各レイヤのスレッドは上位と下位に対して2組のキューを持つ。

このため、レイヤ間の同期を、各レイヤのメインプログラムにおいて、図3に示すように、ロックと条件変数を用いて実現することとした。

(1)各レイヤは、ロックと条件変数から構成される同期用の外部変数(nSynchなど)を持つ。

(2)レイヤ(N)スレッドは、上位と下位のキューが空でないか、内部で保持したプリミティブをキューに書き込むことが可能である場合は、常にキューからのプリミティブの読み込みとその処理(図中①)または保持したプリミティブの書き込み(図中②)を行なう。

(3)このとき、同期を待っている隣接するレイヤのスレッドを起動するために、同期用の外部変数

```
struct { mutex_t lock; condition_t wakeUp;
}nSynch, nPlus1Synch;
NLayerMain(){
  for(;;){
    while(!上位/下位のキューが空||書き込み不可){
      if(上位のキューから読み込み可能){          …①
        プリミティブを読みこみ;
        condition_signal(nPlus1Synch->wakeUp);
        プリミティブの処理;}
      if(上位のキューに書き込み可能 &&
      書き込むべきプリミティブが存在){          …②
        プリミティブをキューに書き込み;
        condition_signal(nPlus1Synch->wakeUp);}
      下位のキューの処理; ..... }
    mutex_lock(nSynch->lock);                    …③
    while(上位/下位のキューが空 && 書き込み不可)
      condition_wait(nSynch->wakeUp);
    mutex_unlock(nSynch->lock);                  …④
  } .....
```

図3　レイヤのメインプログラム

中の条件変数にシグナルを送る。隣接レイヤが同期待ちでない場合はこのシグナルは無視される。

(4)自身のレイヤの処理が無くなった場合は、同期用の外部変数を用いて、他レイヤとの同期を待つ(図中③から④)。この場合は、そのスレッドが実行されているプロセッサに割り当てられている他のスレッドからのシグナルも待つことができるようスリープするロックを使用する。

3.3 プリミティブおよびPDU用のバッファへのアクセス制御

プリミティブとPDUについては、複数のレイヤのスレッドによりアクセスされるため、以下のような方法で、アクセスのための同期を最小限としている。

3.3.1 プリミティブ用バッファへのアクセス制御

プリミティブ用バッファについては、あるレイヤが、プリミティブを作成し隣接するレイヤに渡した後は、そのバッファにはアクセスしないこととする。これにより、プリミティブ用バッファへのアクセスに対するレイヤのスレッド間で同期が不要となる。

3.3.2 PDU用バッファへのアクセス制御

PDU用バッファとしては、レイヤ間でユーザデータのコピーを避けるために、PDUデータ自身を格納するバッファ本体と、バッファ本体を指し示すバッファ・ディスクリプタから構成されるバッファ構成を採用する[4]。図4に示すように、バッファ・ディスクリプタには、バッファ本体中の作業領域とPDUデータの位置(WorkHeadと

PduHead)、バッファ長とPDUデータ長(BufLenとPduLen)、複数のバッファ本体を連結するためのディスクリプタへのポインタ(NextとPrev)などが含まれる。またバッファ本体には、それを参照しているディスクリプタの数を示すリファレンス・カウント(RefCount)と、リファレンス・カウントにアクセスするためのロック(Lock)が含まれる。

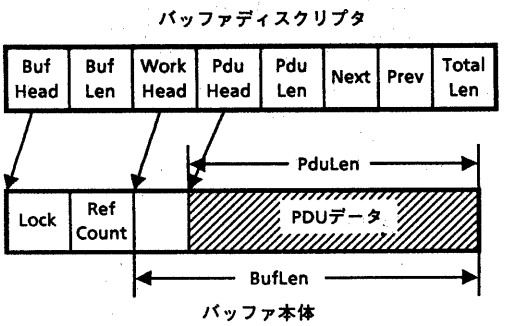

図4　PDUバッファの構成

プロトコル処理においては、送信処理において再送用に保持する場合を除き、1つのレイヤがPDUの処理を終了し、隣接するレイヤに渡した後は、そのレイヤはそのPDU用のバッファについてはアクセスしない。また、1つのレイヤは、PDUの内、自分自身のヘッダのみを操作し、他の領域はユーザデータとして扱い、内容を参照/変更することはない。そこでこれらの性質を考慮し、複数のスレッドがPDUバッファにアクセスする場合の競合を避けるために、以下のような方法を用いることとした。

