■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）

6 章 UNIX ファイルシステム

（執筆者：吉澤康文）［2013 年 2 月 受領］

■概要■

3 章ではファイルシステムの基本的な概念とその理解に必要な最低限の機能を説明した．本章ではケーススタディとして取り上げている UNIX のファイルシステムをより深く説明する．ここでは，プログラマとの接点となるファイルシステムのシステムコールを具体的に示し解説している．これらを理解することで，UNIX を用いたファイル処理が可能となるはずである．

【本章の構成】

UNIX を例示として取り上げるが，具体的なファイルの格納方法を理解するために，他の OS との比較もここに示す（6-1 節）．UNIX におけるファイルを理解するために必要なファイルの分類を解説する（6-2 節）．次に，各々のファイルの構成がどのように構築されているか，またそれらの具体的なデータ構造について説明する（6-3 節）．UNIX が提供するプログラマへのインタフェースであるシステムコールならびに関連するライブラリなどを具体的に示すと同時に，各々のパラメータの意味を解説する（6-4 節）．

■7 群 － 3 編 － 6 章

6-1 基本的な考え方

（執筆者：吉澤康文）［2013 年 2 月 受領］

6-1-1 他 OS のファイルとの比較

各 OS は独自のファイルシステムを備えている．UNIX も同様である．ここでは，1960 年代に開発され今日の OS の源流となっている IBM 社の OS/360 の発展形である MVS との比較を行う．ここに比較する MVS のファイル機能は一部であり，OS/360 時代の原形である．

表 6･1 に比較した一覧を示す．ファイルには名前を付けるが，その方法は UNIX では木構造を提供しているためユーザが管理しやすい方法になっている．これは，MIT の Multics と同じ考えを採用したためである．

表 6・1　UNIX と MVS のファイルに関する概念の違い

ＯＳ名	MVS	UNIX
ファイル名空間	フラットな構造	木構造を提供
ファイル管理	VTOC/Catalogue	ディレクトリ，i-node
ファイル編成	数種類を規定 (SAM,DAM,PAM,VSAM)	なし
アクセス法	ファイル編成に応じて 存在	順ファイル編成のみ
データ構造	各種存在 （レコード形式）	バイト列のみ ☆アプリケーションの 責任で規定すべきである

6-1-2 ファイル格納方式

磁気ディスクにファイルを格納するとき，ファイル名，その属性やボリューム内の所在を示す方法は各 OS で独自の方法がとられている．OS/360 では，VTOC（Volume Table of Content）により個々のファイルの情報を管理している．**図 6･1** には各ボリュームを管理する情報の格納形式を示す．磁気ディスクの先頭には，IPL（Initial Program Loader）レコードが 2 ブロック存在し，その後に，VSN（Volume Serial Number）レコードが書き込まれている．VSN には

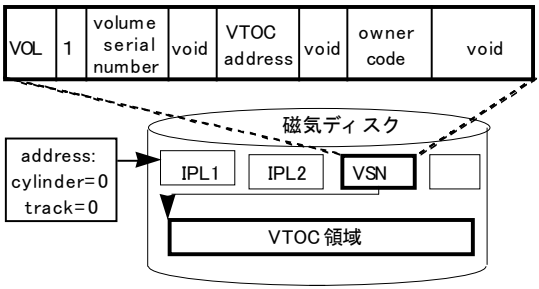

図 6・1　OS/360 における VIOC

ボリューム名称を書き込むだけでなく，VTOC のアドレスが存在し，その先に，VTOC 領域がある．

VTOC は，ボリューム内の空き領域，格納されているファイルに関する管理情報を DSCB（Data Set Control Block）というレコードにして格納しているので，DSCB のレコードの集合になっている．

6-1-3 UNIX でのファイル格納方式

UNIX では，**図 6･2** に示すように各ボリュームに定まった形式のブロックを構成し，ファイル管理を行う．ブートブロックにはカーネルのブートストラップのプログラムが書き込まれている．第 2 番目のブロックは，スーパブロック（Super Block）と呼ばれ，アイノード（i-node）の大きさ，ボリューム内の空き領域管理情報などが格納されている．i-node はインデックスノード（Index）の略称で，UNIX のファイル管理において重要な情報を含んでいる．詳しくは後の節で説明する．

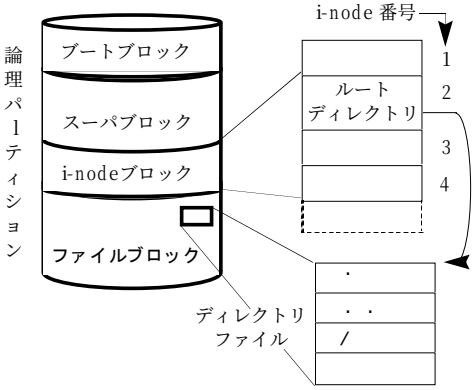

図 6・2 UNIX におけるファイル管理情報の格納

i-node ブロックは i-node のレコードが配列になっており，第 2 番目のレコードが本ボリュームのルートディレクトリ（Root Directory）になっている．つまり，本ボリューム内のファイルの検索を行う場合は，ルートディレクトリを起点にし，ディレクトリファイルと i-node を交互に参照することにより目的のファイルにたどり着くことができる．このアルゴリズムも後の節で説明する．

