システムソフトウェアと
オペレーティング・システム 73－11
（1996．8．27）

ユーザレベルプロトコルのカーネル内実行による大量データ通信の効率的実現

西村 健　猪原 茂和　益田 隆司

東京大学大学院 理学系研究科 情報科学専攻

要旨

近年、高速ネットワーク技術の革新とインターネットの普及にともなって画像や音声などのデータをネットワーク上で通信し加工するアプリケーションが増えつつある。しかし従来のOSではTCP/IPのような特定のネットワークプロトコルのみがカーネル内で実現され、それ以外のプロトコルはカーネル外で実現するしかなかったためアプリケーションレベルのプロトコル層とカーネルレベルのプロトコル層の間でパケットを複製する操作がボトルネックとなっていた。

本研究では、ユーザレベルのプロトコルモジュールをカーネル内にロード可能にし、一連のプロトコル処理をカーネル内で処理することで大量の通信データを効率的に扱う機構を提案する。カーネル内にロードされたモジュールは、通信データに対する主要な処理であるネットワークシステムとファイルシステム間のデータ移動、および内容の加工 (圧縮や展開など) を行う。これらによって、画像や音声データを多くの場合複製操作なしで処理することが可能になる。ユーザコードは不法なアクセスの検査を行うカーネル内のインタプリタ上で実行し、システム全体の安全性を確保する。

High Performance Bulk Data Transfer by Embedding User-level Protocol in the Kernel

Takeshi Nishimura, Shigekazu Inohara, and Takashi Masuda

Department of Information Science, Graduate School of Science,University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113

Abstract

Due to the high-speed network technology and the wide spread of Internet, there are a growing number of multimedia applications that process and exchange large amount of data (audio and video streams). Conventional OSs support a fixed set of protocols in the kernel, and other higher protocols out of kernel. This results in copying of packets between the application-level and the kernel-level protocol layers, which causes large overheads.

This study proposes a mechanism for efficiently transferring and manipulating bulk data, by embedding user-level protocol modules in the kernel, and executing the whole protocol operations in the kernel. The module can move data between the file system and the network system without copying. It also manipulates data in the kernel (e.g. compression, de-compression, or data format transformation). To keep safety, the embedded modules are executed by an interpreter which checks illegal accesses.

1 はじめに

近年のネットワーク技術の進歩とインターネットの普及により、大量のデータを効率的に扱うことを要求するアプリケーションが増えてきた。World Wide Web (WWW) を例にとると、そこで扱われる画像ファイルや音声ファイルは巨大化・精細化し、データ量としては肥大化の一途をたどっている。その大量データを扱う Hypertext Transfer Protocol (HTTP) サーバは、主要な処理であるファイルをそのままネットワークへ流すという処理を効率的に行うことが求められている。また、ネットワークを利用するアプリケーションの中にはデータをそのまま送るのではなく、データを圧縮して送る、または受けとったデータを展開するというようなデータに応じた処理を伴うものも少なくない。Video On Demand や HTTP サーバの Server Side Include (SSI) はその一例である。

しかし従来のオペレーティングシステム (OS) では、このような処理を行おうとするとカーネル空間とユーザ空間との間でのコピー操作が必要になるため効率が悪い。例えばファイルをそのままネットワークに流す際の処理を見ると、ファイルシステムはユーザ空間のあるバッファへデータをコピーし、ネットワークシステムはデータをそのバッファからネットワークシステム内の領域へコピーし、その後実際の通信を行う。このようなコピー操作が必要になるのは、ネットワークシステムからファイルシステム内のデータが見えない、またその逆も同様であるためである。