- あるレイヤで再送用にPDUを保持する場合は、そのPDUに対するディスクリプタを新たに作成する。これにより、すべての場合で、ディスクリプタはスレッド間で共有されることはなくなる。
- バッファ本体はスレッド間で共有させる。バッファ本体に格納されたPDUデータについては、各レイヤのスレッドがアクセスする箇所が定まっているため、ロックによる排他制御は行なわない。一方スレッドが、バッファ本体中のリファレンス・カウントにアクセスするときには、ロック(Lock)をかける。ただしこの場合、リファレンス・カウントの処理時間は非常に小さいと考えられるため、アクセスするスレッドがロックを監視し続けるスピンロックを用いることにする。

本方法を用いたPDUバッファの操作の例を、図4に示す。この図では、レイヤ(N)と(N-1)のスレッドが並列に動作しており、その処理の流れおよびバッファ領域上のPDU用バッファの構成が示されている。レイヤ(N)スレッドは、上位レイヤから、(N+1)-PDUを含むデータ送信要求のプリミティブ((N)-DATA req)を受信し(図中①)、それに(N)レイヤのヘッダを付加する(図中②)。さらに、再送用のPDUを作成し、レイヤ(N-1)スレッドに通知する(図中③)。ただしこの例では、送信済みのPDUと異なる再送用のヘッダを付加するために、(N)レイヤのヘッダを含むバッファ本体を新たに作成している。この後、レイヤ(N)スレッドにおける(N)-DATA req受信と、レイヤ(N-1)スレッドでのヘッダの付加が並列に行なわれる(図中④)。

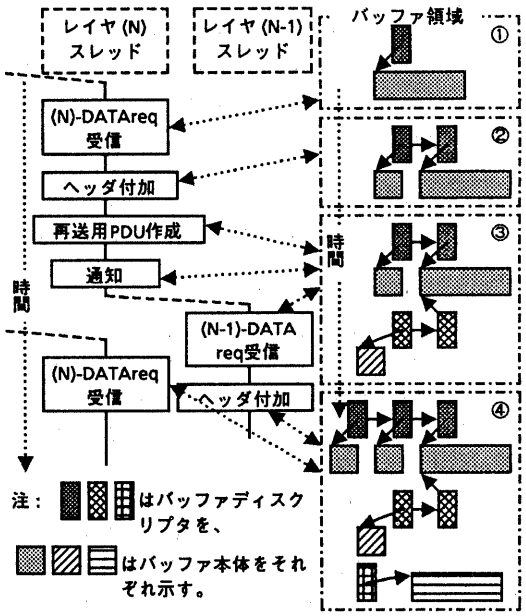

図5　PDUバッファの操作の流れ

3.4 バッファ領域の管理方式

バッファ領域上にプリミティブやPDU用のバッファを確保・解放するために、バッファ領域の制御情報にアクセスするスレッドの間で、同期をとる必要がある。そこで、このオーバヘッドを減らすために、次のようにして、バッファ領域を管理することとした。

