システムソフトウェアと
オペレーティング・システム 61－1
（１９９３．８．１８）

マルチタスク環境における ファイルの圧縮解凍機構の実現

稲田 太　　多田 好克

電気通信大学 電子情報学科　　電気通信大学 情報システム学研究科

ファイルを自動的に圧縮解凍する機構を、UNIX のようなマルチユーザ、マルチタスクのオペレーティングシステム上で実現することを考えた。本研究では、UNIX の Version7 と同じインターフェイスをもつ MINIX 上に、この様な機能を持つシステム ACOMS(Automatic COMpression System) を実現した。

ACOMS では、圧縮アルゴリズムとして、符合化には時間がかかるが復号化は非常に高速に行なうことができる、LZSS 符合化アルゴリズムを使用した。そして、コンピュータの負荷が少ない時に、一定期間使用されていないファイルを圧縮し、ユーザがそのファイルにアクセスした時、自動的に解凍する仕組みを実現した。つまり、ACOMS では、ユーザがファイルの圧縮解凍という作業を意識せずに済むように工夫した。

An Implementation of the Automatic Compression File System on a Multitask Operating System

Futoshi Inada, Yoshikatsu Tada

University of Electro-Communications,
1-5-1 Choufugaoka,Choufu-shi Tokyo 182 JAPAN

An automatic file compression mechanism on multi-task and multiuser Operating System, such as UNIX, is introduced. Then we implemented this mechanism on MINIX that has the same interface with Version7 UNIX.

we used LZSS coding algorithm that can decode files very fast but encodes slowly. We implemented ACOMS(Automatic COMpression System) that compresses files automatically. ACOMS compresses files that have not been accessed certain period of time, provided CPU is idling, and it uncompresses a compressed file automatically on demand.

1 はじめに

ディスクの大容量化にともない、ユーザはより多くのファイルを所有できるようになった。その一方で、システム自身が必要とするファイル量も増加している。ユーザとシステムのファイル量増加に伴い、ディスクの容量不足に苦しむユーザは少なくない。そして、そのファイルのほとんどは、参照されることなくディスク上に保存されていることが多い。

ディスクの容量が不足した場合、ユーザは不必要なファイルを削除したり、他の媒体に移したり、圧縮したりすることにより、空き容量を増やす。しかし、この様な作業は、ファイル数が増えれば増えるほど、ユーザの負担となる。

もし、システム自身が参照されていないファイルを自動的に検索して圧縮し、ユーザが必要とした時に自動的に解凍することができれば、ユーザの負担は軽減される。パーソナルコンピュータの分野では、実際にこの様なシステムが実現されており、資源の有効利用に役立っている。

そこで、ファイルを自動的に圧縮解凍する機構を、UNIX のようなマルチユーザ、マルチタスクのオペレーティングシステム上で実現することを考えた。本研究では、UNIX[1] の Version7 と同じインターフェイスをもつMINIX[2] 上にこの様な機能を持つシステム、ACOMS(Automatic COMpression System) を実現した。

ACOMS では、圧縮アルゴリズムとして、符合化には時間がかかるが復号化は非常に高速に行なうことができる、LZSS 符合化 [3] アルゴリズムを使用した。そして、コンピュータの負荷が少ない時に、一定期間使用されていないファイルを圧縮し、ユーザがそのファイルにアクセスした時、自動的に解凍する仕組みを実現した。つまり、ACOMS では、ユーザがファイルの圧縮解凍という作業を意識せずに済むように工夫した。

本稿では、第 2 章で MS-DOS 上のファイルの圧縮解凍機構について考察する。第 3 章では、システム実現の方針について述べ、第 4 章で、システムの実現上の問題点とその解決法について議論する。そして、第 5 章でシステムの評価を行なう。