6-1-4 ファイル編成とアクセス法

OS/360 では，3 章 3-3-2 項で説明したように，ファイルを順編成と直接編成の二つに分類している．この分類を基にし，SAM，DAM なるアクセス法（Access Method）を提供している．

一方，UNIX のファイルシステムは基本的に順編成ファイルを提供しているだけであり，アクセス法を提供していない．

6-1-5 データ構造

OS/360 と UNIX ではデータ構造に著しい差がある．OS/360 では，ファイルに格納されている情報の基本単位はレコードであり，レコードを格納するファイルとしてファイル編成があり，操作する方法をアクセス法としている．レコードとは，プログラマが設計したデータ構造のことであり，OS/360 はレコードを基本としたファイルシステムを構築している．

一方，UNIX は OS/360 とは対照的で，ファイルに構造はもたない．UNIX では「ファイルはバイトの一次元配列であり，ファイル内のデータに関する構造，データ形式などはプログラムが認識すべきでカーネルは全く関与しない」という考えに立っている．したがって，基本的には指定されたファイルの位置から指定されたバイト数をアクセスするという機能が提供されている．詳しくはシステムコール lseek を参照のこと．

■7 群 － 3 編 － 6 章

6-2 ファイルの分類

（執筆者：吉澤康文）［2013 年 2 月 受領］

6-2-1 3 種類のファイル

UNIX では以下の 3 種類にファイルを分類している．

・通常ファイル
・ディレクトリファイル
・特殊ファイル

6-2-2 通常ファイル

通常ファイル（Ordinary File）はテキストファイルやプログラムをコンパイルしたオブジェクトコード，画像・音声データなどを格納したファイルである．ファイルの内容に意味付けするのは，そのファイルを取り扱うプログラムである．例えば，テキスト編集（Text Editor）プログラムは文字列や行などのテキストファイルとしてデータを取り扱い，プログラムローダ（Program Loader）が対象とするファイルはバイナリコード（Binary Code）としてデータを解釈する，などがその例である．

通常ファイルは順編成であるので，途中にデータを追加し，途中のデータを削除することは不可能である．このような操作をするには，別のファイルを作成することになる．通常ファイルは作成者の所有物であり，多くの場合，作成者のディレクトリの下に格納されているが，許可されていれば，木構造の別の場所に配置することも自由である．

6-2-3 ディレクトリファイル

ファイル名とファイルの実体を対応付けるのがディレクトリファイル（Directory File）である．UNIX ではディレクトリを一つのファイルとして扱う．しかし，通常ファイルとは区別している．ディレクトリファイルはスーパユーザ（Super User）の所有物であるため，一般ユーザのプロセスによるディレクトリへの書込みや削除は直接実行できない．

ディレクトリは木構造のファイルシステムを実現している．3 章 3-3-1 項において説明したように，ファイル名とパス名が UNIX では存在する．UNIX ユーザがログインした時点でのディレクトリはホームディレクトリ（Home Directory）と呼ばれ，ユーザ情報ファイルである/etc/passwd 内に記述され指定されている．ログイン時点ではホームディレクトリがカレントディレクトリ（Current Directory）であるが，cd（Change Directory）コマンドで自由にカレントディレクトリを変更できる．現在のディレクトリを表示するコマンドは pwd（Print Working Directory）であり，ls（List）コマンドと共によく利用される代表的なコマンドの一つである．

6-2-4 特殊ファイル

入出力装置やメモリなどを通常のファイルと全く同じようにアクセスするために設けたファイルある．例えば，磁気ディスク，フロッピーディスク，メモリ，プリンタなどへのアクセスの際に UNIX のファイル機能をそのまま利用することが可能になるが，装置ごとにアク

セス方法は定まっている．

特殊ファイル（Special File）には，ブロックスペシャルファイルとキャラクタスペシャルファイルがある．ブロックスペシャルファイルは一定のサイズからなるデータの入出力である．例えば，磁気ディスクはその例である．これらの装置の入出力には UNIX が性能向上のために，システムバッファを用いている．キャラクタスペシャルファイルはキーボードやディスプレイなどのようにデータサイズに規則がない装置である．

一般的に，特殊ファイルを扱うケースは，UNIX が標準的にサポートしていない装置のドライバプログラムや，UNIX カーネルのサポートしているバッファキャッシュの操作を受けることなく入出力を行うシステムプログラムなどで，スーパユーザの権限で実行されるプログラムが多い．例えば，データベース管理プログラム，OLTP の制御プログラムなどの開発に利用される．

特殊ファイルは，「/dev/....」のファイル名をもっている．これらのファイルは mknod（Make a Special File）システムコールで作ることができるが，スーパユーザの権限がなくてはならない．

■7 群 － 3 編 － 6 章

6-3 ファイルシステムの構成

（執筆者：吉澤康文）［2013 年 2 月 受領］

6-3-1 ディレクトリを作る

ディレクトリを作成するコマンドは mkdir（Make Directory）である．mkdir で新しいディレクトリを作成すルートディレクトリファイルには二つの決まったファイル名をもつエントリが作られる．それらは以下の二つである．

