■7 群（コンピュータ -ソフトウェア）－3 編（オペレーティングシステム）

3 章 ファイル管理

（執筆者：吉澤康文）［2013 年 2 月 受領］

■概要■

一般的に，コンピュータと情報交換する対象はすべてファイルと呼ぶが，ここでは，情報を系統的に管理する OS の基本機能をファイル管理としている．前章では物理的な入出力装置との接点を説明したが，この機能の上位にファイル管理は位置していることを説明する．どの OS にも独自のファイル管理（ファイルシステムともいう）が存在するが，共通した概念，基本的に備えなければならない機能を UNIX をケーススタディとして説明する．

【本章の構成】

序論で説明したとおり，OS も工業製品である．OS はいくつかの機能要素からできているが，その各々の設計の基本は，使い勝手の良さ，性能保証，そして信頼性の提供にある．ファイル管理もこの三つの基本的な考えは同じであることを説明する（3-1 節）．次に，入出力装置は一般的に情報アクセスが低速のため，基本的な性能保証技法を OS は備えている必要がある．ここでは基本的な三つの技法を解説する（3-2 節）が，この技法はアプリケーションプログラムの開発にも応用できるはずである．そして最後に，プログラマがプログラム開発を容易にする OS のファイルアクセス機能とファイルの信頼性向上機能について説明する（3-3 節）．

■7 群 － 3 編 － 3 章

3-1 基本的な考え方

（執筆者：吉澤康文）［2013 年 2 月 受領］

3-1-1 OS における入出力機能

ファイルシステムをもたない OS はない．逆に言えばファイルをサポートしていない OS は制御プログラム（CP : Control Program）として区別されることもある．そして，どんな OS でも独自のファイルシステムを保有している．近年では，OS のファイル管理の方法やデータ形式などが公開（オープン）されている場合があり，ほかの OS のファイルをサポートするケースもある．

2 章で述べたように，OS の発展は入出力装置のプログラミングを共通化することからはじまっている．どんなプログラムでも入出力は必要になる．したがって，多くのプログラマが必要とする入出力プログラムを提供することから OS は発展したのである．

例えば，プログラムで計算を行う場合，計算結果を必ずディスプレイやプリンタに出力する必要がある．現在のプログラミング環境では標準入出力ライブラリのようなソフトウェアが用意されているために，ほとんどのプログラマは容易に計算結果を出力できるようになっている．

コンピュータが計算を中心に利用されていた時代でも，計算すべきデータやその結果などが大量に発生すると，それらを蓄積する必要が生まれていた．また，コンピュータを利用する人たちがそれらのデータを相互に交換するようになると，記録媒体上に格納し共通的に扱えるデータ形式（Data Format）が必要になってくる．初期の頃，データ蓄積に磁気テープが利用されたため，その標準化が進んだのもここに理由がある．

3-1-2 装置独立をめざして

OS の発展は，入出力装置に対するプログラムの共通化，そして，データ格納形式の共通化へと進んできた．ここで説明するファイルシステム（管理）はコンピュータ内でのデータ格納方法の発展形である．2-3-4 項の図 2･8 には入出力に関する現代の OS が備えるソフトウェア階層を示す. 物理的な入出力装置を直接操作するプログラムはデバイスドライバ（Device Driver）と呼ばれ，装置の管理を行う．一方，ファイル管理ではファイルシステムに格納されたデータをプログラマに対して抽象化したオブジェクト（Object）として提供し，そのアクセス法（Access Method）を用意している．

ファイル管理はプログラマにファイルを論理的なデータの集合として提供する．つまり，ファイルが格納されている物理的な媒体を意識させることのないようにするという目的がある．つまり，以下の目的をファイルシステムは達成しなくてはならない．

(1) 記録媒体は年々進歩するが，プログラマに一切負担かけることなくその恩恵に浴することができるようにすること．

(2) 記録媒体の容量を無駄にすることなく利用すること．

(3) 記録媒体に対するアクセス速度を短くするような性能向上機能を提供すること．

(4) 貴重なデータを取扱うために，信頼性向上を図ること．

(5) 装置に依存しない論理的なインタフェースをプログラマに提供すること．

このうち，(1)はファイルを装置から独立にすることを目指した機能であり，プログラマは記録媒体が変わってもプログラムを変更することなく従来のプログラムでファイルをアクセスできるようになる．新しい装置（これを周辺機器：Peripheral Equipment とも呼ぶ）に対する変更ソフトウェアは OS のデバイスドライバになされるので，ユーザプログラムは変更を必要としないのである．

上記, ファイルシステムにおいて重要な点(2)から(4)の各項目については 3-2 節で説明する．そして(5)はプログラマに対するインタフェースを規定することであり，具体例として UNIX のインタフェースを一つの具体例として 3-4 節で説明する．

3-1-3 広義のファイル

現代の OS ではファイルの意味を広くとらえていて,「ファイルはコンピュータに対する入出力のすべてである」と考えている．コンピュータ内の OS の立場からすると，情報の入出力のすべてをファイルと見た方が統一して操作可能となるという利点がある．その意味で，端末に座ってコンピュータを利用している人間も入出力可能なファイルであり，キーボードやマウスなどのポインティングデバイスからの入力行動をしたり，ディスプレイへの出力を見て次の指令を入力したりしている．つまり，キーボード，ディスプレイもファイルとして扱っているのである．

■7 群 － 3 編 － 3 章

3-2 ファイル操作の基本技術

（執筆者：吉澤康文）［2013 年 2 月 受領］

3-2-1 ブロッキング

(1) レコードの概念

一般的にプログラマはファイルを作成する際にデータの論理的な構造を考える．例えば，従業員を管理するファイルを作成するならば，**図 3･1** に示すようなデータ構造を考えることになる．このような論理的なデータの集まりをレコード（Record）と呼ぶ．

```
struct  employee {
        char   name[30];
        int       sex;
        int       age;
        char      address[80];
     .
     .
}
```

図 3・1　論理的なデータ構造の例

ファイル処理における一つの問題点は処理時間が多くかかるということである．この処理時間の短縮の解決法の一つに，入出力実行回数を少なくする方法がある．図 3･1 に示すような短いレコードを一つずつ磁気ディスクなどに書き込んでいたのでは処理時間は短くならない．そこで複数のレコードをまとめて 1 回の入出力操作とすると入出力回数が減り時間短縮になる．このように，複数のレコードを一つにまとめて入出力することをブロッキング（Blocking）と呼び，記録媒体の有効利用として古くから行われている．