これに対してファイルシステムとネットワークシステム間のコピー操作を減らして処理を高速化することが考えられるが、単にそれぞれのシステム内にあるデータへのポインタを他方のシステムに渡す、例えばファイルシステムがアプリケーションへファイルシステム内データへのポインタを渡し、アプリケーションがそのポインタをネットワークシステムに渡すことを行うと、アプリケーションから渡されたポインタの信頼性が問題になる。また別の問題として、アプリケーションがそのデータを加工しようとしても、ユーザプログラムはユーザ空間のデータしか参照できないため、ファイルシステムやネットワークシステム内のデータを直接参照することはできないという問題がある。

本研究ではこれらの問題を解決するために、アプリケーションはデータを参照・加工するモジュールを定義し、そのモジュールをカーネル内で実行することとする。これによりアプリケーションはファイルシステムやネットワークシステム内のデータに自由にアクセスできるようになる。ユーザ定義モジュールをインタプリタで実行することにより、そのモジュールがカーネル内のその他のデータにアクセスすることを防いで安全な処理を行うことができる。

この論文の次節からの構成は以下のようになっている。まず2節で過去に行われた研究について触れる。3節ではこの研究で採ったアプローチについて説明し、4節でインターフェースなど詳細を述べる。5節で実際に行った実験を説明しその結果を考察する。

2 関連研究

ユーザ空間とカーネル空間との間でのコピーを減らす方法には、古くからある方法としてファイルをメモリにマッピングするという手法があるが、これを用いてもファイルをそのまま送信しようとするとユーザ空間 (マッピングされたデータ) からカーネル空間 (ネットワークシステム内) へのコピーは避けられない。

このマッピングをネットワークシステムに応用した fbufs[4] という研究もある。この研究は主にネットワークプロトコル層の間のコピーをいかに減らすかということに重点が置かれ、ネットワークから得たデータをそのままファイルに保存しようとするとやはりユーザ空間からカーネル空間へのコピーが必要になる。また、マッピングに要するオーバーヘッドも避けられない。

我々のアプローチと似たような方法として Kevin Fall と Joseph Pasquale が提案した splicing[5] がある。これは2つのデバイスを直接つなげるというものであるため圧縮・展開のように間で処理を行うアプリケーションには応用できない。このようなアプリケーションに対処するには、両システムから同じデータが見えるようにするとともにアプリケーションからもカーネル内にあるそのデータが見えなければならない。

近年の動向として、拡張可能を謳う OS も数多くある。その中に VINO[2] というものがあるが、これは

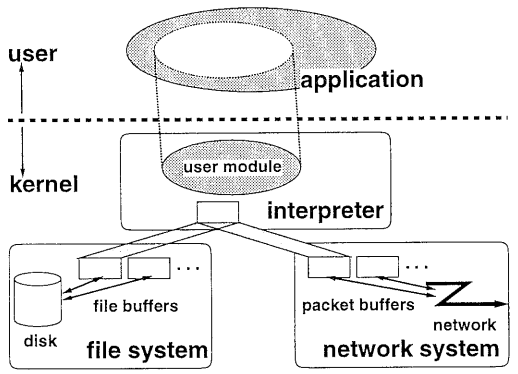

図 1: 本システムの概観

データベースなど従来の OS のポリシーに適合しないアプリケーションに対してアプリケーション独自のポリシーを実現可能にするものであり、我々のように従来の OS になかった機能を実現するものとは異なる。Scout[6] は、ネットワークシステムの最適化などカーネル構築時の拡張に重点が置かれている。また、Aegis[3] のようにハードウェアの機能を剥きだしにして、従来の OS の機能をライブラリとして提供するという手法は、そのライブラリを変更しようとするユーザの負担が大きいと思われる。SPIN[1] は、我々と同じようなアプローチをとっており、Modula-3 という言語で書かれた extension と呼ばれるモジュールをカーネル内で実行することができる。我々は、既存の OS に同様の機能を持たせる方式を模索している。既存の OS を用いることにより、過去のソフトウェア資産が有効利用できると考える。他に、SPIN ではモジュールをコンパイルして実行する点も異なる。我々の研究では実装を簡単にするためモジュールはインタプリタで実行される。