(a) . カレントディレクトリ（ピリオド一つ）

(b) .. 親ディレクトリ（ピリオド二つ）

「.」は作成したディレクトリファイル自身の名前であり，「..」はディレクトリの親ディレクトリのファイル名である．したがって，カレントディレクトリの上位のディレクトリに戻りたい場合は,「cd ..」を端末から入れるだけでよい. また, ホームディレクトリには「~」が予約されているので,「cd ~」を使えばいつでもホームディレクトリに戻ることができる．

6-3-2 ディレクトリの構造

UNIX のディレクトリは以下の二つの情報のみをもつ.

(a) ファイル名

(b) i-node 番号

BSD 版 UNIX ではファイル名の最大長が 255 文字であるが，ディレクトリのスペースを節約するために，固定長のエントリとせず，有効な文字のみ格納している．**図 6･3** にはディレクトリの構造を概念的に示した．各々の UNIX のインプリメンテーション（Implementation）により異なる部分があるが，情報としてはほぼ同一の内容が格納されている．i-node については次に説明する．

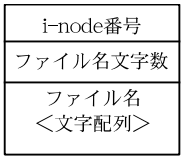

図 6・3 ディレクトリエントリの構造

6-3-3 i-node の情報

UNIX ではファイルをユニークに定める番号として i-node 番号を使っている．i-node 番号は図 6･2 に示したように，磁気ディスクの特定の位置に格納されている i-node 配列のインデックスである．

ファイルの名前以外の情報はすべて i-node 内に格納されている．**図 6･4** に i-node とファイルの実体との関係の一例を示した．i-node の構成は UNIX のインプリメンテーションに依存しているので統一されているわけではない．このため，一般ユーザへの i-node 情報を共通的に提供するインタフェースとして，stat( ) システムコールが用意されており，stat 構造体により図 6･4 の情報を取得することができる．詳しくは後の項で説明する．

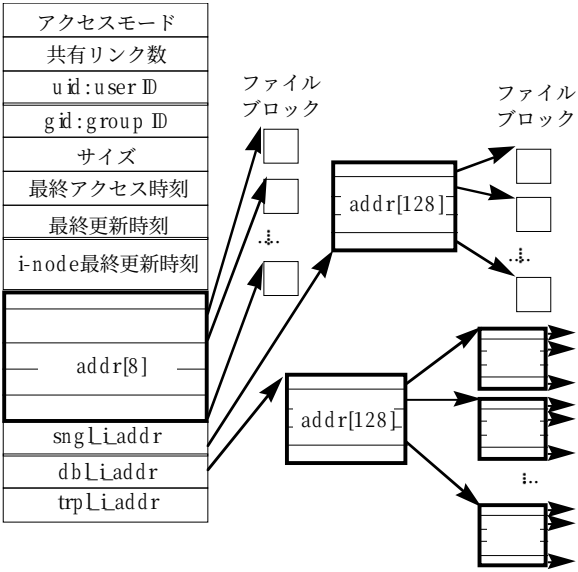

図 6・4　i-node 内情報とファイル実体の関係

(1)　アクセスモード

UNIX のファイルアクセス制御は，アクセスを許すユーザを，以下の三つに分けて行う．

・ファイルを作成したユーザ自身（所有者：owner）
・ユーザの共同作業者であるグループ内のユーザ
・それ以外のユーザ（other）

上記のユーザに対して，各ファイルに属性（attribute）をもたせる．属性は 3 種類あり，読出し（read），書込み（write），実行（execute）を許可するか否かを指定できる．つまり，**図 6･5** に示すような 9 ビットの情報により，所有者，グループ，他ユーザに対して r，w，x が可能か否かを表示する．例えば，作成したファイルを自分とグループのユーザは読み書きを可能とするが，他ユーザには読出しだけ許すようにするときは，「0664」と指定する（オクタル表示であり，ビットイメージでは，110110100）．これらは，ファイルを作成する creat( )，open( ) やモード変更の chmod( ) などのシステムコールのパラメータで指定する．

r w x	r w x	r w x
所有者 (owner)	グループ	他ユーザ (other)

r:　read 許可ビット，w:　write 許可ビット
x:　execute許可ビット

図 6・5　i-node 内アクセスモードの表示

アクスモードのフィールドには，上記のアクセス制御情報以外に，ファイルの種別（通常ファイル，ディレクトリ，特殊ファイル）の表示がある．アクセスモード内の情報はリスト

コマンド（ls -l）で表示できる．

(2) 共用リンク数

UNIX では一つのファイルを複数のユーザで共有することができる．共有は，コマンドのリンク（ln）で行われる．プログラムからは，link( ) システムコールで行う．**図 6･6** にそのイメージを示した．このような共有ファイルは，実体は一つであり，その i-node の共用リンク数が 2 以上になっている．したがって，最初にファイルを生成した点での共用リンク数は 1 である．

