UNIX のファイルシステム アーキテクチャ

関 本 年 彦

序

AT&T ベル研究所の D. M. Ritchie と K. Thompson が，UNIX の原形となったオペレーティングシステムを作り始めたのは 1969 年である。当然，UNIX の元祖は AT&T であるが，もう一つ UNIX に強い影響力を持つのがカルフォルニア大学バークレイ校で，ここで開発されている UNIX は BSD (Berkeley Software Distribution) 版として知られ，大学を始めとするいわゆるアカデミック分野でよく利用されている。本稿は，主として現在配布されている BSD4.3 版を対象としており，東京大学大形計算機センタの副システムで稼働している UNIX の使用経験に基づいて執筆したものである。なお，文中 tansei とあるのは，この副システムの名称である。

UNIX のファイルシステムは，UNIX の最も個性的な部分で，以下のような特色を持っている。

(1) ファイルシステム全体が樹形構造を成す。

(2) カーネルから見たデータ構造は，いづれも均一なバイト列である。したがって，伝統的なメインフレーム OS がもつレコードの概念や，複数のアクセス方式がない。

(3) 利用者から見た入出力装置は，すべて一個のファイルである。

このうち，はじめの二点は，最近でこそ，UNIX の影響でほとんどのパソコン OS が採り入れており目新しいものではなくなったが，IBM 系 OS 万能の時代にあっては極めて斬新な印象を与えたものであり，またこれが実

用的に定着したのは UNIX に負うものであろう。

本稿は，このような UNIX ファイル・システムの分析と考察を試みるのが目的である。

§1. ファイルシステムの階層構造

UNIX ファイル・システムは，データが記録される《ファイル》とそれが登録される《ディレクトリ》とを構成要素とし，一つの処理系下にある全てのファイルとディレクトリを階層的に編成し管理している。これは，図1.1のような樹形構造として図示することができる。以下ではファイル，ディレクトリを共にファイルと呼ぶが，必要があれば前者を《データファイル》ということにする[1]。

樹形構造のもっとも根元にあるディレクトリを《ルート》といい，全てのファイルは，この樹形構造を遡るとルートに辿り着く。ルートは，スラッシュ記号／で表される。たとえば，

/alpha/beta/gamma

をファイル gamma の《パス名 (path name)》というが，この左端の“/”はルートを表し，そのあとにルートから gamma に至る間にあるファイル名がスラッシュ記号で区切って書かれている。したがって，左端以外の“/”は，単なる区切り記号である。パス名の末尾のファイル名は，一般にはディレクトリを指すこともあり，パス名だけを与えられてもその末尾の名前がデータファイル，ディレクトリのいずれを指すかは判定できない。

---

1) 初期の UNIX においては，ディレクトリの構造も純然たるバイト列であったが，大形 UNIX が出現してからは，システム処理効率を左右するディレクトリ検索速度を向上させるため，ディレクトリは後述するような特有の構造を持つようになった。