3 本システムの概要

本システムでは前述のような問題を解決するため、ファイルシステムおよびネットワークシステムのデータを相互に変換するメカニズムと、その変換メカニズムを操作するためのユーザ定義モジュールをカーネル内で実行するメカニズムを用意する。

本システムの概観は 図1 のようになっている。従来の OS においては、ファイルシステムとネットワークシステムはそれぞれにファイルバッファ、パケットバッファと呼ばれる固有の領域を持っていた。本システムではユーザ定義モジュールを通して、このファイルバッファとメッセージバッファを変換しさらにそれぞれのシステムへ受渡すことができるようにする。ここでユーザ定義モジュールとは、必要ならばバッファの内容の加工 (圧縮や展開、データフォーマットの変換など) を行い、ネットワークシステムまたはファイルシステムへバッファを渡すプログラムのことである。

例として、ファイルの内容をそのままネットワークへ流す時のカーネル内の動作を見る。

1. ファイルシステムはファイルをディスクから読み込みファイルバッファに保存し、そのファイルバッファをユーザ定義モジュールに渡す。

2. ユーザ定義モジュールは受けとったファイルバッファをパケットバッファに変換してネットワークシステムに渡す。

3. ネットワークシステムは受け取ったパケットバッファをネットワークに流す。

ここで、ユーザ定義モジュールが実行される環境に制約が必要である。それは、(1) カーネル内の任意のデータにアクセスできてはいけない、(2) (バッファをネットワークシステムに渡すというような) カーネル内で定義されたルーチンを安全に起動できなければならない、の 2 点である。そこで我々は、ユーザ定義モジュールを記述する言語として Java™ [1]を用い、そのモジュールの実行をインタプリタで行うこととする。Java では、型情報を含んだバイトコードを用いているためユーザ定義モジュールのバイトコードが違法なキャストを行わないことを検査できるほか、そのバイトコードをインタプリタで実行することにより配列の添字チェックのような動的なチェックも容易にできる。

[1]Java および Java を元にした商標は米国およびその他の国における Sun Microsystems, Inc. の商標または登録商標です

4 本システムの詳細

4.1 インターフェース

本システムのインターフェースには、ユーザ空間にあるプログラム (メインプログラムと呼ぶ) からの処理を行うためのインターフェースとユーザ定義モジュールの構造やモジュールからの可能な操作を定めるインターフェースの2種類がある。

メインプログラムからのインターフェースには bindjavaclass という新システムコールがある。このシステムコールは、既存のシステムコールである open などにより得られたファイルディスクリプタにユーザ定義モジュールを結び付ける。このシステムコールによってモジュールに制御が移され、ファイルディスクリプタから得たデータを元にデータの加工・転送などが行われる。

ユーザ定義モジュールは Java のクラスとして表され、ユーザはそのクラスの init, processData というメソッドに初期化およびデータの処理を記述する。ファイルシステム内のファイルバッファやネットワークシステム内のパケットバッファは KernelBuffer と呼ばれる同一のクラスのインスタンスとして表される。KernelBuffer には writeTo メソッドがあり、引数として渡されるファイルディスクリプタへバッファの内容を書き出すという処理を行う。

加えて、メインプログラムとユーザ定義モジュールとの間のデータの受渡しを行うためにプロセス毎にプロパティというものが用意されている。これはキーと値の組のリストであり、キーも値も文字列で表される。

4.2 処理の流れ

bindjavaclass によって行われるカーネル内の処理は、(1) まず結び付けられたクラスからオブジェクト (処理オブジェクト) を生成し init でそれを初期化し、(2) 次に bindjavaclass の引数として与えられるファイルディスクリプタからデータを1バッファ分 (ファイルの場合は 8192 バイト) 読み込み KernelBuffer のオブジェクトとして処理オブジェクトの processData メソッドに渡して1バッファ分の処理を行う。(2) の処理を読み込むデータがなくなるまで繰り返す。

```
infd = ...
sockfd = ...
...
{ /* int to string */
    char num[10];
    sprintf(num, "%d", sockfd);
    putjavaproperty("sockfd", num);
}
bindjavaclass(infd,
              "httpd.SendFileWithHeader");
```