2 MS-DOS 上における圧縮解凍機構

MS−DOS 上の圧縮解凍機構は、ディスクの空の部分に、その容量と同じ大きさのボリュームファイルを作成する。そして、デバイスドライバが作成されたボリュームファイルを 1 つの仮想ドライブとみなし、図 1に示すように、そこへ読み書きされるデータを圧縮解凍する。デバイスドライバは、ユーザが仮想ドライブに対し WRITE、READ システムコールを発行したとき、データの符号化、復号化を行なう。

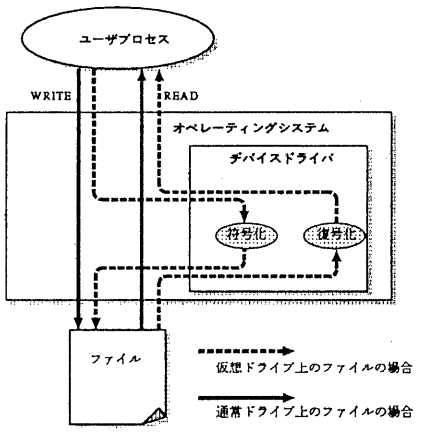

図 1: MS-DOS のファイルの圧縮解凍機構におけるファイルの読み書き

3 マルチタスク環境の圧縮解凍機構

MS−DOS が 1 つのパーティションをドライブとしてアクセスするのに対し、UNIX では複数のパーティションをマウントすることにより、1 つのディレクトリツリーとしてアクセスする。また、MS−DOS ではユーザはファイルシステ

ム全てに対し、読み書き可能であるのに対し、UNIX はアクセス許可されているディレクトリしかアクセスできない。この様な違いにより、MS-DOS と同じ方法で、UNIX 上に圧縮解凍機構を実現することができない。

3.1 圧縮解凍機構の方針

ACOMS は、次の 2 つの方針に基づき設計されている。

- ユーザに圧縮解凍の作業を意識させない
- 圧縮解凍のオーバーヘッドを小さくする

3.2 システムの概要

ACOMS は次の 2 つの部分により構成されている。

[圧縮機構]
ファイルの圧縮はデーモンプロセス (圧縮デーモン) が行う。圧縮デーモンは、計算機の負荷が小さい時、圧縮されてないファイルをディスク上から選び出す。そして、そのファイルが一定期間アクセスされていなければ、それを圧縮する。

[解凍機構]
圧縮ファイルをユーザがオープンした時、OPEN システムコールは、そのファイルを解凍する。そして、解凍後のファイルへのファイルディスクリプタをユーザに返す。

図2からもわかる様に、CPU 資源は常に使用されているわけではない。この様に、圧縮を計算機の負荷の小さい時、デーモンプロセスに行なわせることにより、圧縮のオーバーヘッドが他のプロセスに影響を与えないようにした。この方法は、CPU 資源の有効利用につながり、圧縮の部分をオペレーティングシステム外で実現することにより、オペレーティングシステムのサイズの増加を少なくすることが出来る。

一方、解凍機構は高速性が要求されるので、オペレーティングシステム内で行なうようにした。ただし、LZSS アルゴリズムの復号化は非常に高速に行なうことができ、アルゴリズムも単純である。このため、オペレーティングシステムのサイズの増加は少なく、解凍時のオーバーヘッドも小さい。

別の実現法として、READ、WRITE システムコール実行時に、データの復号化、符号化を行なう方法も考えられるが、ACOMS では以下の理由によりこの方法を取らなかった。

- 圧縮はデータの前後関係に依存するため、単に読み出す場合は良いが、LSEEK システムコールが発行された場合や、書き込みの場合、ファイル全てを解凍しなければならない。
- 圧縮解凍の実行によるオーバーヘッドが、他のユーザのタスクへも影響を与える。