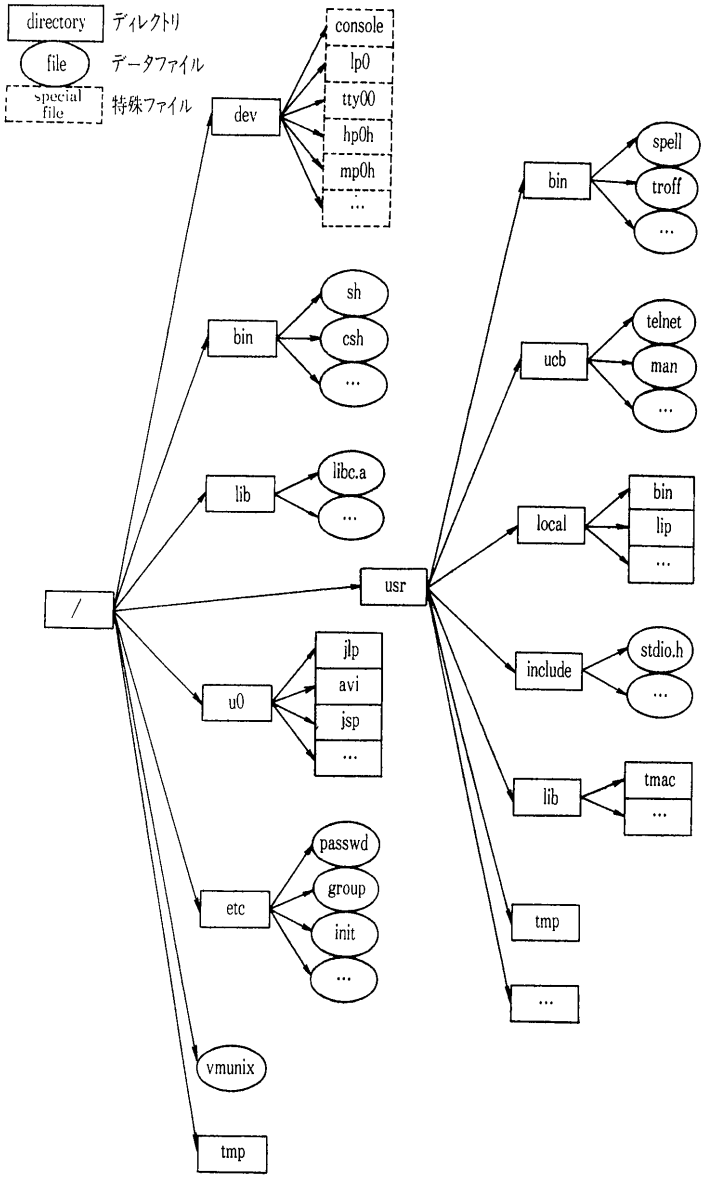

図1.1　ファイルシステムの構造

利用時，ファイル指定のために，いちいちルートから始まる長いパス名をキー入力するのは面倒である。そこで，利用者は，シェル[2]・コマンド change working directory (cd) を用いて，任意のディレクトリをワーキング・ディレクトリとして指定することができる。たとえば

cd /alpha/beta

とキー入力することによってワーキング・ディレクトリは/alpha/beta となり，ファイル gamma を/alpha/beta/gamma ではなく，単に gamma と指定することができる。ルートから書き始めたパス名を《絶対 (absolute) パス名》といい，ルート以外のファイル名から書き始めたものを《相対 (relative) パス名》と言っている。

一般のシステム運用においては，利用者から利用申請があるとその利用者専用のディレクトリ（通常，ユーザ ID をその名前とする）が《ホームディレクトリ》として新設され，その利用者の login 時の都度，システムはこのホームディレクトリを自動的にワーキング・ディレクトリとして登録するようにしてある。ワーキング・ディレクトリをホームディレクトリに戻す必要があるときには，アーギュメント無しの change working directory コマンド

cd

あるいは，シェルのキーワード変数 home を用いて

cd home

を実行する。

ディレクトリの内容表示には，シェル・コマンド list contents of directory (ls) を，また，ディレクトリの新設および削除には，それぞれ

---

2) シェル (shell) は，UNIX が提供するインタプリタ形式のユーティリティ群で，現在は William Joy が開発したCシェルが最も広く用いられているが，このほか Bourne シェルや Korn シェルがある。

make a directory (mkdir) と remove directories (rmdir) を用いる。

データファイルには，複数の名称を与えることができる。たとえば，既存のデータファイル tokyo に別称 osaka を与えるのにはシェル・コマンド make links (ln) を用いて

ln tokyo osaka

とキー入力すればよい。データファイル名をリンクということもあるが，一つのデータファイルに対する複数のリンクは，異なるディレクトリに属していてもよく，さらに，いったん複数のリンクが生成されると，それらはシステム側からも生成順序とは無関係に平等な取扱いを受ける。

データファイルの削除には，シェル・コマンド remove files (rm) を用いる。

rm tokyo

とキー入力した場合，ほかにリンクが無ければデータファイル tokyo は完全に削除されるが，このデータファイルに対して他にもリンクがある場合にはディレクトリからリンク tokyo が削除されるだけであって，データファイルと他のリンクはそのまま残る。ディレクトリの各データファイル・エントリは，その一項目としてリンク数を保持しており，このリンク数はシェル・コマンド make links (ln) および remove files によってそれぞれ増減され，リンク数が 0 になった時データファイルが実際に削除されるのである。

§2. 利用者とのインタフェイス

UNIX は，通常C言語によるプログラムの中で用いられるサブプログラム形式の《システムコール》群を用意して種々のシステムサービスを提供しており，たとえば，アプリケーション・プログラムがファイルに対する読出し・書込みを行う際には入出力関係のシステムコールを利用する。

前述のように，UNIX は全てのファイル内容を単にバイト列と見なして

おり，入出力システムコールの体系は極めて簡素である。UNIX は，IBM 系 OS のレコードのような概念を持たず，またファイル領域の拡張を必要に応じて自動的に行い，利用者はファイル新設に際してファイル・サイズ等の諸パラメータを与える必要が無い。しかしながら，キーワード等ファイル内容に基づく情報検索に関しては，UNIX は，シェルを通して諸ユーティリティを提供しているものの，アプリケーション・プログラムが利用できるような機能は提供しておらず，一般の利用者は，情報検索用プログラムの作成に当たっては UNIX の素朴なランダムアクセス機能を基に必要なプログラムモデュールを自足しなければならない。