図 3･2 にブロッキングの様子を示した．一つのブロックに格納するレコード数をブロッキングファクタ（Blocking Factor）と呼ぶ．図 3･2 の例では，四つのレコードを一つのブロックにしているので，ブロッキングファクタは 4 である．このようにブロッキングすることで，一つのレコードをアクセスするオーバヘッドは，毎回システムコールを発行し，入出力を毎回実行する方法に比べて数分の 1 になる．

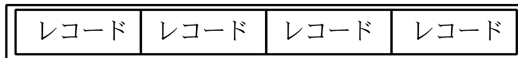

図 3・2　入出力の単位とするブロッキング

(2) ブロッキングの考え

磁気テープなどはファイルのバックアップ（Backup）として利用される．**図 3･3** に示すように，磁気テープはテープ走行制御のために，データを書き込むブロックとブロックの間に IBG（Inter Block Gap）あるいは IRG（Inter Record Gap）なる領域を必要としている．このサイズは一定であるため，データ書込みのブロックサイズを小さくすると実効的な磁気テープ容量が小さくなってしまう．

 © 電子情報通信学会 2013

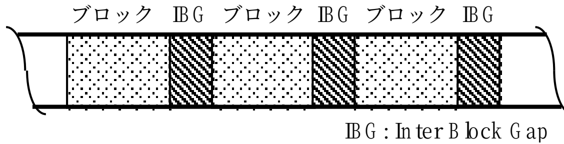

図3・3　磁気テープ上のブロック配列

今，ブロックサイズをBLKSZとし，IBGサイズをIBGSZとする．このとき，磁気テープに格納される実効的なデータの割合をE(BLKSZ) とすると，E(BLKSZ) = BLKSZ/(BLKSZ + IBGSZ) となる．このことから，実効的なブロック容量を計算して，バックアップを行う必要が生まれる．

(3)　性能に関する考察

ソフトウェアの設計において性能保障は重要なことである．いくら機能が優れていても性能が悪くては使いものにならない．設計の初心者がおかしやすい性能上の誤りは入出力にある．上記に述べたブロッキングは重要なポイントであり，ブロッキングすることで一つのレコードに対する入出力実行回数とCPUオーバヘッドを大幅に少なくすることが可能である．この場合，考慮すべき設計のポイントは以下のとおりである．

(a) ブロッキングファクタをいくつにすべきか.

(b) 入出力時間は設計基準を満たすか.

(c) 入出力領域（バッファ）のサイズは確保できるか.

(d) CPUオーバヘッドはいくらになるのか.

ブロッキングファクタは大きいほど性能効果が高いことは理解できるが，大きくすると上記の(b)，(c)の項目が問題となってくる．つまり，大きなブロック入出力を行うと，1回の入出力時間が長くなるため，ブロックの先頭にあるレコードを読み込んだときのアクセス時間がそれ以降のレコードのアクセスに比べて長くなること，そして，(c)に関する問題は，ブロックを大きくすることで必然的に大きな入出力領域（Buffer）が必要になるということである．これらの点が問題とならないか検討する必要がある．この種の問題はコンピュータの資源消費のトレードオフ（Trade-off）の問題であり，CPUと入出力に費やす時間を大きなメモリ容量で置き換える価値があるか否かの選択ということになる．

3-2-2 アクセスギャップを埋めるバッファリング

ファイルアクセスの代表的な装置として磁気ディスク（ハードディスク）が一般的である．コンピュータの2次記憶としては比較的アクセスが高速であり，ビット単価も低いため利用度は高い．しかし，CPUからみると主記憶へのアクセス時間に比べて磁気ディスクへのアクセスは桁違いに遅いことになる．この両者のアクセス時間の大きなギャップは古くから問題になっている．

そこで，入出力を少しでも速くアクセスする技術が開発されてきたわけである．ここに説明するバッファリング（Buffering）もその一つである．バッファリングを全く行わない場合のファイル操作は**図3･4**に示すとおりである．ここでの処理は，ファイルを1ブロック読み

込み，その処理を行う繰り返しとする.

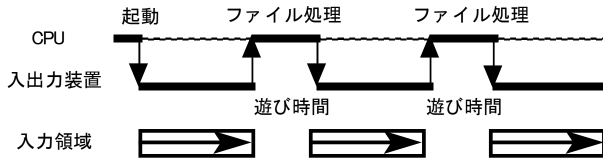

図 3・4　バッファリングを行わない入出力方式

(a) CPU から入出力装置に読込みの起動を行う.

(b) 入力領域にデータが入る.

(c) CPU が入力領域内のファイル処理を行う.

(d) ファイルが完了するまで上記(a)～(c)を繰り返す.

図 3･4 の操作を繰り返すことは CPU と入力処理を逐次に行うことである. この図からも明らかなように，低速な入出力装置に遊び時間が生じている.

そこで，入力領域を二つ用意し，一つ目の入力領域に入力した情報を CPU が処理している間に，もう一つの入力領域に入力動作を行わせる．つまり，CPU と入出力装置の動作を重複させる（オーバラップ：Overlap）のである．**図 3･5** にその様子を示す．ここでは，2 面の入力領域 A，B を設け，入力が完了すると，直ちにもう一面の空き状態になっている領域に読込みを実施する．この結果，低速な入出力装置は全く遊び時間がなくなり性能を最大限引出すことが可能になる.

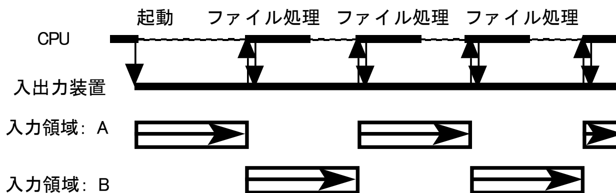

図 3・5　バッファリングを行った入出力方式

バッファリングとは本来「緩衝」という意味である．これは，CPU のメモリアクセスと入出力のアクセスギャップをやわらげるということから派生している．現在，ファイル入出力を行う領域をバッファ領域と呼ぶことがあるが，本来のバッファという意味は，ここで説明した「入出力アクセスのギャップを埋める緩衝領域」という意味である.