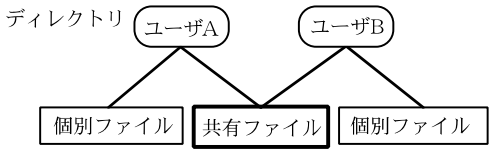

図 6・6　ファイル共有

ファイルを消去するコマンドは「rm」（Remove）であるが，プログラムからは，unlink( ) システムコールが実行される．unlink( ) システムコールは，共用リンク数を 1 だけマイナスする．その結果，ゼロになるとファイルの消滅を意味するので，ファイル実体を消滅させるために，ディスクブロックアドレスの情報を基に占有していたブロックを解放し，その後，i-node の解放も行う．

(3) ユーザ ID，グループ ID

所有者の表示ならびにアクセス制御のために，これらの情報が格納されている．

(4) サイズ

ファイルサイズをバイト数単位で格納してある．

(5) 最終アクセス時刻など

最終アクセス時間，最終更新時刻，i-node 最終更新時刻をセットしてある．1970 年 1 月 1 日午前 0 時グリニッチ標準時からの秒が入っているので，必要に応じて，ローカルタイムに変換する必要がある（ctime( ) なる関数で変換可能）．

(6) ディスクブロックアドレス

ファイルの実体が格納されているブロックアドレスを示すテーブルである．ディスクのブロックサイズは UNIX のインプリメンテーションに依存する．図 6･4 では，addr[8] のように 8 ブロックを一つの割当て範囲にしている．仮にブロックサイズを 8 KB とするならば，64 KB までが i-node から直接示すことができるブロックアドレスである．

しかし，それ以上のサイズのファイルになると，i-node から直接ブロックアドレスを示すことができない．そこで，ブロックアドレスを最大 128 個格納できる領域（addr[128] ）を作成し，セクタアドレスを sngl_i_addr にセットする．このようにすることで，64 KB 以上の領

域を格納することができる．この第一段テーブルでは，1,024 KB 分のファイルブロックを示すことが可能となる．これ以上のサイズになると，同様の構造を第 2 段テーブル構造として作成し，その起点を i-node の dbl_i_addr にセットする．これらを間接ブロックアドレス（Indirect Block Address）と呼ぶことがある．

6-3-4 指定されたファイルへたどりつく

UNIX におけるディレクトリと i-node の仕組みを説明してきた．ここでは，指定されたパス名から，この二つのファイル管理情報を用いてファイルを探索する方法を説明する．ファイル探索は**図 6･7** に示すように入力をファイル名（場合によるとパス名）が与えられて，ファイルの実体に到達するまでの道のりである．

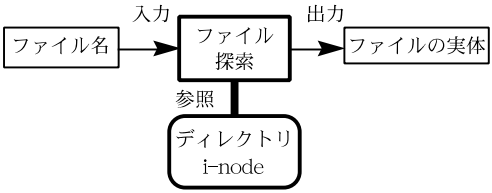

図 6・7　UNIX でのファイル探索の方法

具体的な事例として， /home/yosi/sdl のファイルを探索する．**図 6･8** にはその手順を示した．最初に，ルートディレクトリ "/" 内をサーチする．ルートディレクトリは図 6･2 で説明したように，第 2 番目の i-node と UNIX では定まっているので，その実体を読み込むことができる．図 6･8 にはルートディレクトリを読み込んだ様子を示す．したがって，ルートディレクトリの中からディレクトリ名 "home" を探す．ディレクトリの形式は図 6･3 に示したとおりである．

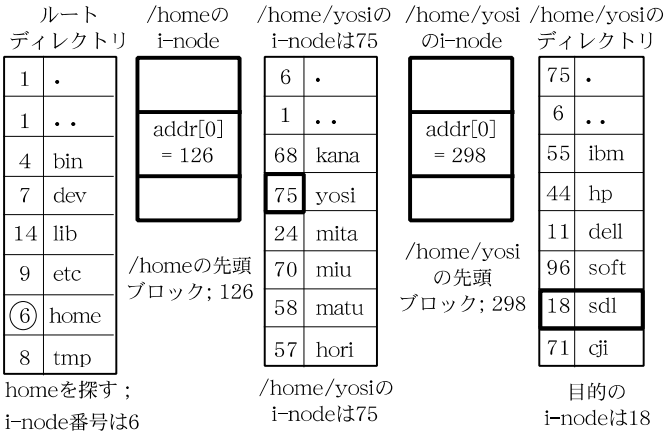

図 6・8　ファイル探索の具体的例

"home" を見つけたエントリの i-node 番号は「6」である．したがって，i-node の第 6 エントリが "/home" の i-node となる．この結果，図 6･6 に示したように，"/home" のディレクトリファイルのブロックアドレスを求めることができる. この例では, ブロックアドレスが「126」である．したがって，この部分を読み込み "/home" のディレクトリ内を探索できるようになる．

/home のディレクトリ内から，次の目標である "yosi" なるファイル名を探す．エントリが見つかると，その i-node 番号「75」を得る．同様の手順で，/home/yosi の i-node を参照し，ブロックアドレス「298」を得る．この結果，ディレクトリ /home/yosi/ の中から目的の "sdl" を探し，その i-node 番号「18」を得る．