以下にファイルシステムに関連するいくつかのシステムコールについて述べる。各システムコールの形式については，細部の議論を省くため若干簡略した形で提示する。

open システム・コールは，*pathname* で指定された既設のファイルを開き，以後の read および write システム・コールにおいてファイル指定に用いられる《ファイル指定子（file descriptor)》と呼ばれる正整数を変数 *fd* に持ち帰る：

fd＝open（*pathname, flag*）

flag には，読出し，あるいは書込み等ファイル操作の種類を指定する。システムは，とくにファイルを開けなかった場合に *fd* に－1を入れる。creat システムコールは，新設ファイルを開いたり，あるいは始めから書き直すために既設ファイルを開く。

通常，ファイルに対する読み書きは，順次的に行われる。すなわち，ファイルに対する読み書きは，前回読み書きが行われた最終バイトの直後のバイトから行わる。このため，システム領域内には，開かれているファイルごとに最初のバイトからつぎに読み書きが行われるべきバイトまでのオフセット（バイト差）を表示する32ﾋﾞｯﾄのポインタが用意されており，システムは，nﾋﾞｯﾄの読出し，あるいは書込みがあると，このオフセットをnだけ

増やす。

開かれたファイルに対しては

n =read (*fd, buffer, count*)

n =write (*fd, buffer, count*)

等のシステムコールでそれぞれ読出し，あるいは書込みを行う。いずれの場合も，システムは，*fd* で指定されたファイルと *buffer* で指定されたバイト配列との間において，*count* で指定されたバイト数だけデータ転送を行い，実際に転送したバイト数を n に置く。$n \neq count$ は，write システムコールにおいては異状の発生を意味するのが普通であるが，read システムコールにおいてはファイルの残余データが *count* で指定したバイト数より少ないこともあって，この時は $n < count$ となる。また，read システムコールにおいて，$n=0$ は，残余データが無いこと，すなわち，ファイル末尾に到達したことを示す。

ファイルに対してランダムアクセス的な読み書きを行うには，move read／write pointer (lseek) システムコールを用いる。すなわち，このシステムコールは，ファイル指定子で指定されたファイルのオフセット・ポインタを指定された数だけ増減する。

§3. ディスク上のデータ構造

ファイルシステムが占めるディスク領域は，順次番号がつけられた一定の大きさの《ブロック》に分割されて用いられる。ブロックの大きさ（ブロック長）は，当初512バイトから出発したが，メモリとディスク間のデータ転送はブロック単位で行われることが多く，ブロック長は大きい方がスループット (throughput) が向上するので，近年メモリおよびディスク容量の大形化にともなってブロック長は漸増の傾向にあり，BSD4.2版でのブロック長は4096バイトか8192バイトのいずれかになっている。一方，UNIX の利用者ファイルの多くは小ファイルであるためブロック長が大きいほど無駄な部分

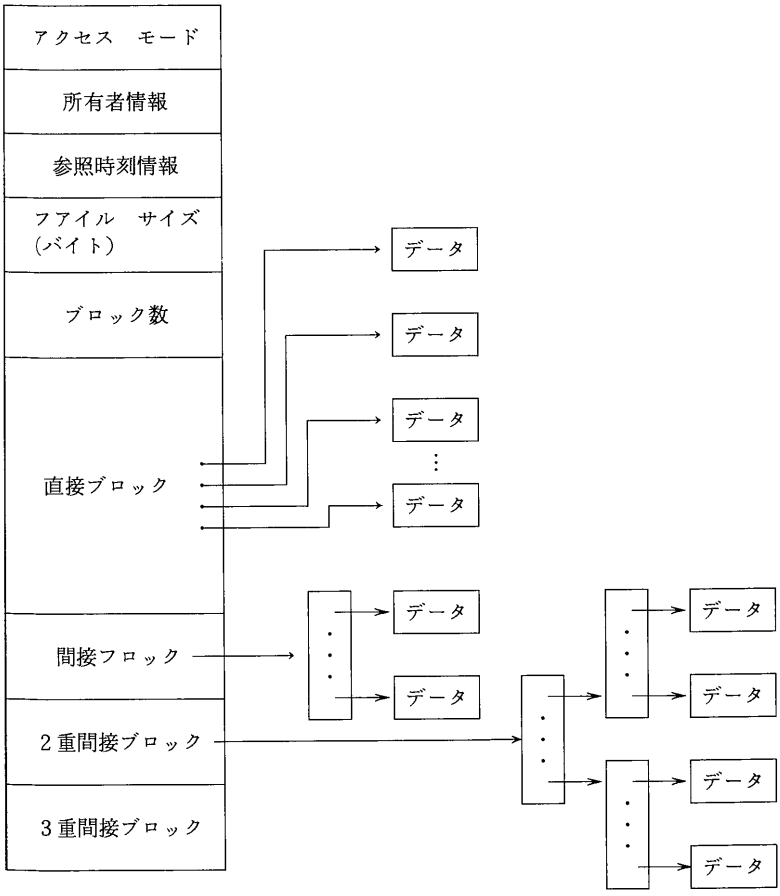

図3.1　iノード

ファイルシステムの内容がこのバイト数を超えると間接ポインタを使用することになる。ブロックポインタは4バイトから成り，4098バイトブロックには1024個のポインタが収納できるので，一重間接ポインタを用いて指定できるバイト総数は$1024\times4098=2^{24}=4M$であり，二重間接ポインタを用いて指定できるバイト総数は$1024^2\times4098=4G$（ギガ）である。ファイルシステ

