システムソフトウェアと
オペレーティング・システム 79－11
（1998. 8. 6）

# ユーザレベル軽量プロセスライブラリにおける効率の良いI/O処理方式

安倍広多†, 松浦敏雄†, 安本慶一††, 東野輝夫†††

† 大阪市立大学 学術情報総合センター
†† 滋賀大学 経済学部
††† 大阪大学大学院 基礎工学研究科

**あらまし**

ユーザレベルマルチスレッド機構は, コンテキストスイッチのオーバヘッドが小さいために, カーネルレベルスレッドよりも効率が良いことが知られている. しかし, 一つのスレッドがプロセス全体をブロックするようなシステムコール (ファイルI/O等) を発行すると, それが完了するまで他のスレッドの実行ができないという制約があった. 本研究では, この制約を回避する方法を提案し, 著者らが開発したユーザレベルマルチスレッドライブラリ (PTL) に実装した. この方法を用いると, スレッドがファイルI/O等のシステムコールを発行した場合でも, 当該スレッドのみがブロックされ, 他のスレッドは実行を継続できる. また, 実験により, この方法の有効性を示す.

**キーワード** マルチスレッド, ユーザレベルスレッドライブラリ, UNIX, ノンブロックI/O

# An efficient method to perform I/O operation on a user-level thread library

Kota ABE†, Toshio MATSUURA†, Keiichi YASUMOTO††, Teruo HIGASHINO†††

† Media Center, Osaka City University
†† Faculty of Economics, Shiga University
††† Graduate School of Engineering Science, Osaka University

**Abstract**

A user-level thread is known to be more efficient than a kernel-level one because of its small context switch overhead. However, user-level thread has a restriction that if one thread issues a system call that forces the calling process to be entirely blocked (such as disk I/O operation), no other thread is runnable until the system call has completed.

In this paper, we have have shown a method to overcome this restriction efficiently. With this method, if one thread requests a system call such as disk I/O operation, only the thread that issues the request is blocked and other threads can continue execution.

We have implemented this method on our user-level thread library (PTL) and have confirmed by some experiments that this method's availability and efficiency.

**Keywords** Multi thread, User-Level Thread Library, UNIX, Non-blocking I/O

# 1 はじめに

近年, 複数のスレッドを用いたプログラミング (マルチスレッドプログラミング) が広く用いられるようになってきている. アプリケーションを複数のプロセスで構成するよりも, マルチスレッドで構成した方がプロセス間通信やコンテキストスイッチのオーバヘッドを削減できるため, 効率が良いことが知られている[3]. スレッドの実装には, 大きく分けて, カーネルレベルでサポートするカーネルレベルスレッドと, ユーザレベルで実装するユーザレベルスレッドの2つがある.

著者らは UNIX 上で動作する移植性の良いユーザレベルスレッドライブラリ PTL[1][2]に関して研究, 開発してきた. PTL は, 1) BSD UNIX ならば CPU に依存しない, 2) POSIX 1003.1c (Pthread[10]) 互換の API を備える, 3) プリエンプティブスケジューリングをサポートする, 等を特徴としている.

一般的にユーザレベルスレッドは移植性, 効率性に優れているが, カーネルがスレッドに関知していないため, スレッドがシステムコールを発行すると, システムコールが完了するまでプロセス全体がブロックされてしまい, 他のスレッドを実行できない. これは, システムコールの実行に長時間を要する場合に問題となる.

特に長時間ブロックするシステムコールとして, ファイルに対する入出力のシステムコールがある. ファイル入出力にかかるほとんどの時間はディスク装置の動作の待ち時間 (NFS の場合はファイルサーバからの転送待ち時間) であるが, この間, 他のスレッドを実行できない. 例えばマルチメディア処理では, メディアデータをファイルから読み込みながらデコード処理や表示処理を行うことが望ましいが, ユーザレベルスレッドでは入出力中の待ち時間を有効に利用できなかった.

UNIX 上のユーザレベルスレッドは幾つか開発されている[4][5][6][7][8]. UNIX 上のユーザレベルスレッドでは, ネットワークや端末入出力に対しては UNIX のノンブロック I/O モードを使用してプロセス全体がブロックしないようにするのが一般的な方法である. しかし, ノンブロック I/O モードであってもファイル入出力に対しては効果がなく, 入出力が完了するまでプロセス全体がブロックしてしまう. プロセス全体をブロックさせずにファイル入出力を行えるユーザレベルスレッドライブラリは見当たらない.