上記の手順により，UNIX ファイルは i-node とディレクトリを順にたどることにより目的のファイルの実体をアクセスすることが可能になる．この操作を行う典型的なケースは，ファイルのオープン処理である．

■7群 － 3編 － 6章

6-4 UNIX ファイルシステム操作

（執筆者：吉澤康文）［2013年2月 受領］

6-4-1 基本ファイル操作

(1) プログラムの幅を広げる

ここでは通常ファイルに対する基本的なファイル操作のシステムコールを説明する．ファイル入出力が行えるようになるとプログラミングの幅が飛躍的に拡大する．そこで，最初に，共通して使用するファイル記述子について説明し，ファイル生成，ファイル使用宣言のオープン，使用完了のクローズ，ファイルの読込み，書込み，そして，ファイルのランダムアクセスを可能にする機能などについて順に説明する．

(2) ファイル記述子

初期の UNIX（version 7）では，各プロセスは 0 から 19 までのファイル記述子が使用できた．最近の UNIX では，この制限数を運用によって変えることができ，大きくなってきている．ファイル記述子はオープンされたファイルを識別する正の整数であり，プロセスはファイル記述子で指定できる複数のファイルを取り扱える．

シェル（Shell）を備えた UNIX の端末でログインすると，**図 6･9** に示すように，デフォールト（Default）で三つのファイル記述子がオープンされた状態になっている．

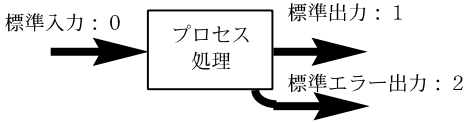

図 6･9　UNIX でのログイン直後のファイル記述子

この場合，ファイル記述子 0 はキーボード，標準出力とエラー出力はディスプレイである．標準出力はほかのファイルやプロセス管理で述べたパイプなどになっていることがあるため，重要なメッセージの出力は標準エラー出力 2 に行うべきである．

ファイル記述子はシステムコール，creat( )，open( )，dup( )，fctl( )，pipe( ) などが成功したときに作成される．BSD ではソケット（socket）を識別する整数値として，ソケット記述子（Socket Descriptor）があり，ファイル記述子と区別せずに使う場合もある．この場合は，socket( ) システムコールが成功したときに作成される．

ログイン直後では，0～2 のファイル記述子がオープンしているので，read( )，write( ) システムコールで直ちにキーボードやディスプレイに対して読込み，書込みが可能となる．

(3) ファイルの生成：creat

新しいファイルの作成はシステムコール creat( ) で作成する．仕様は**図 6･10** に示すとおりである．第 1 パラメータにはパス名（ファイル名）を指定するが，同一名があるとファイルサイズがゼロとなるだけで perms は反映されない．システムコール名は "creat" であって，"create" ではないことに注意すること．

```
int creat(path, perms) /* ファイルを作成する */
char     *path;        /* パス名を指定  */
int        perms;     /* アクセスフラグを指定 */
/* 返り値：正の整数ならファイル記述子 */
/*         失敗のとき -1   */
```

図 6・10　システムコール creat **の仕様**

新ファイル作成時に，ファイルのアクセスフラグを設定する．これは図 6･5 に示したように 9 ビット存在し，自分，グループ，他ユーザの各々に，read，write，execute を許可ときは 1 を指定する．したがって，自分とそのグループに read，write を許し，他ユーザにはアクセスさせないようにするには，オクタル（Octal）表示で，0660 とする．返り値は，正の整数ならばファイル記述子である．失敗は-1 となる．

creat では，ファイルの初期作成であるため，たとえ perms のパラメータで読込み専用ファイルと指定しても書込み可能な記述子が返され，creat したプログラム実行中はファイルへの書込みが可能となる．そこで，読込みも書込みもしたいときは，一度ファイルをクローズして，更新モードでオープンする必要がある．

(4)　ファイルオープン：open

既に存在するファイルへのアクセスはオープンしてファイル記述子をカーネルから取得することから始める．**図 6･11** にシステムコール open の仕様を示す．既存ファイルが存在するときは flags で指定したアクセス要求となる．flags は，0 は読込み，1 は書込み，2 は更新（読込みと書込み）を示す．これらはシンボリックな指定をすることが可能である．例えば，O_RDONLY は読込みのみ，O_WRONLY は書込み，O_RDWR は更新，O_APPEND はファイルの最後から書込みなどである．

```
int open(path, flags[, perms])
char     *path;    /* パス名を指定  */
int        flags;  /* read, wirte, update */
                   /* の要求フラグ */
int        perms;   /* アクセスフラグ */
/* 返値：正の整数ならファイル記述子  */
/*    失敗のとき  -1   */
```

図 6・11　システムコール open **の仕様**

指定した path のファイルがなく，第 3 パラメータが指定されている場合は creat と同一で，新ファイルの作成になる．すなわち，第 3 パラメータは creat と同じ意味となる．システムコールの返り値は成功するとファイル記述子が，失敗ならば-1 である．