ム内の特定バイトへのアクセスは前述したオフセットによるが，システムはこれを32ビットとしているため，三重ポインタの使用が可能でありながら現在のところ実際に用いられることはない。

シリンダグループは，上述のようにスーパブロック，ｉノード，データブロックの三つの部分から成るが，ディスク障害などによってスーパブロックあるいはｉノードが損傷を受けた場合，一般に大量のデータが修復不能となる。そこで，シリンダグループ内のこれら三部分の配置については，頭部にそれぞれのシリンダグループで異なった大きさの小データブロック部を置き，その後にスーパブロック，ｉノードおよび残りのデータブロックを置くことによって，各シリンダグループのスーパブロックとｉノードが同一のディスクプラッタに集中しないようデータ保全のための配慮が払われている。(図3.2)

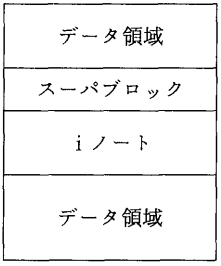

図3.2　シリンダグループ

ディレクトリは，ファイル名の検索・登録・削除などの UNIX において頻度の高い処理の対象であることから,特有のデータ構造をもっている。ディスクに対する入出力操作の最小単位はセクタで，これは通常512バイトであることから，ディレクトリのブロック内は512バイトのデータ塊に分割され，ディレクトリからの読出しはこのデータ塊単位で行われる。ファイル名の長さは，最大255文字で可変長であり,したがって，ディレクトリの各エントリも可変長でつぎのような４項目から成っている：

1. ｉ番号（４）
2. エントリ長（２）
3. ファイル名長（２）
4. ファイル名＋［空領域］

（ ） 内はフィールド長を示すバイト数。

ように，同一ディレクトリ内のファイルのｉノードは同時にその全部が参照されることが多いので，同一ディレクトリに属する各ファイルのｉノードは，原則として親ディレクトリのｉノードと同じシリンダグループ内に置くことにしている。一方，各シリンダグループ使用率の均等化のため，ディレクトリのｉノードだけは，全シリンダグループ中から使用ｉノード数が平均以下でディレクトリ数が最小であるものを選び，そこに置いている。

つぎに，データブロックの割当ての方であるが，書換え頻度の高い小ファイルについては，全データブロックのｉノードと同一シリンダグループ内への集中化を図り，順次処理の対象となることの多い大形ファイルについては，一定数の論理的に連続なブロックを同一シリンダグループに配置後シリンダグループを変える方式を採り，シリンダグループ使用率の均等化を図りながらも遠隔シリンダへのアクセス頻度を低く抑えている。すなわち，直接ポインタで指定できるはじめの48Ｋないしは96Ｋバイトについてはファイルのｉノードと同一のシリンダグループからの空ブロックを選んで割り当て，間接ポインタが指定する部分については1Ｍバイトごとにシリンダグループを変え，そこから空ブロックを選び出している。この場合新しいシリンダグループは，平均以上の空ブロック数をもつ後続シリンダグループの中から最も近いものが選ばれる。

§5. パス名からｉノードへの変換

利用者はパス名によってファイル指定をするが，システムはｉノードに基づいてファイルへアクセスする。システムが行う処理の中で，このパス名からｉノードへの変換はシステム時間全体の中で占める割合が最も高く，約25パーセントにも上るというデータがある。