3-2-3 キャッシング

バッファリングを行っても現代の高速な CPU の入出力アクセスギャップはゼロにはならない．アクセスギャップの究極の目的はギャップをゼロにすることである．1980 年代後半から大容量記憶を利用できるようになり，このアクセスギャップの解決が可能になった.

アクセスギャップをゼロにするということは，ファイルアクセス時間が主記憶アクセス時間と同じになるということである．このためには，ファイルと同一容量の入出力領域を主記

憶上に確保する必要がある．しかし，このアプローチには以下の問題がある．

(1) 高速で安価な半導体メモリは揮発性（Volatile）であり，電源を切ると内容が保存されない．

(2) 大容量主記憶が利用可能となってもコンピュータシステム内の全ファイルを保管するほどの容量は経済的に不可能である．

このような状況からすると，ファイルのよく利用される部分だけを主記憶に保存し，その部分に対するアクセス要求があったときはアクセスギャップをゼロにするという工夫が妥当である．つまり，バッファリングのように入出力領域を数倍程度増やすのではなく，一度アクセスしたブロックを入出力領域に格納しておき，再度アクセスが生じたときは入出力領域から該当するブロックをユーザの入出力領域にコピーする方法である．

このような方法を一般的にキャッシング（Caching）と呼び，その領域をキャッシュ（Cache）という．**図3･6** にその様子を示す．もし，あるファイルの特定のブロックに対するアクセスが頻度高く発生するならば，キャッシュの効果は高くなる．そのような部分をホットスポットと呼ぶ．もし，読込み専用のファイルにおいて，容量が小さくホットスポットとなるようなファイルがあるならば，このアプローチは性能向上効果を高くする．そこで，ファイルアクセスの特性を分析し，頻度高くアクセスされる領域だけをキャッシュの対象とすると効果的なことは明らかである．

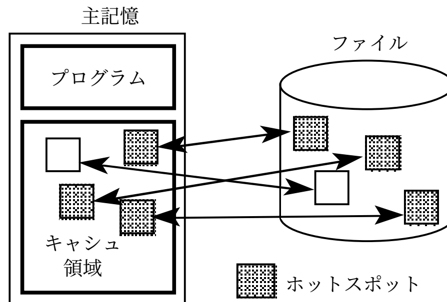

図3・6　ファイルのキャッシュによる高性能化

キャッシュを用いた場合の問題点は，主記憶上の情報と磁気ディスクなどの記録媒体との情報の一致性にある．読込み専用のファイルについては問題ないが，書込みを伴うアクセスの際に問題が生じる．つまり，ファイルブロックへの書込み操作を行った時点で毎回記録媒体上に書込みを行っているのではキャッシュの効果がなくなる．したがって，ある時点でファイル更新の結果を記録媒体に書き込む必要がある．例えば，オンラインシステムなどのトランザクション処理では，一つのトランザクション（取引）が完結した時点でファイルへの書込み操作を行う，などということがよく行われる．

キャッシュのもう一つの問題点は，キャッシュ内のブロック領域が不足したときにどのブロックを解放するかということである．この問題は，仮想記憶におけるリプレースメントアルゴリズムと類似している．両者には「将来使用される見込みのないブロックを解放する」という共通の考え方がある．しかし，将来のファイルブロック参照の予測は難しい．そこで，

「最近使用されたブロックは近い将来再度参照されることがある」という仮説のもとに，LRU（Least Recently Used）が使用されることがある（LRU については，仮想記憶制御方式の章で説明する）．

入出力のキャッシングをカーネルが自動的に行っているのが UNIX である．UNIX では，一般的なファイルアクセスを行っている場合はキャッシュに情報が存在するので，ファイルの書込み時に信頼度を確保するには，システムコール sync( ) を実行する必要がある．

3-2-4 信頼性の確保

(1)　ニーズ

現代のコンピュータは計算機械というよりむしろデータ処理に利用されている．したがって，データの保存，維持，管理は重要な機能となっている．ファイルの信頼性を低下させる要因はハードウェア，ソフトウェア両方に存在する．ファイルの信頼性を向上するためにはいろいろなアプローチがあるが，ここでは基本的な機能のみを説明する．

(2)　ファイルになぜ矛盾が生じるのか

ファイルに障害が生じるのは，多くの場合コンピュータの不意のダウンに起因する．あるファイルに書込み中の中断，あるいは，上記のキャッシュ上に存在するデータがファイルに反映されていない場合などである．一般的にファイルシステムは，ファイルを管理するディレクトリ（Directory）やファイルに記録媒体上の領域を割り付けている管理テーブル，そして，ファイルの実体からできている．これらの情報の間には相互に関連付けがあり，ファイルシステムが一連の操作を行っている間に処理が中断すると，これらのデータ構造内で矛盾を引き起こすことになる．**図 3･7** にその様子を示す．ここでは，ファイルの格納媒体を磁気ディスクとする．

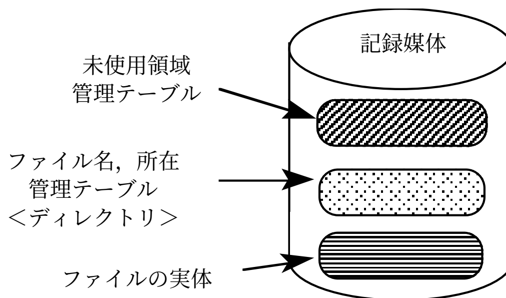

図 3・7　記録媒体上のファイル管理情報

一つの磁気ディスクのことをボリューム（Volume）と呼ぶことがある．一般的に，ボリューム上に格納されているファイル関連の情報は，一つのボリュームで自己完結するように作られている．この拡張形態として，複数のボリュームを一つの論理的な媒体（Logical Volume）として扱う技術も存在するが，ここでは，説明の都合上一つのボリュームを前提とする．つまり，記録媒体上に存在するファイルのすべての名称ならびにファイルの実体の所在などを

示すディレクトリ，記録媒体の未使用領域の管理テーブル，そしてファイルの実体などが一体となって一つのボリューム上に存在するのである.