3.2.1 圧縮を行なわないファイル

以下のファイルに対しては圧縮を行なってはならない。

- 他のプロセスにより、現在参照されているファイル
- 最近アクセスされたファイル

また、圧縮を行なった結果、利潤が 1 ブロック未満だった場合、圧縮前の状態に戻す。

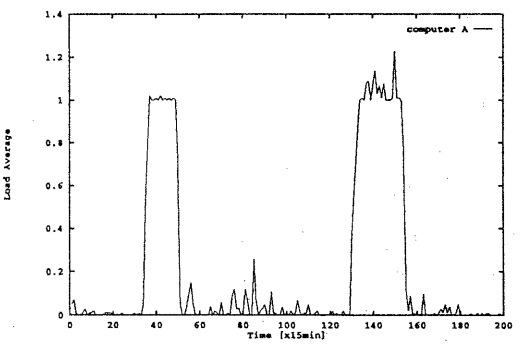

図 2: コンピュータの負荷の推移 (15 分間の平均)

```
#define I_TYPE              0170000         /* inode type        */
#define I_REGULAR           0100000         /* regular file      */
#define I_BLOCK_SPECIAL     0060000         /* block special     */
#define I_DIRECTORY         0040000         /* directory         */
#define I_CHAR_SPECIAL      0020000         /* character special */
```

図 3: i ノードのファイル種別の定義

```
#define I_COMPRESS          0010000         /* compress file     */
```

図 4: 圧縮ファイルのモードの定義

4 MINIX 上での実現

4.1 圧縮ファイルの識別

ファイルシステムプロセスは、圧縮ファイルと通常のファイルを区別しなければならない。そこで、ACOMS では、i ノードの属性情報に新たにモードを追加した。図3に i ノードのファイル種別の定義を示し、図4に圧縮ファイルのモードの定義を示す。

この様な変更を i ノードに加えることにより、以下の問題が発生した。

[問題点 1] ユーザが STAT システムコールを用いてファイルの属性を取得し、新たに追加したビットが 0 であると仮定して使用した場合、そのプログラムが正しく動作しない可能性がある。

[問題点 2] fsck コマンドを実行すると、圧縮ファイルの i ノードの属性に対して、エラーが出る。

[問題点 1] は、STAT システムコールを変更し、圧縮ファイルの場合も通常のファイルとしてユーザに値を返せば解決することができる。しかし、この変更を行なうと、圧縮デーモンが圧縮を行なう場合、そのファイルが既に圧縮されているのかどうかを知る方法がなくなってしまう。これを知るためのシステムコールを、新たに追加することは好ましくない。また、ユーザのプログラム次第では正常に動作するし、実際 ls コマンドは問題なく動いている。そこで、この問題に対する解決は行なわなかった。

[問題点 2] は、fsck コマンドを修正しなければならない。これは、fsck コマンドがディスクブロックを直接読み込み、チェックしているため、オペレーティングシステムへの変更では対応できないからである。

4.2 ファイルの解凍機構の実現

ファイルの解凍機構は、OPEN システムコールを変更することにより実現した。ユーザが OPEN システムコールによりファイルをオープンした時、そのファイルが圧縮ファイルであれば、そのファイルを解凍し、解凍した結果のファイルへのファイルディスクリプタを返す。

ファイルの解凍は、次のような手順で行なった。

1. 解凍するファイルと同じファイルシステム上に、一時ファイルを作成する。

2. 圧縮ファイルからブロック毎にデータを読み込み、解凍して一時ファイルに書き込む。

3. 一時ファイルと圧縮ファイルのゾーン番号、ファイルサイズをそれぞれ入れ換える。

4. 一時ファイルのiノードのリンク数を0にして、一時ファイルを削除する。

5. 一時ファイルと圧縮ファイルのiノードをディスクに戻す。

4.3 ファイルの圧縮機構の実現

圧縮デーモンの実現には、以下の問題がある。

[問題点1] 圧縮中のファイルにユーザがアクセスした場合の処理をどうするか。

[問題点2] 圧縮によってファイルの更新時刻が書き換えられ、makeなどのコマンドに影響を与える。

[問題点3] 圧縮前のファイルサイズを取得できない。

[問題点1] は、圧縮するファイルをMINIXのファイルシステムプロセスに知らせ、他のプロセスがそのファイルをオープンした時、ファイルシステムプロセスが圧縮デーモンにシグナルを送るようにして対応した。ACOMSでは、ファイルシステムプロセスに圧縮するファイルを知らせるために、新規にシステムコールを追加した。