3.4.1 ローカルバッファ領域

スレッド内に閉じたローカルバッファ領域を設け、コネクション管理テーブルなどの、1つのレイヤ内でのみ使用される情報のためのバッファは、その領域から確保させる。

3.4.2 グローバルバッファ領域

プリミティブやPDUなど複数のスレッドからアクセスされる情報のためのバッファに対して、次のような方針に基づいて、バッファ領域の管理を実現することとした。

- 各レイヤごとにグローバルバッファ領域を用意し、1つのバッファ領域からプリミティブ/PDU用バッファを確保するスレッドを特定する。一方、バッファの解放については、任意のスレッドで可能とする。
- バッファ領域上には、バッファとして使用可能な空き領域と、確保されたバッファとが混在する。そこで、1つの空き領域内でバッファを確保できる場合は、確保の処理を、解放の処理と同期をとることなく行なう。
- 解放要求については、複数のスレッドが競合して行なう可能性があるため、バッファ領域の制御情報全体にロックをかける。

これらの方針に基づき、以下のようにして、グローバルバッファ領域を管理する(図5参照)。

(1) 1つのスレッドの管理するグローバルバッファ領域においては、空き領域は図5に示すように空き領域管理リストにより管理される。このリストはバッファ領域中の空き領域を表すFACB (Free Area Conrol Block)から構成され、FACBは空き領域の先頭と最後部へのポインタ(TopPtrとBottomPtr)と、他のFACBとの間で双方向リンクを構成するためのポインタを含む。ここで、バッファ領域上の空き領域には、バッファ制御用の情報は含まない。

(2) 確保されたバッファは、先頭に制御情報として、そのバッファのサイズを含む。

(3) バッファの確保は、次の手順で行なう。

① バッファの確保を要求するスレッドは、まず空き領域管理リストにロックをかけることなく、その先頭のFACBのTopPtrとBottomPtrを参照し、参照した時点での空き領域のサイズ(BottomPtr − TopPtr)が、要求されたバッファサイズよりも大きい場合は、その空き領域内にバッファを確保し、バッファの先頭にそのサイズを設定し、HeadPtrをずらして、バッファの先頭ポインタを返す。

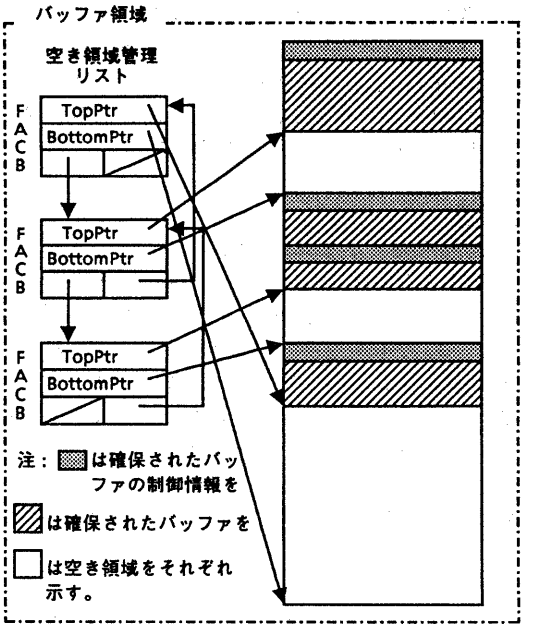

図6 グローバルバッファ領域の構成

② 空き領域のサイズが、要求されたバッファサイズよりも小さい場合は、空き領域管理リスト全体にロックをかけ、現在参照しているFACBを、空き領域管理リストの最後に並べかえるとともに、その次の制御ブロックを参照し、それが管理する空き領域からバッファを確保するよう試みる。空き領域管理リストの各FACBに対して、バッファが確保されるまでこの処理を続ける。

③ 空き領域管理リストの管理する空き領域内にバッファが確保できない場合は、さらに別のバッファ領域を割り当てる。

④ 空き領域管理リストのロックについては、スピンロックを用いる。

このような処理により、先頭のFACBの管理する空き領域内にバッファが確保できる限りは、空き領域管理リストをロックする必要がない。

(4) バッファの解放は、次の手順で行なう。

① 解放を要求するスレッドは、空き領域管理リスト全体にロックをかける(スピンロック)。FACBを先頭からたどり、解放するバッファを挿入すべき位置を求める。

② 解放される領域が、前後のFACBの管理する空き領域と隣接しているかを判断し、この領域を管理するFACBの登録、前後FACBのTopPtrまたはBottomPtrの更新、前後の空き領域の統合の内のいずれかを行なう。