(3) ファイル管理情報の更新順序

ファイル管理はこのように複数の論理的ならびに物理的に分割された情報により管理されている．したがって，ファイルの更新を行う際には，これら 3 種類の関連付けられた情報を同時に更新する必要があるが，複数回の入出力を伴うことになる．この過程で何らかの原因により処理の中断が生じると，ファイル更新が正常に終了しないのは明らかである．ファイル管理では，システムがどの処理段階でダウンしたとしてもファイルシステムに矛盾が生じないように，3 種類の情報の更新順序に配慮がなされている.

このような観点から，図 3･7 に示した 3 種類の管理テーブルの更新順序は以下のようになる. 大切なことは, ディレクトリのような管理テーブルは最後に書き込むことである. また, ファイルの実体を最初に書いても問題はない.

(a) 未使用領域管理テーブル
(b) ファイルの実体
(c) ファイル名，所在管理テーブル

(4) 矛盾の発生と波及

ファイル処理の中断は避けねばならないが，システムダウンが避けがたいことであるならば，フェイルセーフな設計にしておく必要がある．つまり，システム設計では最悪の状況を考えておかねばならないということである．ここで問題となるのは，複数のファイルが同一のファイル実体を示してしまうことである．つまり，自分のファイルを参照したとき，他人のファイルの内容が参照されるようになってしまうことは絶対に避けねばならない.

また，上記の事態が生じたときに，ユーザによっては他人のファイルと自分のファイルが混在していることを発見し，そのファイルを消去してしまうことがあるかも知れない．その場合，最悪のケースとして考えられるのは，他人のファイルの一部を消去してしまうことである．**図** 3･8 にはそのケースを示す.

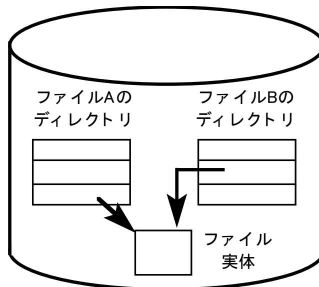

図 3・8　同一ファイル実体をポイントする矛盾

・ファイル A と B が同一のファイル実体を示している.
・ファイル A を消去してしまう.

・ファイル実体の領域が空き状態になる．
・新たなファイルを作成したとき，上記の空き領域が割り当てられる．
・この結果，新たなファイル作成によりファイル B の実体が消去される．
・新ファイルとファイル B との間に再び 2 重保有状態が生じる．

このような状況が生じると，誤りが波及していくことになる．もっと恐ろしい事態は，未使用領域管理テーブルの矛盾にある．仮に，未使用領域管理テーブルを最後に更新するようなアルゴリズムでファイル更新処理がなされていたときに，途中でシステムダウンに遭遇するとどのような結果となるか想像してみよう．このとき，ファイルシステムとしては空きの領域にファイルが作成された状態になる．このため，リブート後，ファイルをユーザが作成したり更新したり作業が進むに従って該当ファイルを参照すると，ほかのファイルの内容を参照するようになる．仮に，このファイルを消去すると，ほかのファイルの一部，あるいは全部を消去してしまうことになる．

(5) フェイルセーフな処理

上記の誤りは深刻で，場合によると複数のファイルに影響を与える．このためファイル管理では，いつコンピュータがダウンしてもファイルシステムとして矛盾が生じないこと，ならびに誤りが波及しない方法をとる必要がある．しかし，コンピュータがダウンした場合に，完全に問題を排除できるとは限らない．少なくとも，処理途中のファイルには最新の情報が格納されたかどうか不明である．

システムプログラムにとって大切なのは，コンピュータがいつダウンしてもその影響を残さないように設計しておかなければならないということである．つまり，ダウン時にフェイルセーフに処理がなされるように処置されてなければならない．上記（3）に示したような順に書込み操作すると，場合によるとボリューム内のある領域は使用中の状態になっているがどのファイルにも割り付けられていないかも知れない．しかし，たとえ無駄な領域を作ったとしてもファイル間に矛盾を生じ，その後誤りが波及してしまうよりはましである．

(6) ユーティリティソフト

フェイルセーフな設計を行い，ダウンに備えることは重要である．あとは，ファイルシステムの保守の問題に帰着する．そこで，定期的，あるいは非定期にファイルシステムの診断（Diagnostic）を行う必要がある．そのため，ファイルシステムに関しては各種のファイルシステム支援のプログラムがメーカ（Vendor）から提供されるか，あるいはユーザが目的に応じて開発するプログラムが存在する．これらは，一般的にユーティリティ（Utility）と呼ばれるプログラムである．

(7) 多重ファイル方式

信頼性を確保するためには冗長度（Redundant）をもたせる方法が頻繁に利用される．つまり，同一機能をもつ装置を複数備えておき，一方がダウンしたときに代替え装置に切り替えて処理を続行する方法である．ファイルシステムの信頼度確保にも同様の方法が利用される．その代表的な方法がファイルミラーリング（File Mirroring）やファイルデュプレックス（File Duplex）である．

これらの方法は一般に多重化ファイルと呼ばれるもので，高信頼化を要求される企業情報システムなどに実用化されている．**図 3･9** に両方式を示す．ファイルミラーリングでは，一つの入出力制御機構に接続されたボリューム上に同一のファイルを 2 重に保有し，書込みは両ファイルに対して行う．ここでは，正副ファイルと呼ぶことにするが，正常状態での読出しは正ファイルから行う．仮に，正ファイルへの障害が発生したときは，副ファイルで運転することにする．

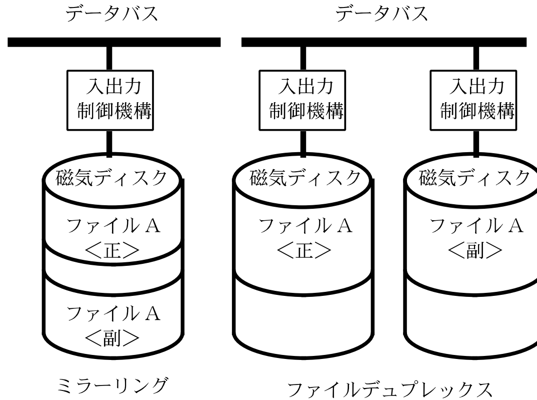

図 3・9　多重ファイルによる信頼性向上

デュプレックス方式はハードウェア的に 2 系統を保有する．操作はミラーリング方式と同一である．この方法は，入出力系統を 2 重化しているためミラーリングに比べて性能ならびに信頼性が高くなる．両方式とも，2 倍のファイル容量を必要とすることになる．このように，信頼性を向上させるには一般的に処理時間やコストを犠牲にすることになるが，情報を企業活動の基幹としている産業ではコストより信頼性に力を注ぐべきであろう．