[問題点2] は、圧縮対象ファイルと一時ファイルのiノードの内容を入れ換えるシステムコールを追加することにより対応した。

[問題点3] については、iノード内に新たに圧縮前のファイルサイズを格納する領域を追加すれば、この問題を解決することができる。しかし、この方法はファイルシステム全体の再構築が必要であるため、ACOMSでは未解決のままである。

4.4 圧縮の手順

圧縮デーモンは、まだ圧縮されていないファイルを検出する。次に、そのファイルが圧縮しても良いファイルかどうかチェックし、圧縮しても良い場合には、以下の手順で圧縮を行なう。

1. 圧縮するファイルをファイルシステムに知らせる。

2. 圧縮するファイルと同じファイルシステムに、一時ファイルを作成する。

3. 圧縮するファイルからデータを読み込み、それを圧縮し、一時ファイルに書き込む。(図5参照)

4. システムコールにより、圧縮元のファイルと、一時ファイルのゾーン番号とファイルサイズを入れ換える。(図6参照)

5. 一時ファイルを削除する。

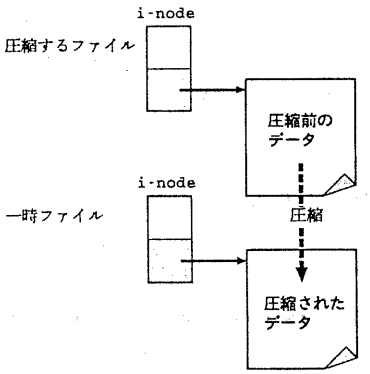

図5: データの圧縮

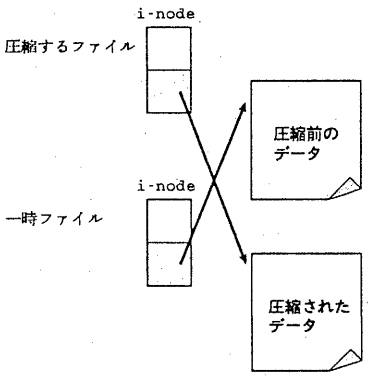

図6: iノード間のゾーン番号の入れ換え

4.5 新規に追加したシステムコール

ACOMS では、圧縮デーモンのために、新規に2つのシステムコールを追加した。これらのシステムコールは圧縮デーモンのみが使用する。

STCOMP システムコール

```
fd = stcomp(fname);
     int fd;
     char *fname;
```

STCOMP システムコールは、引数で与えたファイルを読み込みモードでオープンし、そのファイルへのファイルディスクリプタを返す。そして、ファイルシステムプロセスはそのファイルのiノード番号と、システムコールを発行したプロセスのPIDを記憶し、そのファイルが他のプロセスによりオープンされた時に、このシステムコールを発行したプロセスへシグナルを発行する。

EXCHZONE システムコール

```
exchzone(fd1,fd2)
    int fd1,fd2;
```

EXCHZONE システムコールは、引数で与えられた2つのファイルディスクリプタに対応するiノードの、ゾーン番号、ファイルサイズ、属性情報を入れ換える。

圧縮対象ファイルのファイルシステムへの通知は、新しいシステムコールを追加しなくても、オープンシステムコールに新たなモードを追加すれば実現可能である。ところが、ファイルシステム内でオープンシステムコールを発行したプロセスのPIDを取得することが出来ない。PIDが取得できなければ、KILL システムコールを発行して、圧縮デーモンに圧縮の中断をさせることが出来ない。オープンシステムコールのメッセージに呼出側のPIDを入れようとしたが、メッセージにそれを入れる領域がなかった。そこで、ACOMS ではSTCOMP という新しいシステムコールを追加した。