システムがパス名/alpha/beta/gamma から gamma のｉノードを導出する過程を見よう。ルートディレクトリのｉ番号は，UNIX の慣行で各フ

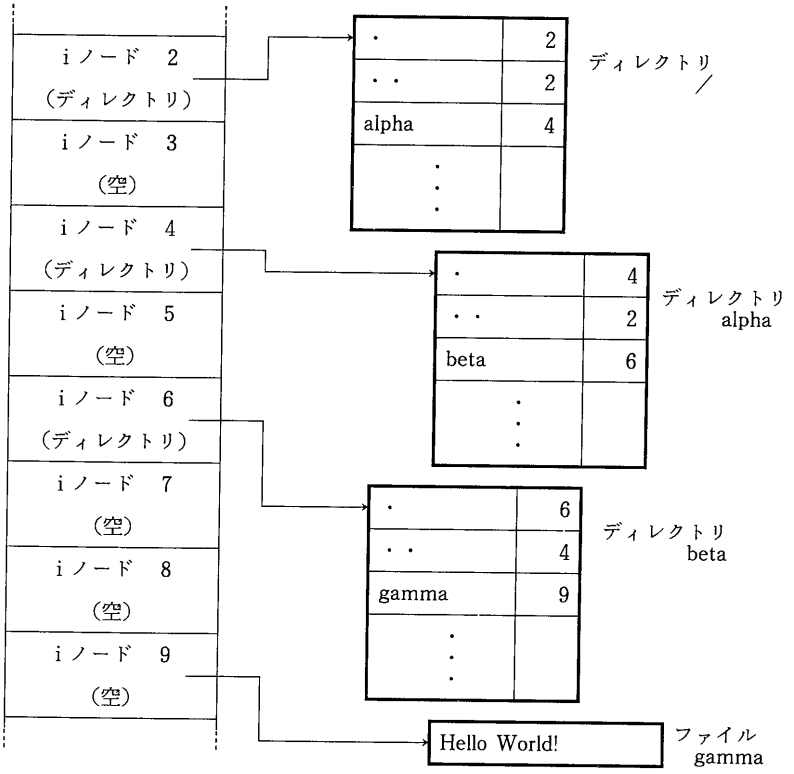

図5.1　iノード変換

ァイルシステム共通に2である。そこで，ルートファイルシステムの第2iノード枠にあるiノードをメモリに読み出し，それが持つブロックポインタにしたがって，ルートディレクトリをディスクからメモリに読み出す。つぎに，ディレクトリ alpha の内容の読出しを行う。

(1)　メモリに読み出したルートディレクトリに対してファイル名 alpha をキーワードとする検索を行ってi番号4をひき出す。

(2)　ファイルシステムから，第4iノード枠のiノードをメモリに読み出し，さらに，このiノードのブロックポイントにしたがってディレクト

リ alpha のディスクブロックの内容をメモリへ読み出す。(図5.1)

パス名からiノードへの変換において，ディレクトリ・エントリを一つずつ検索して行く上記(1)の部分を内側ループという。この後，ファイル beta, gamma について(1)，(2)からなる部分が外側ループである。

UNIX には，システム領域内にファイルシステムの各種のデータのキャッシュ（cache）を設けている。iノードもその対象の一つで，システムはその領域内に一定容量のキャッシュを設けており，あるiノードを必要とする時には，実際には，ディスクから読み出す前にこのキャッシュを検索し，見つからなかったときはじめてディスクから読み出す。読み出されたiノードはキャッシュに収納され，システムからの参照がない時間がキャッシュ内のiノードの中で最長になった場合に，あらたにファイルシステムから読み込まれるiノードに置き換えられ，キャッシュから消去される。

UNIX のファイルシステムは，前述のように物理的には複数のファイルシステムから構成されている。各物理的ファイルシステムは，ディスクのパーティッション，ディスクパック，光ディスク等に構築され，既存ファイルシステム内のあるディレクトリを新ファイルシステムのルートディレクトリに結合させることにより装塡（mount）される。システムは，装塡[8]（mount）されている全てのファイルシステムの表をファイル/etc/fstabに保存しており，表5.1は tansei のそれの一部である。UNIX の各周辺装置は，ファイルシステム内のその特性データが記録されている《周辺装置特殊ファイル（device special file)》を持っているが，たとえば表5.1の1行目に見える/dev/ra0a はディスク装置 ra0 のパーティッションaに対応する周辺装置特殊ファイルのパス名である。実際のファイルシステムの装塡には mount シェルコマンドを用いるが，表5.1の2行目に見えるファイルシステムは，たとえば，コマンド

```
mkdir/usr
mount/dev/ra0g/usr
```

によって装填されたものである。(表5.1)

/etc/fstab の内容は，メモリのシステム領域に装填表（mount table）として複写されており，一方，各ファイルシステムのルートディレクトリに対応するｉノードは特別なフラッグをもっている。システムは，パス名からｉノードへの変換の際，このフラッグをもったｉノードに出会うと装填表を参照して変換処理を進める。