4. 考察

(1) 本稿で検討した並列処理方式により、OSIなどの階層構造を持つ通信プロトコルの並列処理が可能となっている。各レイヤのプログラムは、スレッドにより実行され、個別のプロセッサに割り当てられることにより、キューから通知されたプリミティブを読み出してそれを処理するという動作を、互いに並行に行なうことができる。

(2) 本方式で使用したレイヤ間のキューでは、

- 読み書きを行なうスレッドをそれぞれ1つずつに特定し、
- ポインタを用いたリストではなく、配列を用いてキューを実現し、キューの制御情報の操作する際に、全体にロックをかける必要をなくし、
- 配列の要素にプリミティブがつながれているか否かの情報を持たせる

ことにより、読み書きを行なうスレッドが並行してアクセスできるようにしている。

(3) 本方式では、レイヤ間の同期のためのロックを、以下のように使い分けている。キューが空またはフルの場合の処理のように、他のスレッドの処理(キューの書き込みまたは読み出し)を待つ必要がある場合は、スレッドをスリープさせるロックを使用する。一方、PDU用バッファのリファレンス・カウントや、グローバルバッファ領域の制御情報へのアクセスなどのように、単に他のスレッドの競合を待てばよい場合は、スピンロックを用いる。これにより、複数のスレッドが1つのプロセッサに割り当てられた場合でも、デッドロックなしで動作させることができる。

(4) 通信デバイスとインタフェースするスレッドでは、デバイスからの受信と、条件変数による隣接レイヤからのプリミティブ受信を同時に待つことができないため、通信デバイスからの受信処理を行なうスレッドと、送信処理を行なうスレッドとを個別に設ける必要がある。

(5) グローバルバッファ領域から確保されるプリミティブ/PDU用バッファは、そのプリミティブやPDUの処理が行なわれている間のみ確保され、処理が終わると解放される。すなわち、コネクション管理テーブルなどと比較すると、比較的単時間で解放される。このため、本稿で検討したバッファ領域の管理方式を用いると、多くの場合、先頭の空き領域からロックなしでバッファを確保できると考えられる。

(6) 筆者らは、数個から十数個のプロセッサを持つワークステーションを想定している。この規模のプロセッサ数では、レイヤごとの並列処理が現実的であると考えられる。

5. おわりに

本稿では、共有メモリ型マルチプロセッサ上で、オペレーティングシステムが提供するスレッド機能を用いて、各レイヤのプロトコル処理を並列に実行する方式について述べた。本方式では、レイヤ間のインタフェースに、並行して読み書きが可能なキューを用い、各レイヤのスレッドがキューにプリミティブがあるかぎり、他のスレッドとは並行して処理を続けることを可能としている。また、レイヤ間で共有されるプリミティブやPDU用のバッファへのアクセスや、そのバッファの確保などにおいても、レイヤのスレッド間での競合制御のオーバヘッドを極力減らす方法を採用している。最後に日頃御指導いただくKDD研究所浦野所長、眞家次長に感謝する。

参考文献

[1] : M. Goldberg, G. Neufeld and M. Ito, "A Parallel Approach to OSI Connection-Oriented Protocols," Proc. 3rd IFIP Workshop on Protocols for High-Speed Networks, May 1992.

[2] : M. Bjorkman and P. Gunningberg, "Locking Effects in Multiprocessor Implementations of Protocols," Proc. SIGCOMM '93, Sept. 1993.

[3] : 加藤, 三宅, 鈴木, "共有メモリ型マルチプロセッサを用いたプロトコルの並列処理方式に関する検討," 情処第49回全大, 6C-7, Sept. 1994.

[4] : 加藤, 井戸上, 鈴木, "OSIプロトコル実装のためのユーザデータをコピーしないバッファ制御方式," 情処マルチメディア通信と分散処理研究会, 62-13, Sept. 1993.