4.6 圧縮の中断

別のプロセスが圧縮中のファイルをオープンしようとした場合、圧縮デーモンは圧縮の作業を中断しなければならない。ACOMS では、別のプロセスが圧縮中のファイルをオープンした時、ファイルシステムプロセスから圧縮デーモンにシグナルを送信することにより、この問題を解決した。シグナルは次の手順で送信し、図7に処理の流れを示した。

① ユーザプロセスが圧縮中のファイルに対し、オープンシステムコールを発行する。

② ファイルシステム・プロセスからメモリマネージャにKILL システムコールの発行を要求する。

③ メモリマネージャから圧縮デーモンへシグナルを送る。

④ 圧縮デーモンのシグナルハンドラが圧縮の中断処理を行なう。

4.7 MINIX 上での実現の限界

MINIX は UNIX とシステムコールレベルで互換性があるが、 UNIX に存在するシステムコールやシステムの情報のなかで、特に、必要ないとされるものは省略されている。その中で、次の2つがACOMS において問題になる。

- NICE システムコールが存在しないため圧縮デーモンの優先順位を下げることができない。ファイルの圧縮はCPU 資源を大量に消費する処理であるため、圧縮デーモンがシステムの処理効率の劣化を招く。

- MINIX では、最後にファイルが更新された時刻しか記録されていない。そのため、実行形式のファイルなど読み出ししか行なわれないファイルは、圧縮と解凍を繰り返す結果となる。一方、UNIX でACOMS を実現する場合には、最終アクセス時刻も記録されているのでこの様な問題は発生しない。

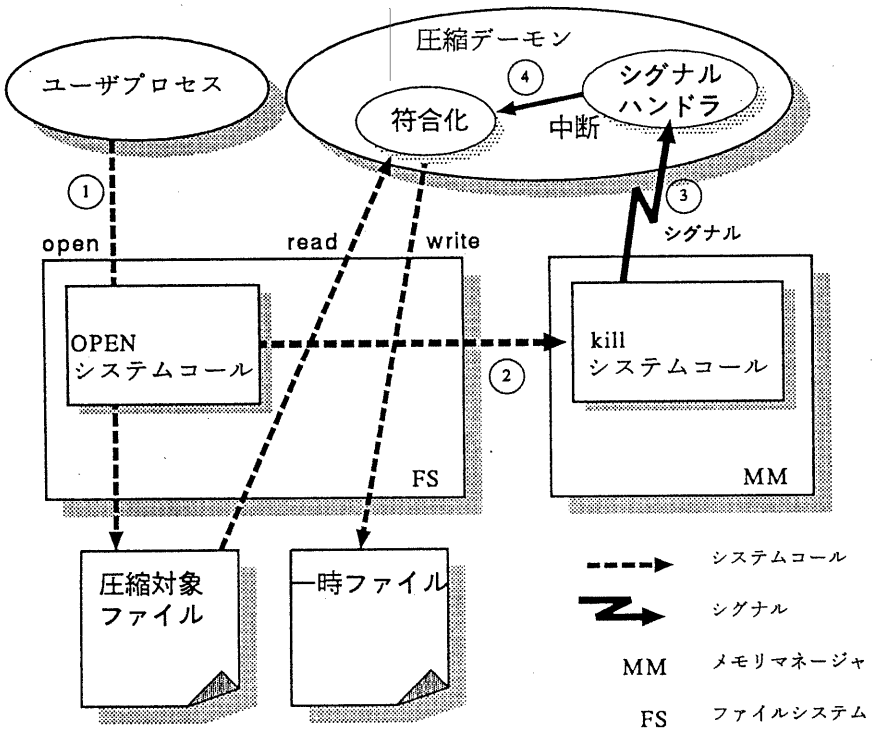

図 7: 圧縮の中断処理

5 システムの評価

ACOMS は MINIX 1.2 上で実現されており、PC-9801VM(CPU V30 10MHz) で、cat コマンドを実行した時の圧縮ファイルと非圧縮ファイルの表示時間を計測した。この時の平均圧縮率は 42.3%であった。時間の計測には time コマンドを使用し、ファイルのサイズと実行時間関係を調べた。図 8に示すように、約 25%のオーバヘッドが発生した。ただし、このオーバーヘッドは、圧縮ファイルに最初にアクセスした時にだけ生じる。1 度アクセスすると、それ以後、通常のファイルとしてアクセスできるので、ユーザにとって大きなオーバーヘッドではない。