(8)　バックアップ

ファイルの信頼性確保のために最も単純で効果的なことはバックアップ（Backup）をとることである．ファイルの重要度に応じてバックアップを行うのが常套手段である．磁気ディスク上のファイルは，磁気テープなどにバックアップコピーされることが多い．この作業には人手を要する場合もあるが，近年では自動化されたハードウェア機構ならびにソフトウェアが実用化されている．

バックアップにも各種の方法がある．パソコン，ワークステーションあるいは小規模のサーバなどで使用するコンピュータでは定期的なバックアップを行えばよいかも知れない．しかし, コンピュータを 365 日 24 時間運転しているようなコンピュータシステムではバックアップ作業をオフライン（Off Line）で行うことは不可能である．したがって，そのような場合はシステムが運用されている間にバックアップを行うオンラインバックアップという方法がとられている．

(9) ジャーナル

OLTP などでは信頼性に対する要求が厳しい．システム停止から再開までの時間は秒のオーダでなされなければならないし，システムが再開始されたときに，ファイルの内容は最新の状態でなければならない．

そのためには，オンラインバックアップだけでは不十分であるので，ファイル更新を行うごとに変更履歴を取得し，これを各トランザクション単位に管理した情報としてとっておかねばならない．このようなファイル更新情報や端末へのメッセージなどの記録採取をジャーナル（Journal）と呼ぶ．

OLTP ではジャーナルを取得しておき，システムダウン直後の再開処理において最新のファイルを構築する．この方法により継続的なオンラインサービスが実施できるのである．

■7 群 － 3 編 － 3 章

3-3 ファイルシステムの概念

(執筆者：吉澤康文)［2013 年 2 月 受領］

3-3-1 論理化したインタフェース

(1) はじめに

ファイル管理が果たすべき役割は次のとおりである.

(a) 名前を付けたディレクトリやファイルが作成できる.

(b) ファイルに情報を格納する.

(c) ファイルに対して読込み，情報の追加，更新ができる.

(d) ファイルやディレクトリを消去する.

(e) ファイルアクセスの保護機構を提供する（共用，排他など).

これらはファイルという論理的な情報の集合に対する操作であり，物理的な装置，記録媒体を全く意識させることはない.正確なインタフェースは 6 章に述べることにし,ここでは,各機能のインタフェースとして UNIX の例を中心に概要を説明する.

(2) 名前の管理：UNIX の木構造

ファイルを理解しやすい形式で，かつ，ユニークに識別できるようにするには，ユーザが任意に名前を付けられる方法が望ましい．そしてシステムが管理するファイル，企業活動に必要なファイル，個人の保有するファイルなどが統一的に扱えると理解しやすい．このような要求から，木構造（Tree Structure）のファイル名構造が考案された.

図 3･10 には UNIX における例を示す．UNIX では，木構造の原点になる部分をルートディレクトリ（Root Directory）と呼んでいる．この図は一部を例示しているだけであるが，原点から別れる木により一般ユーザファイルを格納する部分（/home），ユーザ固有のファイルを格納する部分（/usr）などに分類することができる．現代の OS では，UNIX と同様な木構造のファイル名管理機能を提供しているものが多くなっている.

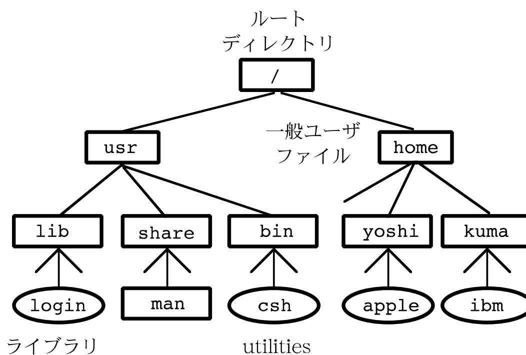

図 3・10 UNIX における木構造のファイル名空間

(3) ファイル名とパス名

UNIX では，図 3･10 に示す usr，home，lib，share などをディレクトリファイル（Directory

File）と呼んでいる．そして，usr の下にある lib，share，bin などをファイル名（File Name）と呼ぶ．したがって，lib，share，bin などはディレクトリであり，またファイルでもある．つまり，上記に述べたように UNIX では，ディレクトリ自身もファイルなのである．

UNIX の例では, ’/’ より始まるファイル指定（例 : /home/yoshi/apple）を絶対パス名（Absolute Path Name）と呼び, ’ /’ のないファイル指定（例 : apple）を相対パス名（Relative Path Name）と呼ぶ．このように，あるディレクトリの下に付けられた名前をユニークにしておけば，ファイルシステム内でユニークなファイル名を特定することができる．

UNIX でファイル名に使用できない文字は，スラッシュ’ /’ とスペースである．’ /’ はディレクトリを示すので使用できない．また，スペースはディレクトリ内にファイル名を格納する際にファイル名の終端を区別するために必要な場合があり, 使用できない. 初期の UNIX ではファイル名が 14 文字以内であったが，BSD 系 UNIX では 255 文字まで許されている．

(4) ファイル属性

ファイルはコンピュータで使用するすべての情報格納機構であり，基本的な機能である．ソースプログラム，オブジェクトプログラム，書類，図，表，音声，動画像情報など，すべての情報が統一したファイルとして格納されている．もちろん，オペレーティングシステム自身も一つのファイルとして格納されている．

小さなサーバや多くのワークステーション，パソコンなどでは，ほとんどの場合一つの記録媒体上に各種のファイルが混在して存在しているので，ファイルを参照するプログラムからの保護属性，ファイル属性などを管理する必要がでてくる．

例えば，ファイルの所有者，作成年月日，実行可能か否か，書込み可能か否か，アクセスを許可するプログラムやユーザの区別などが代表的なファイル属性（File Attribute）としてあげられる．**表 3･1** にこれらの属性を区分してまとめた．この表に示すように，属性は大きく分けると三つになる．