本研究では, この問題に対する1つの解決策を示し, PTL をそれに基づいて改良した (これを PTL-N と呼ぶ.). PTL-N を利用する際に, 個々のアプリケーションプログラムを変更する必要はなく (透過性), また, PTL の持っている移植性を損なうことがないようにしている. PTL-N ではファイル I/O を専門に行うプロセスを別途設けることによって, プロセス全体をブロックさせることなくファイル入出力を実行できる. この際, プロセスを分離したことによるプロセス間通信のオーバヘッドが新たに発生するが, これを削減する方法も考案した. 実験の結果, ファイル I/O におけるプロセス全体のブロックが回避されたことによって, I/O の完了待ち時間の90%程度を別の処理に使用できることを確認した. また, プロセス間通信のオーバヘッド削減の効果として, この方法によるファイル入出力の性能は, 通常の方法によるファイル入出力の性能と大きな差はないことを確認した.

# 2 I/O 処理方式と実現上の問題

一般の UNIX では, カーネルに手を入れない限り, ファイル入出力を行う間, プロセス全体はブロックされてしまう. このため, 我々はファイル I/O を専用のプロセス (I/O サーバと呼ぶ) で行うことにした. 本来のアプリケーションが動作するプロセスをメインプロセスと呼ぶことにする.

I/O サーバはメインプロセスとプロセス間通信を行ってファイル I/O 要求を受信し, ファイル入出力のシステムコール (read, write) を実行する. 完了したら結果を送信する (図 1参照).

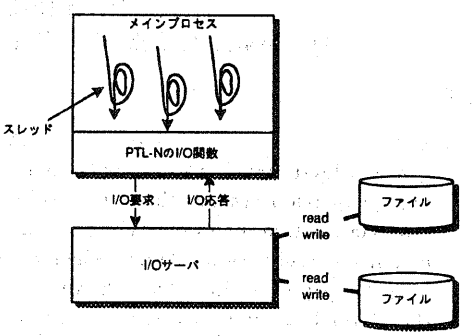


図 1: PTL-N の構成

この処理方式では, プロセスが分離している

ため，効率良くI/Oを行うには次のオーバヘッドを削減しなければならない．

- I/Oデータ転送のオーバヘッド

  図1で，ファイルから読み込んだデータ，あるいはファイルへ書き込むデータ(以下I/Oデータと呼ぶ)をプロセス間でどのように受渡しするかが問題になる．一回のファイル入出力で数キロバイト程度を読み書きすることは普通であり，大きいときはメガバイト単位になる可能性もある．これをなるべく高速に受渡しする必要がある．

- I/O完了通知のオーバヘッド

  I/Oサーバ側でI/Oが完了したら，その旨をメインプロセスに伝える必要がある．これにはプロセス間でのやり取りが必要なため，I/Oを頻繁に行うと性能が低下するおそれがある．このため，I/O完了を通知する際のオーバヘッドを極力削減する必要がある．

# 3 PTL-Nの高速化の方法

## 3.1 I/Oデータ転送の高速化

UNIXで最も速いプロセス間通信は共有メモリ機構を使用する方法であるが，これを利用すると，I/Oデータを共有メモリセグメントとメインプロセスのメモリ空間の間でコピーする必要がある．(例えばファイルからの読み込みの場合，I/Oプロセスがreadシステムコールで共有メモリ上にデータを読み込み，メインプロセスが共有メモリからデータをコピーする．)サイズが大きい場合，これはかなりのオーバヘッドとなる．

コピーを避けるためには，I/Oサーバが直接メインプロセスのメモリ空間にアクセスできればよいが，UNIXではプロセスのアドレス空間は独立しているため，通常の方法では不可能である．

我々は，UNIXのmmap機構(通常のファイルをプロセスのアドレス空間にマップする機構)を利用することで，この制限を回避し，通常のUNIXの下で，アドレス空間をプロセス間で共有する方法を見出した(詳細は4.1)．最近のほとんどのUNIXでは，mmapシステムコールをサポートしているため，この方法は移植性が高いと考えられる．