図 2: メインプログラム例

```
public class SendFileWithHeader {
  public static byte[] RES_HEADER = ...
  int sockfd;
  public void init(int fd) {
    sockfd = KernelSystem.
             getJavaProperty_int("sockfd");
    KernelBuffer header = new KernelBuffer();
    header.data = RES_HEADER;
    header.writeTo(sockfd);
  }
  public void processData(KernelBuffer buf) {
    buf.writeTo(sockfd);
  }
}
```

図 3: ユーザ定義モジュール例

本システムを用いた例を図 2,3に示す。 この例は HTTP サーバの処理の一部を抜き出したもので、infd で指定されたファイルの内容を sockfd で指定されたソケットへヘッダを付けて送るというものである。メインプログラムの方では putjavaproperty によって sockfd の値を文字列形式でプロパティに設定し、bindjavaclass で実際の転送を指示している。ユーザ定義モジュールでは、init で sockfd の値をプロパティから得てヘッダを出力している。processData では、単にファイルシステムから得たバッファ buf を writeTo メソッドでソケットに送っている。

4.3 実装上の問題

ファイルバッファとパケットバッファを相互変換する上での問題点が1つある。それは、それぞれのバッファの管理方式の違いである。ファイルバッファがfree list で空きバッファを管理しているのに対してメッセージバッファは reference count を用いている。さらには、パケットバッファの最大サイズはファイルバッファのそれより小さいため、ファイルバッファをパケットバッファに変換するにはファイルバッファのデータを分割しなければならない。つまり、1つのファイルバッファが複数のパケットバッファと対応することになる。ファイルバッファを疑似パケットバッファに変換した場合、複数のパケットバッファの中の1つの reference count が 0 になったとしても、ファイルバッファを free list につなぐわけにはいかない。

これを解決するためにファイルバッファ自身に reference count の総和を保持しておき、それが0になって初めてファイルバッファを解放するという機構を用いる。これにより、既存のカーネルの構造を維持しつつファイルバッファとパケットバッファとの間の変換を可能にしている。

5 実験

この節では本システムを使った予備実験の結果を示す。最初の実験は、比較的大きなファイルをファイルシステムから読み、ネットワークに送出する速度を測るもので、次の実験では小さいファイルの場合のシステムの挙動をみる。

実装は NetBSD/sparc 1.1 上で行い、本システムのうち Java インタプリタの部分とファイルバッファをインタプリタに渡す部分、およびファイルバッファをパケットバッファに変換して送信する部分を実装した。Java インタプリタの実装には Kaffe という Java インタプリタのソースを利用した。

実験を行なったマシンは 25 MHz の SPARC チップが載った SPARCstation ELC で、16 M バイトのメモリを持つ。ネットワークは 10Mbps ethernet で、SPARCstation 20/71(メモリは 160 M バイト) と接続している。