表 3・1　ファイル属性

ファイル本来の性格	読出し(read)，書込み(write)，実行(executable)の組合せを指定する．
アクセス権限	所有者以外の者にアクセスを許可する範囲を指定．グループ内，権限を放棄するなど．
その他ファイル情報	作成者，作成年月日，最終更新年月日，ファイルサイズ，ファイル実体の所在マップ，など

ファイル「本来の性格」は，読出し，書込み，実行可能を許可するか否かの組合せであるので，8 とおりの属性がある．これらの組合せの中には「すべて禁止」のように意味のない属性もあるので，実際の組合せは少ない．「本来の性格」はアクセス権限と組み合わせることにより有益なものになる．例えば，所有者は読出しと書込みは許すようにするが，グループのユーザに対しては読出しのみを許す，などがよくある例である．

これらの属性は，次に述べるファイル作成時のパラメータとして与えられる．

(5) ディレクトリ，ファイル作成

ファイル管理では，任意の名前を付けディレクトリや一般のファイルを作成する機能が必要である．そこで，いろいろなシステムコールが用意されている．UNIX の例では，一般のファイルの作成には creat( )，open( ) など，またディレクトリの作成は，mkdir( )，mknod( ) などのシステムコールが用意されている．

ファイルの作成にあたっては，ファイルに属性を与える必要がある．ファイル属性は表 3･1 に示したように，ファイル本来の性格を示す情報とアクセス権限の範囲を限定する情報の 2 項目からなる．UNIX では，ファイル生成時にこの両者を組み合わせたファイル属性を与える．表 3･1 にあるそのほかの情報はプログラマが指定しなくても自動的にファイル管理で作成される．

(6) ファイルのオープン

作成したファイルに対して情報を読み出したり，情報の更新を行ったり，追加したりできなければならない．このためには，我々が本を読むときに，最初ページを開くように，ファイルを開かねばならない．この操作をオープン（Open）と呼び，UNIX では open( ) というシステムコールが用意されている．

一般に，オープンでは，ファイル名を指定するばかりでなく，そのファイルをどのようにプログラム内で使用するか宣言する必要がある．それは，ファイルをアクセスするプログラムが，該当するファイルに対してアクセス権を保有しているか否かのチェックを行う必要があるためである．

例えば，読込み専用のファイルに対して，書込み要求のオープンが実行されたときはこれを禁止しなくてはならない．このようなファイル保護属性のチェックが厳しく行われことによりファイルが保全される．したがって，ファイルオープン時に読込み要求とし，後に，そのファイルに対し書込み操作すると，ファイル管理はこれをエラーとして書込み要求をはねのける．

(7) ファイルの読み書き

このように，ファイルのオープン操作が完了すると，ファイル管理はプログラムに対し，オープン完了の情報を渡す．例えば UNIX では，ファイル記述子（Descriptor）を返すことになる．この値は，単なる小さな整数値であるが，ファイルアクセスが認可された証拠（Token）となる．したがって，この後のファイルに対する読み書き操作のシステムコールには，このファイル記述子をトークンとしてファイル管理に示すことになる．

(8) ファイルの消去，ディレクトリの作成など

ファイルやディレクトリが不要になったときそれらを消去する必要がある．UNIX ではコマンド（rm : remove）でファイルを消去する．また Windows や Macintosh OS の GUI 環境ではごみ箱（Trash Box）にファイルアイコンをドラッグすればよい．このような操作をプログラムの中から実行するためには，システムコールが用意されている．UNIX の例では，unlink( ) でファイルを，rmdir( ) でディレクトリを消去できる．

3-3-2 順編成ファイル

ファイルに対するアクセス方式は大きく分けると以下の 2 とおりである．

・順編成

・直接編成

図 3･11 は順編成（Sequential Organization）のファイルのアクセスの様子を示す．順編成のファイルではファイル内の先頭のデータから順にアクセス（Sequential Access）を行うのが原則であり，データが時系列に入っている場合やレコードがある順序に並んでいる場合が多い．例えば，音声・画像情報や従業員レコードなどである．つまり，何らかの形でデータがソート（Sort）されているのである．このような応用例は数多く存在する．

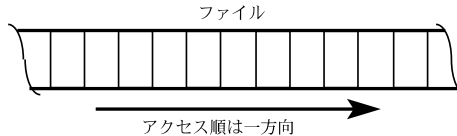

図 3・11　順編成ファイル

磁気テープのような記録媒体上に格納されたファイルは順編成である．順編成ファイルではほとんどの場合，処理を高速に行うためにレコードをあらかじめソートして処理する．**図 3･12** に示す処理は順編成ファイルを用いた典型的な応用例である．IBM 社の MVS では，このような順編成ファイルに対するアクセス法（Access Method）として SAM（Sequential Access Method）をプログラマに提供している．

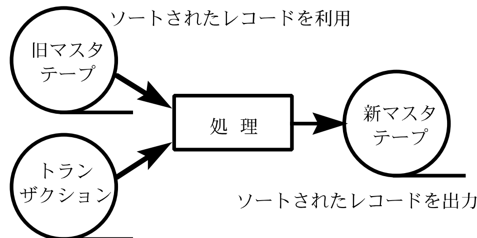

図 3・12　順編成ファイルを使う典型的な応用例

給与計算を例にとってみる．残業時間，出張など日々発生するデータをトランザクションとして一時的にファイルに蓄積しておき，そのトランザクションレコードをあらかじめ処理するキー（Key），例えば，従業員番号などでソートしておく．そして，従業員番号順にソートされているマスタファイル（テープ）に対してトランザクションレコードの情報を基に変更を加え，新マスタテープを作成するのである．

順編成ファイルを利用するケースは，上記の例のように，データを溜込んで，それらを定期的あるいは非定期にマスタとなるファイルと照合して目的の情報を取り出し，新たなファイルを生成する処理である．このような処理をバッチ処理（Batch Processing）と呼び，事務処理の代表的な形態となっている．

3-3-3 直接編成ファイル

一方，直接編成のファイル（Direct Organization）では，順編成とは対照的なアクセス法をとる．図 3･13 に示すように，ファイル内の任意のデータを参照するケースである．このように任意のデータに直接アクセスするためには，ターゲットとするデータのポジションにアクセスアーム（Access Arm）を直接運べる磁気ディスクのような装置にファイルを格納しておく必要がある．IBM 社の MVS では，直接編成のファイルに対するアクセス法として DAM（Direct Access Method）を提供している．