この方法を使って，I/Oサーバプロセスとメインプロセスのアドレス空間を共有させることができる．これにより，I/Oサーバは，I/O要求があった場合に，指定されたバッファのアドレスでread, writeシステムコールを発行するだけでよい．UNIXカーネルはI/Oサーバのアドレス空間にデータを書き込む(readの場合)が，それはメインプロセスのアドレス空間に自動的に反映されることになる(図2)．

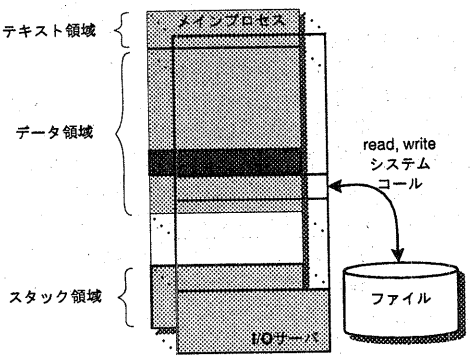


図2: アドレス空間の共有

## 3.2 I/O完了通知の高速化

メインプロセスでは，I/Oサーバに I/Oを要求した後も，別のスレッドが動作している．メインプロセス側では，I/Oを要求したスレッドの実行を再開するために，I/Oが完了したことを知る必要がある．これには，以下のような方法が考えられる．

1. (共有されている)データ領域を通じて通知．
2. UNIXドメインソケット等を使って通知．
3. シグナルでメインプロセスに割り込む．

方法1は，データ領域を定期的にチェックする必要がある．UNIXのインターバルタイマーは，一般に10msecが最小の割り込み間隔であるため，1スレッド当たり1秒間に高々100回のI/Oしか行えない．方法2は，ソケットにデータが到着していないかどうかをチェックする必要がある．これは，定期的にselectシステムコールでチェックするか，非同期I/Oモード(データの到着と同時にシグナルで割り込む)を使う方法しかないが，前者の方法は方法1と同じ問題があり，後者の方法は方法3と同じことである．

これに対し, 方法 3 では定期的なチェックが必要ないため, これらの方法の中では一番効率が良い. このため, PTL-N では方法 3 を用いることにした.

しかし, シグナル割り込みは比較的高価な処理であるため, 頻繁に I/O を行う場合には, そのコストは無視できない. このため, 必要がない場合にはシグナルを送信しないようにして, 割り込むシグナルの数を削減する.

メインプロセスが I/O 要求を送信することによって, ブロックしていた I/O サーバが起きることになるため, プロセス間のコンテキストスイッチが発生する可能性がある. キャッシュにヒットする等の理由で I/O サーバでの I/O 処理が短時間で終了する場合, メインプロセスからみると, I/O 要求を送信 (システムコール) した後, その次のプログラムコードを実行した時点で既に I/O が終わっている可能性がある (図 3). この場合は, メインプロセスに割り込む必要はないため, シグナルの送信を省略できる (詳細は 4.2).

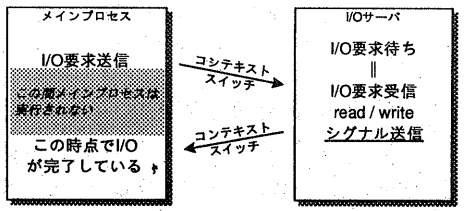


図 3: プロセス間の制御フロー

# 4 実装

ここでは, PTL-N の処理の流れに沿って, 実装方法について述べる.

メインプロセスでは PTL-N の初期化処理中に fork し, I/O サーバを生成する. I/O 要求を送信するため, メインプロセスと I/O サーバとは UNIX ドメインソケットで接続する. I/O サーバは, (*exec* せずに) PTL-N 中に埋め込まれたコードを実行する[1].

I/O サーバは recvmsg システムコールによって I/O 要求を待つ. 実際は, メモリ空間を共有しているため, I/O 要求メッセージ自体はメモリ上に置き, メッセージのアドレスだけをメインプロセスから I/O サーバに送るようにしている.

メインプロセスから I/O サーバへは 4 種類 (open, close, read, write) の I/O 要求を送信する. プロセスをブロックさせないファイル操作 (lseek 等) は, メインプロセスで直接実行する. このため, メインプロセスと I/O サーバはファイル記述子を共有する[2].