(5)　ファイルの読込み：read

オープンされているファイルからデータを読み込むのが read システムコールである．指定するパラメータは，ファイル記述子，ファイル読込み領域アドレス，そして読み込むデータ長である．仕様は**図 6･12** に示すとおりである．

```
int read(fd, buf, nbytes)  /* ファイル読み込み */
char   fd;             /* ファイル記述子を指定 */
int      *buf;           /* 読み込み領域を指定  */
int      nbytes;          /* 読み込み長を指定 */
/* 返値： 0 より大きなの値なら読み込みバイト数    */
/*       read 失敗なら -1                      */
```

図 6・12　システムコール read の仕様

返り値は，-1 のときは read( ) のエラーであるが，それ以外は読み込んだバイト長である．この値は，第 3 パラメータで要求した読込みデータ長に等しいか小さな値である．読込みバイト数がゼロの場合は，ファイルの最後にファイルポインタが到達していることを意味している．つまり，EOF（End of File）である．

読込みが完了するとファイルポインタは読み込んだバイト数だけインクリメント（Increment）される．そして，次の読込み要求は，インクリメントされたファイルポインタから実行される．

(6)　ファイル書込み：write

ファイルの書込みはシステムコール write( ) で行う．read( ) 同様にファイルはオープンされていなければならない.パラメータの指定も読み込みシステムコールと形式は同じである．返り値は書き込んだデータ長であり，通常は指定した書込みバイト長と同じであるが，正しくファイルを書き込むためにはチェックを行う必要がある．**図 6･13** にその仕様を示す.

```
int write(fd, buf, nbytes) /* ファイル書き込み */
char   fd;             /* ファイル記述子を指定 */
int      *buf;         /* 書き込み領域を指定     */
int      nbytes;       /* 書き込み長を指定       */
/* 返値： 0 より大きなの値なら書き込みバイト数    */
/*       write 失敗なら -1                     */
```

図 6・13　システムコール write の仕様

書込みもファイルポインタから書込みを行い，書込みしたデータ長の分だけファイルポインタはインクリメントされる．磁気ディスクなどに格納されている通常ファイルへの書込みは，システムバッファに書き込むだけで完了してしまう．つまり，磁気ディスクにまで書込みは行われない．UNIX ではこのような動作であるため，システムバッファから本来のファイルの実体の位置への書込みの遅延（Delay）があるので，実際の書込み操作はプログラムが完了した後のこともある．つまり，物理的な書込み時のエラーが知らされることはない，という問題がある．

したがって，ファイルシステム内にデータ間の関連があるようなレコードを書き込むような処理で，かつ，フェールセーフな設計を行う場合には十分な注意が必要となる．このときは後に述べる，システムコール sync( ) を用いることにより，確実に磁気ディスクへの書込みが完了してから次の処理に進めるように設計しておかねばならない．このことからも，信頼性と性能とはトレードオフ（Trade Off）の関係にあることを見てとれる．

(7) ファイルポジション：lseek

read( ), write( ) しているファイルのファイルポインタを操作するシステムコールである．**図 6･14** に仕様を示す．ここでは，第 3 パラメータ where が重要である．where が 0 のときは，第 2 パラメータの offset でファイルの先頭からのバイト位置にファイルポインタを移動できる．したがって，このときはマイナスの値は許されない．

```
long lseek(fd,offset,where)/* ポインタを移動 */
int fd;   /* ファイル記述子を指定 */
long offset; /* ファイル内のオフセット値 */
int where;  /* 0:オフセット，1:相対的オフセット
                  2:最後のポインタ＋オフセット  */
/*  返値：ファイルポインタ値          */
/*       失敗なら  -1                 */
```

図 6･14　システムコール lseek の仕様

ほかの方法でも同様であるが，ファイルサイズを越えるようなファイルポインタ値を指定してもよい．もし，ファイルサイズを越えるようなファイルポインタとした後に，write( ) で書込みを行うと，途中のデータは不定となる．つまり，何が入っているかの保障はない．この部分は read( ) してもエラーにはならず，内容は保障されないことになっているが，大抵はゼロの値を返される．このようなキセルのようなファイルは特別な理由がないかぎり作るべきでない．

第 3 パラメータ where の値が 1 のときは，現在のファイルポインタに対して第 2 パラメータ offset 値が加算される．したがって，この場合は offset 値はマイナスであってもよいが，ファイルポインタがマイナスになるとエラーとなる．

第 3 パラメータ where の値が 2 のときは，ファイルの最終アドレスに対して offset 値が加算される．大抵の使い方としては，offset をゼロとして EOF のポジションにファイルポインタを移動して，データを付け加える（append）か，もしくは，マイナスの値を指定して，EOF よりも手前にあるデータのポジションとするかのどちらかである．なお，open( ) で O_APPEND が指定された場合は，ファイルオープン時に EOF のポジションにファイルポインタは位置付けられている．

lseek( ) で最も注意しなくてはならないのは，オフセット値が long 型整数でなくてはならないことと，返り値も同じように long 型整数である点にある．このようなことから，lseek( ) と名付けられている．