一方、圧縮デーモンがファイルを圧縮するのに要する時間は、約70byte/sec と非常に遅いが、圧縮はコンピュータの負荷の小さい時、自動的に行なわれるので、この時間は問題ではない。

また、表 1に示すように、圧縮解凍機構の実現に伴うファイルシステムプロセスのサイズの増加は小さい。これは、解凍のアルゴリズムが非常に簡単であるためである。

表 1: ファイルシステムプロセスのサイズの増加

	変更前 [byte]	変更後 [byte]
text	17280	19312
data	1288	1312
bss	25040	25168
TOTAL	43608	45792

6 今後の課題

ACOMS、次のような未解決の問題が残されている。

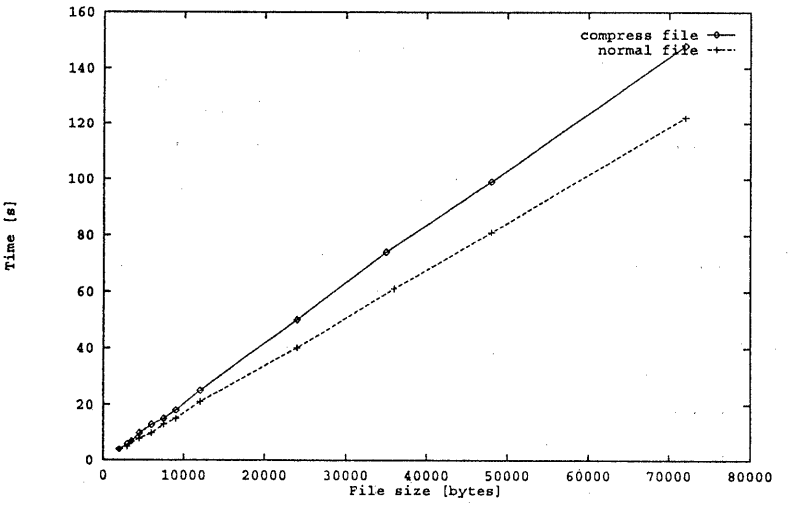

図 8: ファイル解凍のオーバーヘッド

- 圧縮ファイルを解凍する際、一時ファイルを作成する容量がディスクに存在しなければ、そのファイルを自動的に解凍することができない。
- 圧縮ファイルを解凍せずに、圧縮前のファイルの大きさを知る方法がない。
- 実行可能ファイルを圧縮すると、実行できない。
- 空のゾーンを含むファイルを圧縮するとファイルサイズは小さくなるが、消費するゾーン数は増加する。

また、本システムは MINIX 1.2 上で実現されているが、現在、MINIX 1.5 へ移植中である。オペレーティングシステムのバージョンを上げるとともに、次の事項についても検討中である。

- 圧縮する (または、圧縮しない) ディレクトリをユーザが指定できるようにする。
- 現在、ユーザが圧縮ファイルをオープンした時、読み込みのみであっても圧縮されたデータを捨てているが、読み込みだけの場合には、圧縮されたデータを残すようにする。

これらの問題を解決することができれば、本システムは実際に使用できるものとなる。

参考文献

[1] 坂本 文, 多田 好克, 村井 純訳, UNIX カーネルの設計: 共立出版, 東京 (1991).

[2] 坂本 文(監訳), MINIX オペレーティング・システム: アスキー出版局, 東京 (1989).

[3] Timothy C.Bell,
Better OPM/L Text Compression,
IEEE,Trans. Commun.,VOL.COM-34,
No.12,1986,pp.1176-1182.