ファイルのオープン処理では, open システムコールはメインプロセス側で発行し, 得られたファイル記述子を open 要求と一緒にデスクリプタパッシング (UNIX ドメインソケットを通じて, プロセスのファイル記述子を別のプロセスに送る方法) によって I/O プロセスにファイル記述子を送る[3]. close 要求ではファイル記述子を close する.

メインプロセスと I/O プロセスはアドレス空間を共有しているため, read, write 要求では, 単にシステムコールを発行するだけでよい.

I/O サーバは I/O 要求を処理した後, I/O 完了通知としてシグナル (SIGUSR1) を送信する.

以下, 4.1でプロセス間のアドレス空間を共有する方法, 4.2で I/O 完了通知シグナルの削減方法, 4.3では, ファイル記述子の管理方法を述べる. 4.4では, プロセスを分離したことによって発生する, プロセスの優先度の問題について述べる.

## 4.1 プロセス間のアドレス空間の共有方法

UNIX の mmap 機構を使用することによって, 次のようにして, プロセス間でデータ領域を共有することができる.

- 一時ファイルを作成し, プロセスのデータ領域全てをファイルにコピーする.
- 一時ファイルを mmap によってデータ領域の先頭にマップする. mmap では, マップした領域に書き込んだ場合, 書き込んだ結果をプロセス固有とする (MAP_PRIVATE) か, 共有するか (MAP_SHARED) を選択で

[1] exec してしまうと, mmap システムコールによってマップした領域が解除されるため, アドレス空間を共有できなくなる.

[2] ブロックする可能性があるシステムコールは他にもある (readv, mkdir, unlink 等) が, まだ I/O サーバで実行するように実装していない.

[3] ファイルの open システムコールは時間がかかる場合があり, その間プロセスがブロックされるため, open システムコールは I/O サーバで実行したほうが良いが, 現在はそうしていない. 将来は, I/O サーバで open し, メインプロセスにデスクリプタパッシングでファイル記述子を送るように変更する予定

きるが，ここではプロセス間で共有するために MAP_SHARED を選択する．

- プロセスが fork する際，通常のデータ領域は共有されないが，mmap された領域は子プロセスに引き継がれる．このため，以後 fork すると，fork したプロセス間でデータ領域を共有することができる．

このようにしてプロセス間でデータ領域を共有できるが，スタック領域はどう対応するかという問題と，malloc 等によってデータ領域が拡大した場合にどう対処するかが問題となる．以下これらの問題について述べる．

#### 4.1.1 スタック領域の非共有

上記の方法でデータ領域は共有できるが，PTL-N では (UNIX の) スタック領域は共有しない．それは，スタックが共有されていると，プロセスを fork することが困難になるためである．スタックが共有されている場合，fork した後に正常に処理を行うためには 2 つのプロセスで同時にスタックにアクセスしないようにする必要がある．これは (少なくとも) 移植性のある方法では達成できないと思われる．このため，プロセスを fork するために，共有されていない領域が残っているほうが都合が良い．

しかし，スタック上に確保される変数の配列上に read するようなユーザのプログラムが存在する．透過性を保つため，ユーザのコードを実行するスレッドのスタックは全てデータ領域中に確保するようにした．このため，UNIX のスタック領域が共有されなくても問題はない．

#### 4.1.2 malloc の問題

malloc 等の関数を用いることによって，プロセスのデータ領域は拡大する．(実際は sbrk システムコールでデータ領域が拡大される) これによって，最初に一時ファイルをマップした領域を越えてしまうと，そこがプロセス間で共有されないことになる．malloc した領域にファイルデータを読み込むようなプログラムのために，これに対処する必要がある．

PTL-N では，malloc 関数内部で，新たなデータ領域を確保する (sbrk の呼び出し) 箇所を修正し，新たに確保した領域も I/O サーバと共有するようにした[4]．malloc によってマップした領域を越えるデータ領域が必要な場合には以下の処理を行う．

- 新たな一時ファイルを作成し，マップする
- 一時ファイルのファイル記述子と，マップするアドレスを I/O サーバに伝える
- I/O サーバでも同様にマップする