/dev/ra0a mounted on	/
/dev/ra0g mounted on	/usr
/dev/ra0h mounted on	/usr/usr0
/dev/ra1g mounted on	/usr/adm
/dev/ra1h mounted on	/usr/usr1
/dev/ra2g mounted on	/usr/spool
/dev/ra2h mounted on	/usr/usr2
/dev/ra3g mounted on	/ra3g
/dev/ra3h mounted on	/usr/usr3
/dev/ra4g mounted on	/usr/local
/dev/ra4h mounted on	/tmp
/dev/ra5a mounted on	/root.bak
/dev/ra5g mounted on	/usr/prj
/dev/ra5h mounted on	/usr/usr5
/dev/ra6g mounted on	/usr/prj2
/dev/ra6h mounted on	/usr/usr6
/dev/ra7c mounted on	/usr/usr7
/dev/ra8c mounted on	/usr/spool/oldnews
/dev/ra9c mounted on	/ra9c
/dev/ra10c mounted on	/usr/public

表5.1 装 填 表

ｉノード枠のアドレスであるｉ番号は，各ファイルシステムにおいて個別に割り当てられる0から始まる順番号である。したがって，システムは，パス名が指定するファイルにアクセスする際，ｉ番号だけではなくファイルシステムの ID である装置番号（device number）をも必要とする。この装置番号は，装置特殊ファイルに記録されており，キャッシュに複写されているｉノードは，その一項目にファイルシステムのｉノードにはない装置番号を含んでいる。

ここで，§1で行ったリンクの議論に戻るが，リンクは，一個のデータファイルに対して複数の名前を与える，あるいは，複数のディレクトリに同一ファイルを登録するための機能であった。したがって，一個のデータファイルに対する各リンクは，いずれも同じｉ番号を持ったディレクトリエントリである。ところが，ｉ番号の体系は各ファイルシステムごとに独立

であるから，あるファイルシステム内に別のファイルシステム内のデータファイルに対するリンクを設けることは，上の方式を採る限り不可能である。そこで，このような場合には，《シンボリックリンク（symbolic link)》と称するリンクを設ける。シンボリックリンクは，ソフトリンクとも言われるので，これまでのリンクをシンボリックリンクと区別する場合ハードリンク（hard link）と言う。シンボリックリンクのディレクトリエントリは，i番号の代りに原ファイルのパス名を持っており，システムは，パス名からi番号への変換においてシンボリックリンクに出会うと，それが持つパス名について改めて変換処理を始める。この場合，さらに別のシンボリックリンクに出会う可能性があるが，永久ループに陥る危険性を回避するため，システムは，パス名の変換処理を初めてから8個目のシンボリックリンクに出会うとパス名変換処理を打ち切り，誤りの警告を出す。実際のシンボリックリンクの設定には，−s オプションを指定した make links (ln) コマンドを用いる。

open システムコールは，指定されたパス名をファイル指定子に変換する。このとき，システムは，パス名に対応するiノードを見付け出すためキャッシュを検索し，見つからなければファイルシステムから読み出してキャッシュに収納する。ファイル指定子は，実質的にはキャッシュのiノードに対するポインタである。しかしながら，実際には図5.2のように，利用者プロセスごとに設けられているプロセスの全オープンファイル指定子表と，iノード・キャッシュとの間にはシステム領域内に設置されたファイル表が介在している。複数利用者のプロセスが同一ファイルを同時に参照する場合を考えると，read, write システムコール等で利用されるファイルのバイト位置指定オフセットは，利用者プロセスごとに設置される必要があるが，ファイル表はこの目的に沿って設けられたものである。すなわち，ファイル表には，各プロセスの各オープンファイルに対して個別にエントリが設けられ，それぞれのファイルのオフセットはここに配置されて

いる。(図5.2)

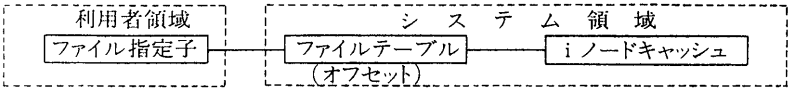

図5·2

ファイル指定子は，ファイル指定子表のエントリインデクスである。UNIX では，《標準入力 (standard input)》はキーボードに，また《標準出力 (standard output)》と《診断出力 (diagnostic output)》はディスプレイ画面に割り当てられており，これらのファイル指定子はそれぞれ0,1,2 (すなわち，ファイル指定子表の第1～3エントリインデクス) と定められている。これら三つの特殊ファイルは，利用者の login 時に自動的に開かれ，その後開かれる各ファイルの指定子は，通常3以上となる。

標準入力あるいは標準出力を用いるシェルコマンドにおいて，とくにそれら以外のファイルを指定するための《つなぎ換え (redirection)》と称する機能があるが，この実質は，指定した標準入出力以外のファイルをいったん開いてファイル指定子表にエントリを作成し，それを標準入出力用のインデクス0または1のエントリ枠へ移すことである。たとえば，シェルコマンド ls>dirf が入力されると，まずファイル dirf のエントリがファイル指定子表に作成され，つぎにその内容がインデクス1をもつ標準出力用エントリ枠へ複写され，ls コマンドが実行される。