(8) ファイルのクローズ：close

オープンされているファイル記述子を解放するだけである．close( ) を実行しても write( ) によりシステムバッファ内のブロックを磁気ディスクにフラッシュバック（書込み）することはない．仕様は**図 6･15** に示したとおりである．何もしないからとの理由で close( ) を実行しないプログラミングの習慣はよくない．open( ) に対して close( ) することで閉じた論理になるので，行儀のよいプログラムを心掛けるべきである．ファイル記述子は無限に利用できないので，その意味でも不要になったファイル記述子は close( ) によって解放する方がよい．

```
close(fd)  /* ファイルのクローズ */
int fd;    /* ファイル記述子を指定 */
```

図 6・15　システムコール close 仕様

6-4-2 関連システムコールとライブラリ

(1)　ディレクトリを作る：mkdir

システムコールの仕様を**図 6･16** に示す．作成するディレクトリ名を指定し，第 2 パラメータでは，アクセスモードを指定する．アクセスモードは open( ), creat( ) などと同じであるが，umask( ) によって支配される．umask( ) は後で説明する．

```
int mkdir(pathnm, mode)  /* directoryを作成する*/
char *pathnm;            /*  path name指定     */
int  mode;               /* ファイルモードの指定 */
```

図 6・16　システムコール mkdir の仕様

(2)　ディレクトを読む：opendir，readdir

UNIX のインプリメンテーションに依存するが，ディレクトリはファイルとして読むことが可能である．しかし，BSD 系とそれ以外ではディレクトリ形式が異なるので，opendir( ) でディレクトリをオープンし，readdir( ) によってディレクトリエントリを読み込むのが便利である．

図 6･17 には opendir( ) の仕様を示す．パラメータにはディレクトリ名を指定する．返り値は，ディレクトリをストリームデータとして読むためのポインタ DIR である．エラーの場合は，NULL が返される．

```
#include <sys/types.h>
#include <dirent.h>
/* ディレクトリをオープン */
DIR *opendir(const char *name);
/* 返り値：ディレクトリストリームへのポインタ
    エラー発生はNULLを返す */
```

図 6・17　ディレクトリオープン

opendir( ) では，ディレクトリをオープンし，そのファイル記述子やディレクトリをストリームとして読み込むので，次に読み込むディレクトリエントリの相対アドレス，そして，ディレクトリエントリを読み込む領域などを DIR 構造体に作り，そのアドレスを返す．

ストリームとは，磁気ディスクなどの装置とプロセスが，データを連続的に送り込んだり，読み込んだりすることを意味している．DIR は ANSI 標準の標準ライブラリにおける<stdio.h>で定義されているファイルポインタ FILE と同じ性格のものである．FILE は fopen( ) などの返り値ストリームである．<stdio.h>では，ストリーム stdin，stdout，stderr がログイン開始時点でオープンされている．これは，ファイル記述子 0，1，2 に対応している．

opendir( ) で取得した DIR を指定して，readdir( ) ライブラリによってディレクトリエントリを次々と読込むことができる．成功すると dirent 構造体のアドレスが返り値となる．ディ

レクトリエントリがなくなるかあるいは，エラーの場合は NULL が返される．**図 6･18** は readdir( ) の仕様である．

```
#include  <sys/types.h>
#include  <dirent.h>
struct dirent *readdir(DIR *dir)
/* directoryを読む*/
/* 返り値：dirent構造体のポインタ．ファイルの
    最後あるいはエラー時はNULL */
```

図 6・18　ディレクトリを読む

図 6･19 に readdir( ) の結果，読み込まれた dirent の構造体を示す．d_ino に i-node 番号を，d_name[ ] にファイルあるいはディレクトリ名を参照することができる．d_name[ ] は，最大 NAME_MAX の文字と，終端するヌル文字が格納されている．ここで，d_ino[ ]の値がゼロのときは，そのエントリは無効となっていることを示しているので，必ず判定してからファイル名を参照すべきである．

```
struct dirent {
long d_ino;      /* inode番号 */
off_t  d_off;    /* 本エントリまでのオフセット値 */
unsigned short d_reclen;   /* 名前の長さ */
char d_name [NAME_MAX+1];  /* ファイル名 */
}
```

図 6・19　Linux でのディレクトリエントリの構造

(3)　ファイル生成マスク：umask

ファイルの生成はシステムコールの creat( ), open( ) などで行う．また，ディレクトリファイルも mkdir( ) により作成される．このとき，いずれもパラメータに 9 ビットのアクセスモードを指定する必要があるが，umask( ) は，たとえそれらのシステムコールで指定されたアクセスモードであっても設定を拒否するマスクを定義することができる．

システムコールの仕様は**図 6･20** に示す．例えば，umask(022); を実行しておくと，creat("yoshiza", 0666); のように設定を試みても，アクセスモードのビットは 0644（0666 & ~022 = 0644 = rw -r- -r- -）となり，グループと他ユーザは書込みが不可能な属性になる．つまり，umask( ) で指定したビット位置はファイル生成時にはビットをオンにできないことになる. 返り値はそれまでの cmask 値である. また, コマンドに本機能を使用する umask がある.

```
int umask(cmask) /* file mode creation mask の実行*/
int cmask;       /* ファイル作成時のpermの該当ビット
                    をクリアする */
/* 返り値：前回設定したcmaskの値 */
```