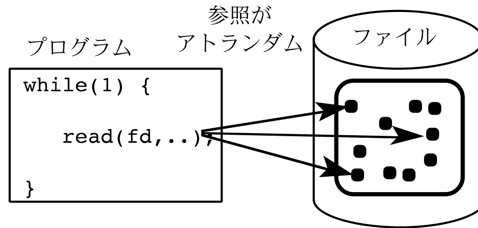

図 3・13　直接編成ファイル

直接編成のファイルの利用はオンラインシステムにおいて数多く見られる．銀行の預金引出しなどは，口座番号を基にしてファイルにアクセスする．この場合は口座番号から該当する口座情報データベース内のレコードにマッピング（Mapping）を行い，ファイル管理をとおして情報を主記憶内に読み込み，更新処理をすることになる．座席予約システム，チケット販売システムなども同様の方法で処理を行っている．

このように，磁気ディスクのような直接アクセス可能な媒体の出現により高速なデータ引出し，更新が可能となった．そして，多くの労働者が定型のルーチンワーク（Routine Work）から解放され，またサービスの質の向上が図られるようになったのである．現代のコンピュータ利用が社会のインフラストラクチャとなっているのは，この技術のおかげである．

3-3-4 ファイルアクセスの高速化

先の節で述べたように入出力のアクセスギャップを埋めることはコンピュータの課題であった．プログラムによるファイルアクセスの動作が予測できるならば OS により処理性能の向上が図れる．メモリ管理の章で触れ述べるように，プログラムのページ参照動作の予測がページリプレースメントアルゴリズムの前提条件であったのと同じである．

このような考え方を基にすると，ファイルアクセスでは意外と予想のつく場合がある．順編成ファイルの読出しなどでは，プログラムは確度高く次のブロックを読む．図 3･12 に示したような応用では，旧マスタテープやトランザクションファイルは最初のブロックから次々と順次読み込むはずである．したがって，プログラムがバッファリングしていようがいまいが，ファイル管理は次の読込み要求がなくても，システムバッファ領域にブロックを次々に先読みしておいても無駄はないはずである．

図 3･14 にその様子を示した．この例では，アプリケーションプログラムが要求しているのはブロック番号 4 であるが，ファイル管理はその先のブロックを次々と読み出してブロック

7 まで読み込んだ状態を示している．このように先読みを行うことはファイル管理が自動的にバッファリングを行っていることになり，ユーザは自分のプログラムによるバッファリングを行うことなく高性能化を達成できることになる．

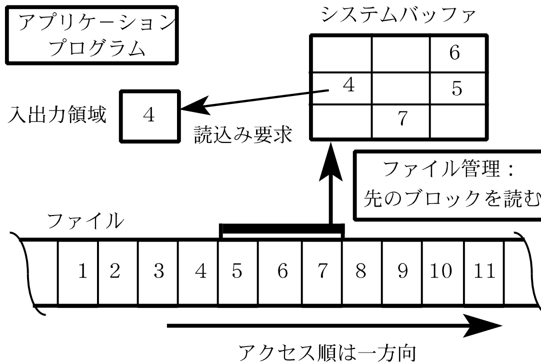

図 3・14　順編成ファイルを先読みする高性能化技術

ただ，この方式ではユーザにとってはどこまで先読みがなされているか不明である．これを解決するには，自分のプログラムによりバッファリングを行う以外に手段はない．また，先読みはシステムバッファの利用状況に依存することになり，安定的ではないという欠点もある．

一方，直接編成のファイルに対する高性能化を行うには次の仮定が必要である．

・アクセス頻度の高い部分がある領域に集中している．

つまり，ファイルアクセスにおいてホットスポットが存在するという仮定である．ランダムにアクセスされるファイルにおいては，すべてのブロックが等しくアクセスされることは少なく，多くの場合，参照が，ある部分に集中するということが知られている．例えば，座席予約システムなどでは,ある特定の日や特定の列車に予約が殺到する現象がよく見られる．また，チケット販売においても，発売初日には予約の列ができやすく，株の取引にも同様の傾向がある．いずれもある特定のブロックがホットスポットになっているはずである．

このようなホットスポットに対しては前節で述べたキャッシングが最も適した解決法になるのであるが，キャッシュの容量と入出力削減効果の予測が難しいという欠点がある．この解決法に一般論はなく，効果を高め，確認するためには特定の利用に限定した十分な解析が必要になる．

3-3-5 同期・非同期入出力

(1)　同期・非同期入出力の違い

ここでは，ユーザプログラムの実行と入出力のタイミングの説明を行う．ファイル管理の提供する入出力のタイミングには 2 とおりの方法が考えられている．つまり，同期入出力と非同期入出力である．

ファイル入出力のシステムコールをアプリケーションプログラムが実行し，カーネルでの

実行が完了して再びアプリケーションプログラムに制御が戻った時点で，要求した入出力が完了している場合を「同期入出力」と呼び，完了していない場合を「非同期入出力」と呼んでいる．同期入出力の代表的な OS が UNIX であり，非同期入出力の代表は IBM 社 MVS である．同期，非同期入出力の機能，利点，欠点を**表 3･2** に示す．

表 3・2　同期・非同期入出力

同期入出力	機能	入出力要求のシステムコールが完了した時点でデータが入出力領域に存在する
	利点	ユーザは同期をＯＳと取る必要ない
	欠点	複数のファイル入出力を同時に起動できない
非同期入出力	機能	入出力要求のシステムコールが完了した時点ではデータが入出力領域に存在しない
	利点	複数のファイル入出力を同時に起動できる
	欠点	ユーザはＯＳと同期を取らねばならない

(2)　同期入出力の特長

同期入出力の利点はプログラマにとってわかりやすいことと，入出力の同期などという面倒なことを行わなくて済む点にある．したがって，プログラムが一つのファイルしか扱わないような小規模な処理を行う場合には全く問題はない．しかし，複数のファイルを同時に操作し，しかも，それらのファイルが別々のボリューム上に配置してあるようなデータ処理においては，致命的な欠点となる．

図 3･15 に同期入出力方式の代表である UNIX を例にして説明する．プログラムからは読込みのシステムコール read( ) が実行される．この結果，制御が UNIX カーネルのファイル管理に渡り，入出力ドライバに入出力要求を出す．磁気ディスクに対して起動かけると数十ミリ秒間を必要とするので，システムコールを発行したプロセスをブロック状態にする．つまり，待ち状態にし，ほかの実行可能なプロセスをディスパッチすることになる．