§6. バッファキャッシュ

ディスク上のデータファイルに対してはブロック装置インタフェースとキャラクタ装置インタフェースとがあるが，UNIX において重要であり，かつ，特徴的であるのは前者である。そこで，以下では UNIX ファイルシステムのブロック装置インタフェースについて話を進める。

利用者は，ファイルシステムを用いるとき，それがブロックから構成さ

れていることを必ずしも意識する必要はなく，ファイルシステムに対するデータの読み出しや書き込みをブロック単位で行うことは，処理時間について特別な考慮を要するとき以外必要としない。しかし，システムは，あるブロックあるいはフラグメントに対して最初に読出しを行う場合、たとえ利用者プロセスの要求するデータ量が小さいものであっても，いったんブロックあるいはフラグメント全体の内容をシステムバッファに読み出しておき，この時点では要求された部分だけを利用者プロセスへ引き渡す。システムは，このバッファの内容を可能なかぎり保存することを図り，その後にこのブロックあるいはフラグメントに対する読出し要求があると，実際のディスク読出しは行わずバッファに在る複写から所要のデータを取り出して引き渡す。一般に，ある記憶装置内の利用頻度の高いデータの複写を記憶し，より高速にこれを供給する記憶装置を《キャッシュ（cache)》と称するが，ここに述べる UNIX のバッファは，キャッシュとしても機能するものであり，バッファキャッシュとも呼ばれている。書込みに関しても，システムがあるブロックに対して最初に書込みを行う場合，バッファを一つ割り当て，そこにデータの複写を行うだけで実際のディスクへの書込みは行わない。

各バッファは，《見出し（header)》と実際にファイルデータの複写を収容するデータ配列とから成る。データ配列の容量は，4.3BSD においては，最大ブロックの容量に等しい8192バイトであるが，データ配列は仮想メモリ上に配置されており，これに対する物理メモリの割付けはシステムのバッファ制御プログラムが行う。この制御を容易にするため，データ配列は，常時物理メモリ上にある見出しとはかけ離れたアドレス領域に置かれ，見出し内のポインタによって見出しと論理的に結ばれている。見出しは，複写元のファイルが記録されているディスクの装置番号，物理ブロック番号，割り当てられている物理メモリバイト数，バッファ内データのバイト数などを含んでいる。（図6.1）

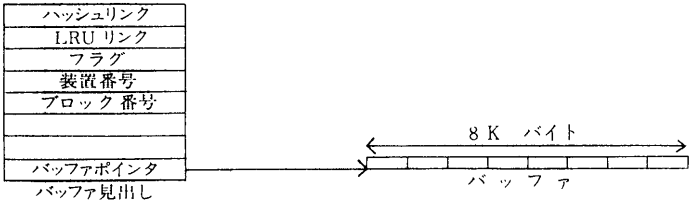

図6.1 バ ッ フ ァ

システムがファイル・システムのデータブロックを必要とするときは，まずはじめにその複写をもつバッファの有無を検索するが，この検索を容易にするため，有効な複写内容を持った全てのバッファは，ハッシュ連鎖として編成されている。さらに，これらのバッファは，最近利用されたバッファほど再度利用される確率が高いという前提のもとに，利用された時刻順に繋がれている連鎖形リスト（linked list）にも組み込まれている。システムは，利用が終わったバッファをこのリストの最後尾に繋いで戻し，新ファイルブロックのためにバッファが必要になると，このリストの最前部のバッファをそれに当てる。キャッシュ機能は，順次処理の対象となるファイルについては必要性がなく，むしろ先行読出しが有効であるが，UNIX のバッファキャッシュは，この点も考慮して設計されている。

上述のリストは，前方のバッファから成る AGE リストと後方のバッファから成る LRU (Least Recently Used) リストとに二分されており，純然たるキャッシュとして機能するのは LRU リストの方である。システムは，利用が終わったバッファを，通常は LRU リストの後尾に戻すが，読出しあるいは書込みがバッファの末尾まで達してその利用が終わった場合には，AGE リストの後尾に戻す。また，プロセスからのファイル読出し要求が論理的に連続したブロックについてあった場合，システムは順次処理が行われていると判断して先行読出しを開始し，これによってディスクからのブロック転送を受けたバッファは最初の読出し要求があるまで AGE リストに繋がれている。

プロセスの書込みは，外見上はディスク上のファイルブロックに行われるものであるが，実際には，書込み要求があった時点で行われるのは，データのバッファ転写だけである。バッファに対する最初のデータ転写が行われると，ディスクブロックの内容の有効性が失われたことを示すため，バッファ見出し内のあるフラッグがオンにされるが，このようなバッファを《ダーティ（dirty）バッファ》という。ダーティバッファは，その内容がディスクに転送されるとダーティであることを示すフラッグがオフに戻され，AGE リストの最前部に置かれる。AGE リストに対しては，バッファがその最前部と最後尾のどちらにも戻されることがある。(図6.2)