### 4.2 I/O 完了通知シグナルの削減

3.2で述べたように，I/O 完了通知で用いるシグナルの数を削減するため，以下のような最適化を行った．

- メインプロセスは，I/O 要求を送信した後，I/O が完了しているかチェックする．完了していれば，次に進んでよい．完了していなければスレッドを I/O キューに繋いで，ブロックさせる．
- I/O サーバは，I/O を完了した時点で，メインプロセスが上記のチェックを終えていれば，メインプロセスのスレッドはブロックしているので，シグナルを送る．まだチェックを終えていなければ，シグナルを送る必要はない．

これを行う疑似コードを図 4に示す．図中の lock, unlock は，プロセス間のレース条件を排除するために必要なロックである．今回は，2 つのプロセス間での相互排除であるため，ロックにはシステムコールが不要な Peterson のアルゴリズム[11]を用いてユーザレベルで実現した．

### 4.3 ファイル記述子の管理

PTL-N では，ファイルの open, close 時に，デスクリプタパッシングを用いて，メインプロセスと I/O サーバの間でファイル記述子を共有している．

これに対し，I/O サーバだけがファイルを open し，メインプロセスではファイル記述子を保持しない方法もあるが，この場合ファイル記述子を扱う全てのシステムコール (lseek, fcntl, stat 等) を I/O サーバ側で実行する必要があり，オーバヘッドがかかるため，採用しない．

### 4.4 プロセスの優先度

メインプロセスで，I/O サーバに I/O を要求し，応答を待つ間に，他のスレッドが実行を続ける場合を考える．

[4]GNU malloc ライブラリを修正した

```
// (main process side) send I/O request
send_request(request)
{
  request->status = YET;
  send I/O request to I/O server;
  lock;
  if (request->status != COMPLETE) {
    request->status = WAIT;
    enqueue to I/O queue;
    unlock;
    block until signal;
  } else {
    unlock;
  }
}
// (I/O server side) process I/O request
process_io_request(request)
{
  int sendsig = 0;
  perform requested I/O;
  lock;
  if (request->status != YET) sendsig = 1;
  request->status = COMPLETE;
  unlock;
  if (sendsig) {send signal to main process}
}
```

図 4: I/O 完了通知の削減方法

I/O サーバでは, 発行したシステムコール (read, write) が完了したときに, シグナルを送る必要がある. しかし, メインプロセスでスレッドが継続して実行していると, I/O サーバが発行した read, write が完了しても, I/O サーバになかなか CPU が回って来ないため, シグナルの送信が遅れる (そのため, メインプロセスが I/O の完了に気付くのも遅れる) という状況が発生する.

この状況は, I/O サーバのプロセスの優先度をメインプロセスよりも高くすることによって解決できる.

# 5 評価

実際に PTL-N を実装し, 評価を行った.

## 5.1 移植性と透過性について

現在の所, PTL-N は, SunOS4.1.4, FreeBSD2.2.6, Solaris2.5.1(x86) 上での動作を確認している. また, 4.1の方法によるアドレス空間の共有は, これらに加えて, NEWS-OS6.1.2 でも達成できることを確認した (PTL-N 自体はまだ移植されていない). HP-UX10.10 ではアドレス空間の共有はできなかった. これは, HP-UX の mmap がデータ領域上にファイルをマップできないことが原因と思われる.

以上のことから, アドレス空間の共有法は一部で制約があるが, 大部分の UNIX で使用できるということを確認した.

また, PTL-N での改良は, 従来のシステムコールのセマンティクスを維持したまま行っているため, アプリケーションプログラムを変更する必要はない.

## 5.2 I/O 性能

PTL-N の I/O 速度の評価を行うため, 10M バイトのファイルを read, write するときの所要時間を以下の条件で測定した.

- PTL-N を用いた場合と, 用いない場合
- ファイルが UFS(ローカルディスク) 上にある場合と, NFS(ネットワークファイルシステム) 上にある場合
- キャッシュにヒットする場合と, ヒットしない場合 (read の場合)

1 回あたりの I/O のサイズは, 8K バイト (典型的な標準 I/O ライブラリ (stdio) がファイル I/O で用いる大きさ) から 256K バイトまで変化させた.

PTL-N での測定では, I/O 待ちの間に他のスレッドで使用できる CPU 時間 (有効 CPU 時間と呼ぶ) を測定するため, I/O 実行中に, 別の低優先度のスレッドで無限ループを実行させ, それに費した CPU 時間も測定する. 有効 CPU 時間が長い程, I/O 待ち時間を有効に利用できたことになる. このとき, プロセスの優先度は, メインプロセスの優先度を 5 下げて, I/O サーバを優先するようにして実験を行った. これを行わないと, 4.4で述べた原因により I/O にかかる時間が長くなってしまう. なお, このスレッドを動作させない場合のほうが, I/O 処理時間は短縮されることを確認している.