図4は、ファイルをそのままネットワークへ流す際

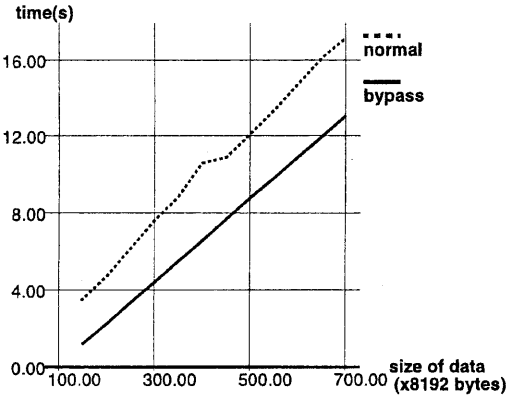

図 4: ファイルからネットワークへの転送

	time (μs)
normal	2368
bypass	3211

表 1: HTTP サーバの転送時間

に要する時間を、普通に C で書いたもの (normal) と本システムを使ってカーネル内で転送を行うようにしたもの (bypass) で比較した結果である。normal では、ユーザ空間に 8192 バイトのバッファを用意し、そのバッファを介して転送を行っている。計測した時間は、最初の (ファイルに対する)`read` システムコール の手前から最後の (ソケットに対する)`write` システムコール が終了するまでの実時間である。bypass の時間にはプロパティ設定などに要する時間も含まれている。ファイルサイズを 150 × 8192 バイトから 700 × 8192 バイトに変化させて時間を計測した。この結果は、本システムを用いたものの方が処理時間が改善されていることを示している。

次に、HTTP サーバの処理としてヘッダ付きでファイルを送る時間を計測した。プログラムは図2,3にあるものを用いている。送られるファイルは 2322 バイトの HTML ファイルで、ヘッダは status-line のみ (常に "`HTTP/1.0 200 OK`") としている。ここでは、ファイルバッファ中に当該データが存在する、つまりディスクアクセスを伴わない場合の時間を見ている。結果を表1に示す。さらに、それぞれの処理を細かく分割して処理時間を計測したのが表2である。

2番目の実験の結果は bypass の方が遅いことを示している。これはサイズの小さいデータを送る時には

normal	time (μs)
ヘッダ転送時間	381
ファイル read& write	1987
上記のうちコピー時間	846
bypass	
プロパティ設定	313
処理オブジェクト生成	440
init メソッドの処理	1108
ファイル read	692
processData メソッドの処理	658

表 2: 転送時間の内訳

ユーザ空間とカーネル空間との間のコピーに要する時間よりもインタプリタの起動などに要する時間の方が長いためであると考えられる。表2を見ると、normalの方では処理自体が存在しない、プロパティ設定と処理オブジェクト生成に時間がかかっていることが分かる。プロパティ設定の部分は値を引数として渡すことによりなくすことができる。その場合、initメソッドの対応する部分も減らすことができる。

別の実験で、Java インタプリタの起動と終了合わせて 169 μs かかっていることが分かっている。この部分での処理はメソッドの検索と局所変数などのための領域確保である。処理オブジェクトや KernelBuffer 生成の際にもコンストラクタ (実際は何もしない) を実行するためにインタプリタが起動されるので、先の HTTP サーバの実験では合計5回インタプリタが起動されていることになる。initメソッドで行っている処理をコンストラクタで行うようにするなど、何らかの対策が必要である。

6 まとめ

カーネルの中でユーザ定義モジュールを用いてファイルバッファとメッセージバッファを統一的に扱うことにより、ユーザ空間とカーネル空間との間のコピー操作を減らす手法を提案した。そしてユーザ定義モジュールの実行には既存の Java インタプリタを流用して実装を行った。

サイズの大きいファイルを転送する場合は改善が見られるが、サイズの小さいファイルを転送する場合は、コピー時間より Java の処理のオーバーヘッドのほうが大きいため、処理時間を改善することができない。これを改善する方法として、アプリケーションとカーネルとのインターフェースの見直しやインタプリタの最適化を行うことが考えられる。

7 謝辞

この研究に対して忌憚のない意見を下さった益田研究室の方々、特に実装に関して助言をいただいた中村隆幸氏に感謝の言葉を捧げる。

参考文献

[1] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. Technical report, Department of Compuer Science and Engineering, Univ. of Washington, March 1995.

[2] Margo Seltzer Christopher Small. VINO: An integrated platform for operating system and database research. Technical report, Harvard University, 1994.

[3] and James O'Toole Jr. Dawson R. Engler, M. Frans Kaashoek. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[4] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, 1993.

[5] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *USENIX Technical Conference Proceedings*, pages 327–333, San Diego, CA, Winter 1993. USENIX.

[6] Allan B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Petersonand, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, June 1994.