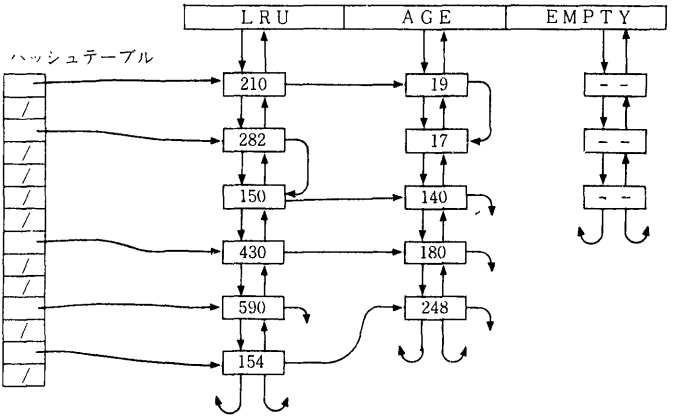

図6.2　バッファキャッシュ

システム内に設けられるバッファの個数はシステム生成時に定められ，システムの運転開始時に各バッファは2048バイトずつの物理メモリが割り当てられて AGE リストに繋がれる。システムが新バッファを必要とする時点では，AGE リストの最前部のバッファが取り出されるが，ここにバッファが無い場合には LRU リストの最前部のバッファが取り出されて使用される。したがって，この場合には AGE リストと LRU リストとは連結された一個のリストと見なすことができる。取り出されたバッファについて

は，つぎに物理メモリを必要量具えているかどうかが調べられ，必要に応じて物理メモリの追加または削除が最小フラグメント長に等しい512バイト単位で行われる。

取り出されたバッファが余剰物理メモリを持っている場合には，物理メモリを全く持たないバッファから成る空バッファリストのバッファを一つ取り出してそれに余剰物理メモリを付加し，AGE リスト最前部に移す。このとき，もし空バッファリストにバッファが無ければ，余剰物理メモリはそのままにしておく。

取り出されたバッファの物理メモリが不足である場合には，AGE リストの最前部からさらにもう一つバッファが取り出され，この物理メモリが必要量だけ利用される。この結果，AGE リストから二度目に取り出されたバッファに物理メモリが残っているかどうかで，それは AGE リストあるいは空バッファリストへ返却される。もちろん，これでなお新バッファの物理メモリに不足があれば，必要な全量が確保されるまで上の操作が繰り返される。

UNIX のバッファキャッシュは，ディスク入出力頻度を低く抑えるという目的を効果的に果たしており，ある典型的なサイトにおいて，読み書き要求の85パーセントがディスク入出力を必要としないという計測結果が報告されている。一方，バッファキャッシュが必然的に抱える問題点としては，プロセスからバッファへのデータ転写とバッファ内容のディスク転送との時間的ずれである。これは通常何ら問題は無いのであるが，不時の機械障害発生の際，ダーティバッファの内容が消失する可能性が高い。このため，すべてのダーティバッファをディスクへ強制的に転送するシステムコールが30秒ごとに自動的に実行され，また，一般にその消失に伴う被害の大きいディレクトリについては，バッファ上で内容変更があると即時にディスク転送が行われ，データ消失の危険回避が図られている。

参 考 文 献

[1] J. S. Quarterman, A. Silberschatz, & J. L. Peterson, "4.2BSD and 4.3BSD as Examples of the UNIX System", *ACM Computing Surveys* **17** (4), pp. 379-418 (December 1985).

[2] D. M. Ritchie & K. Thompson, "The UNIX Time-Sharing System", *Bell System Technical Journal* **57** (6 Part 2), pp. 1905-1929 (July-August 1978).

[3] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1986).

[4] D. M. Ritchie, "The Evolution of the UNIX Time-Sharing System", *AT&T Bell Labortories Technical Journal* **63** (8), pp. 1577-1593 (October 1984).

[5] S. J. Leffler, M. K. McKusick, M. J. Karels & J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley Publishing Company, Inc., Reading, MA (1989).*

[6] A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ (1986).

[7] K. Thompson, "UNIX Implementation", *Bell System Technical Journal* **57** (6 Part 2), (July—August 1978).

[8] G. Anderson & P. Anderson, *The UNIX C Shell Field Guide*, Prentice-Hall, Englewood Cliffs, NJ (1986).

[9] R. Morgan & H. McGilton, *Introducing UNIX System V*, McGraw-Hill Book Company, NY (1987).

[10] M. G. Sobell, *A Practical Guide to the UNIX System*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA (1989).

[11] B. W. Kernighan & P. J. Plauger (木村 泉訳), ソフトウェア作法, 共立出版, (1981).