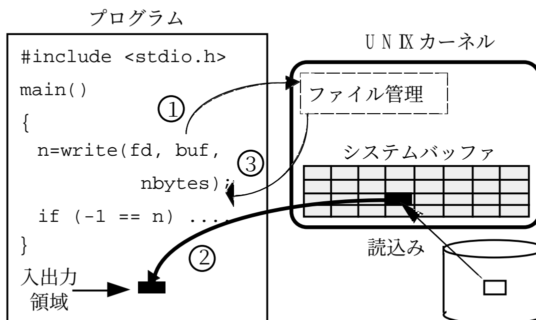

図 3・15　UNIX の同期入出力方式

磁気ディスクからの入力完了により割込みが入り，デバイスドライバに制御が渡る．この結果，ファイル管理の要求が満たされたので，該当ブロックをシステムバッファからユーザ

の入出力領域に転送する．この操作が完了すると read( ) 発行のプロセスの待ち状態が解除され，実行可能状態になり，ディスパッチされる．つまり，プロセスは read( ) システムコールを実行した後に制御が戻った時点で，既にファイルからデータを読み込んだ状態になっている．

このような同期入出力方式は，プログラマにとっては入出力処理が単純で明快であるが，複数のファイルを扱う少し高度な処理の場合には性能面での欠点がある．

例えば，バッファリングをユーザプロセスのレベルで実現することは不可能になる．UNIX ではこの問題を回避するために，システムバッファにより順編成ファイルに対しては先読みを行い，直接編成のためには，システムバッファをキャッシュのように使って，問題の解決を試みている．しかし，高度なデータ処理を実現するときに，この方法では効果が間接的であり正確なシステム性能設計ができないという欠点がある．

図 3･16 に二つのファイルを処理する簡単な例を示す．ファイル内容を読み込み内容を更新してその結果を別のファイルに出力する単純な例である．もしこの二つのファイルがハードウェア的に独立したアクセスパスをもっていても，処理の時間的な流れは**図 3･17** のようになる．つまり，入力ファイルと出力ファイルは独立に逐次的なアクセスしかできない．したがって，ユーザプログラムの範囲で性能向上は望めない．

```
while(1) {

    read(rfd, rbuf, rlen);
    /* 読み込みデータの更新 */

    write(wfd, wbuf, wlen);

}
```

図 3・16　複数ファイルを操作する例

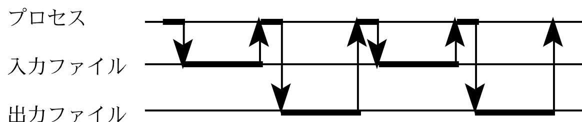

図 3・17　同期入出力による入出力操作

(3)　非同期入出力

上記の欠点を解決する方法が非同期入出力方式である．この方式では，入出力完了事象をユーザプログラムによって確認するシステムコールを実行しなくてはならない．つまり，カーネルとユーザプログラムが同期をとることになる．

図 3･18 には先ほどと同じプログラムを同期入出力にしたときの例を示す．ここでは，読込みの同期をとるシステムコールを rwait( ) とし，書込みの同期を wwait( ) とした．読込みの同期は読込みデータを処理する前に行う必要があり，また，書込みの同期は次に書込みを行うために前回の書込みが完了していることを確認するために行う．

```
While(1){
        read(rfd, rbuf, rlen);
        /* 読込み完了を待つ */
        rwait(rfd);
        /* 読込みデータの更新処理 */
        /* 書込みの完了を待つ     */
        wwait(wfd);
        write(wfd, wbuf, wlen);
}
```

図 3・18　複数ファイルを非同期入出力操作する例

このような操作を行うには，2 面の入出力領域を用意する必要があるが，処理時間としては入出力がオーバラップできるので, 同期入出力処理に比べて, 2 倍程度の性能向上になる.

図 3･19 には二つのファイルがハードウェア的に独立に動作できる環境の場合, 定常状態になっている非同期入出力処理の時間的な流れを示す．このように，入出力がオーバラップされるので，処理時間の短縮につながる.

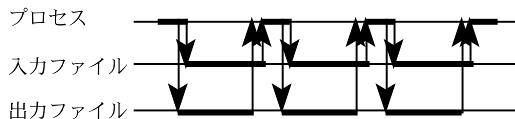

図 3・19　非同期入出力による複数ファイル操作

■7 群 － 3 編 － 3 章

3-4 演習問題

（執筆者：吉澤康文）［2013 年 2 月 受領］

(1) ファイルシステムの目的を 2 種以上あげ，説明せよ．

(2) ブロッキングの効果を述べよ．

(3) 入出力でのバッファリングの効果が生まれるのはいかなる場合か説明せよ．

(4) UNIX のファイルシステムなどで採用されている木構造の利点は何か.

(5) ファイル入出力のソフトウェア階層はなぜ存在するのか.

(6) ファイルのオープン処理は何を行うのか説明せよ．

(7) 順編成ならびに直接編成ファイルとはどのようなものか説明し，各々の利点と欠点をあげよ．

(8) ファイル入出力の高速化方式を 3 種類あげ，それらの方式を説明せよ．

(9) ファイルの保護の目的は何か説明せよ．そして，具体的にはどのような保護機構があるか説明せよ．

(10) UNIX 系ではファイルを作成する場合にあらかじめファイルサイズの指定を行わない．一方，MVS では指定を行い，領域を確保できることを確認する．この両者の利点と欠点を考察せよ．

(11) UNIX ファイルシステムではファイルの内容をまったく意識しない．一方，MVS ではファイルはレコードの集合としてとらえている．このような考え方で設計されたファイルシステムの利点と欠点は何か.

(12) ファイルはスペース管理，ディレクトリ，ファイルの実体の三つの部分から成っているが，信頼性を確保するためにはこの書込みの順序を考察せよ．

(13) 入出力の実行に失敗したとき，どのような方法をオペレーティングシステムはとればよいか.

(14) ファイルの信頼性を上げるためにとられる手段にはどのようなものがあるか説明せよ．

(15) ファイルの同期入出力と非同期入出力の違いを述べ，各々の利点，欠点を述べよ．