測定には, SPARC Station 20 (SunOS4.1.4, メモリ 64MB) を用いた. NFS サーバには, IBM-PC 互換機 (Solaris2.5.1, MMX-Pentium166MHz, メモリ 64MB) を 10Base/T 経由で使用した.

### 5.2.1 NFS write の場合

NFS write での結果を図 5に示す. 横軸は 1 回の I/O サイズ, 縦軸は所要時間である.

8Kバイト単位のI/Oでは, PTL-NではPTL-Nを使わない場合に比べて20%程度のオーバヘッドがあるが, 1回当たりのI/Oサイズが大きくなるにつれてほとんど差はなくなっている. これはI/Oの頻度が減少するためである. また, NFS writeの待ち時間の少なくとも90%程度は別のスレッドで有効に利用できている.

キャッシュヒットしない NFS read と UFS write もほぼ同様の傾向となった.

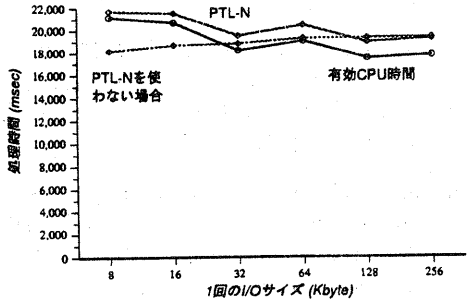


図 5: I/O の速度 (NFS write)

### 5.2.2 UFS read の場合

次に, UFS read での結果を図6(キャッシュにヒットする場合), 図7(ミスする場合) に示す.

キャッシュにヒットする場合, read システムコールはキャッシュからコピーするだけの処理となる, このため, 有効 CPU 時間はほとんど残らないことがわかる. キャッシュヒットする NFS read も同様の傾向となった.

ヒットしない場合は, read がすぐには完了しないため, NFS write 同様, 他のスレッドで時間を有効に使用できている. UFS read は NFS write に比べ短時間で完了するため, PTL-N を使用した場合, I/O の処理時間は 2〜2.5 倍程度かかっているが, 有効CPU 時間から, I/O 時間のほとんどは有効に使用できているといえる.

## 5.3 高速化の効果

プロセス間でのアドレス空間の共有と, I/O 完了通知の削減の効果を測定するために, 以下の 2 つの場合について I/O 速度を測定した.

- アドレス空間を共有せずに, 共有メモリ機構を使った場合を想定し, I/O のたびにメモリブロックのコピーを行う

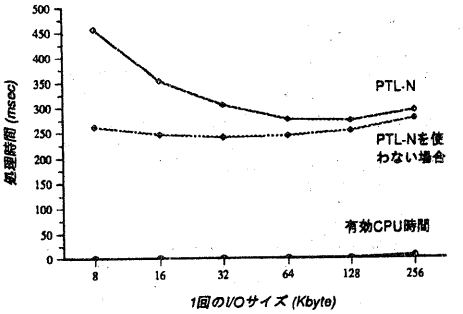


図 6: I/O の速度 (UFS read, キャッシュヒット時)

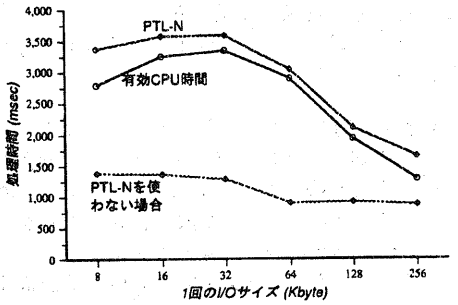


図 7: I/O の速度 (UFS read, キャッシュミス時)

- I/O 完了のたびに必ずシグナルで割り込む

UFS read での結果を図 8に示す. メモリのコピー削減の効果はどのI/Oサイズでも効果が高いことがわかる. シグナル数の削減は, I/Oサイズが小さい場合に効果が高い. これはI/Oサイズが小さい程シグナルの数が多いためである. NFS の場合も同様に効果があるが, I/O にかかる時間が長いため, これほど顕著な差は現れない.

UFS read でキャッシュヒットする場合, PTL-N ではシグナル割り込みは 1 回も発生しなかった. NFS write 等, I/O がすぐには完了しない場合, ほとんどの場合, シグナルで割り込まれることを確認している.

# 6 おわりに

本研究では, ユーザレベルスレッドでファイル I/O を行うことによるプロセス全体のブロックを回避するための効率が良い方法を考案し, その実装を行った. この方法を用いることで, 特

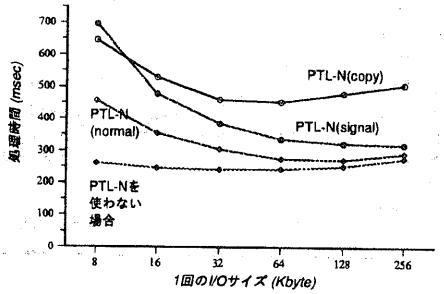


図 8: PTL-N の高速化技法の効果

に NFS での read, write や UFS write 等の低速なファイル I/O では, I/O 時間を有効に使用できることを実験で確認した.

今後の課題としては, 以下のような点が挙げられる.

- 現在は I/O サーバはメインプロセス 1 つにつき 1 つだけ生成しているため, 複数の I/O を同時に実行することができない. 複数の I/O サーバを生成し, 同時に複数の I/O システムコールを発行できるようにすると, アプリケーションの I/O 性能が向上すると思われる.
- メインプロセスと I/O サーバで適切な優先度を設定し, 要求する I/O の種類に応じて, 動的に優先度を変更して, 適切な I/O を行うようにすることも興味深い.
- 本研究で考案した, 複数のプロセス間でメモリを共有する技法を使って, PTL をマルチプロセッサ対応に改良することができる. マルチプロセッサマシンでは複数のプロセスを同時に実行できるため, プロセスを仮想 CPU として使うことによって, 複数のスレッドを別々のプロセスに割り当てて実行すればよい. 同様の研究は [8] や [9] でもなされているが, 前者のプロセス間のメモリ空間を共有する方法は OS に依存する方法を用いており, 後者はプロセス間で全てのデータ領域を共有しているわけではないという問題点がある. PTL-N での方法を用いるとこれらの問題を解決できる.

## 参考文献

[1] 安倍広多, 松浦敏雄, 谷口健一: BSD UNIX 上での移植性に優れた軽量プロセス機構の実現, 情報処理学会論文誌, Vol.36, No.2, pp.296–303, (1995-2).

[2] 安倍広多, 他: UNIX 上で周期スレッドを実現するユーザレベルスレッドライブラリの実現法, 信学技報 CPSY-97-24, pp.49–54 (1997-6).

[3] 追川修一, 徳田英幸: ユーザレベル実時間スレッドの高速化技法, 電子情報通信学会論文誌, Vol.J79-D-I, No.11, pp.946-953 (1996-11).

[4] 多田好克, 寺田実: 移植性・拡張性に優れた C のコルーチンライブラリー実現法, 電子情報通信学会論文誌, Vol.J-73-D-I, No.12, pp.961–970 (1990-12).

[5] Frank Mueller: A Library Implementation of POSIX Threads under UNIX, Proc. the Winter 1993 USENIX Technical Conf., pp.29–41, (1993).

[6] C. Provenzano: Pthreads, `http://www.mit.edu:8001/people/proven/pthreads.html` (1996).

[7] Roland Mechler: A Portable Real Time Threads Environment, master thesis of Univ. of British Columbia, (1997).

[8] 坂本力, 他: 並列性と移植性をもつユーザレベルスレッドライブラリー PPL の設計および実装, 電子情報通信学会論文誌, Vol.J80-D-I, No.1, pp.42–49 (1997).

[9] 小熊寿, 他: SMP 型計算機を活用する軽量プロセスライブラリ, 情処研報 97-PRO-16, Vol.97, No.112, pp.13–18, (1997).

[10] ISO/IEC: POSIX Part 1: System Application Program Interface (API) [C Language], ISO/IEC 9945-1 (ANSI/IEEE Std 1003.1), (1996)

[11] Peterson, G. L.: Myths about the Mutual Exclusion Problem, *Information Processing Letters*, Vol.12, pp.115–116, (1981).