図 6・20　システムコール umask の仕様

(4)　カレントディレクトリの位置を変える：chdir

カレントディレクトリの上位のディレクトリに移動するには，chdir(".."); である．コマン

ドには cd がある．仕様は**図 6･21** のとおりである．

```
int chdir(pathnm)  /* current working directory
          を変更する：cd コマンドをプログラムで実行*/
char *pathnm;       /* path name指定 */
/* 成功の場合は 0，失敗は -1 */
```

図 6・21　システムコール chdir **の仕様**

(5)　i-node の内容を知る：stat

ファイル管理はディレクトリと i-node によって行われている．ディレクトリは opendir( ), readdir( ) のライブラリで読むことができた．ここでは，i-node を読み込むライブラリである stat( ) を説明する．stat( ) では i-node のすべての情報ではないが，一般ユーザに必要と思われる情報を UNIX に共通した構造体 stat で提供する．システムコールの仕様は**図 6･22** に示すとおりである．

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(char *pathname, struct stat *buf);
int fstat(int fildes, struct stat *buf);
/* 成功の場合は 0，失敗は -1 */
```

図 6・22　システムコール stat，fstat **の仕様**

stat( ) ではパス名を指定するが fstat( ) では open( ) の返り値であるファイル記述子を指定する．共に，指定した stat 構造体の領域に i-node の一部の情報が格納される．stat 構造体の一部の変数を**表 6･2** に示す．i-node の一部の情報はコマンド ls によって表示することができる．表 6･2 では，ファイルが作成された時刻が GMT で表示されているが，この値の変換は ctime( ) ライブラリなどで文字にすることができる．UNIX では，オンラインマニュアルが充実しているので，man コマンドを用いて更に詳しく知ることができる．

図 6・2　**stat **構造体の代表的な変数

変数名	型	変数の意味
st_ino	ushort	i-node番号
st_mode	ushort	アクセス，ファイル種別など
st_nlink	short	リンクカウント＜共有数＞
st_uid	ushort	所有者のuser ID
st_gid	ushort	グループID
st_size	long	ファイルサイズ
st_atime	long	最後にアクセスされた1970年からの秒
st_mtime	long	最後に書き込まれた時刻
st_ctime	long	i-nodeの変更更新された時刻

(6)　i-node 内の情報を変更する

ファイルを生成した後にアクセスモードを変更し，また所有者名を変更しなくてはならな

い場合が起きることがある．コマンドでは chmod（Change Mode），chown（Change Owner）などで実行できるが，それらをプログラムで実行する場合のシステムコールは，chmod( )，chwon( ) である．

図 6･23，**図 6･24** に chown( )，chmod( ) の仕様を示す．chown( ) はスーパユーザ以外の使用には制限がある場合もある．

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *path, uid_t owner,
                                gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
/* 成功の場合は０，失敗は -1 */
```

図 6・23　システムコール chown **の仕様**

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
/* 成功の場合は０，失敗は -1 */
```

図 6・24　システムコール chmod **の仕様**

(7)　システムバッファのライトバック：sync

UNIX ではファイル入出力の高速化のために，システムバッファ（バッファキャッシュと呼ぶこともある）を使ってファイルのキャッシングを行っている．ファイルの先読みはキャッシングによる物理的入出力削減の効果が期待できる．しかし，書込み時には，システムバッファへの書込みがなされ磁気ディスク上のファイル実体への書込みは行わない．このため仮にシステムダウンを起こすと，システムバッファの内容が消えてしまい，ファイル書込み結果が実際のファイルに反映されないのでファイルの信頼性が問題になる．

そこでファイル書込みにおいて，確実に磁気ディスクに書込みを実施しておきたい場合には，ライトバック（Write Back）要求をシステムコール sync( ); で行う必要がある．このとき，パラメータは不要であり，また返り値もない．本システムコールと同じ働きをするのが，コマンドの sync である．

sync( ) では，スーパブロック，i-node，ファイルの実体のすべてを書き込み，システムバッファ内に書込みのあった部分のすべてを磁気ディスクに書き戻す．システムバッファには定期的に書込みが行われ，システムバッファ領域に空きブロックが少なくなったときに解放処理として書込みがなされるが，このシステムコールではユーザが明示的に書込みを行うことになる．したがって，不用意に sync( ) を多発するとシステムの性能劣化要因になるので使用には十分注意が必要である．

(8)　ファイルを消去する：unlink

ファイルの消去はコマンドでは rm（Remove）である．この機能をプログラムで実行する

システムコールが unlink( ) であり，仕様は**図 6･25** のとおりである．

```
int unlink(pathnm)  /* 削除するファイル名*/
char *pathnm;       /* パス名を指定 */
/* 成功の場合は０，失敗は -1 */
```

図 6・25　システムコール unlink **の仕様**

unlink( ) は表 6･2 に示した i-node 内のリンクカウント st_nlink を-1 する．その結果，ゼロになるとファイルの実体を原則として解放する．

(9)　シンボリックリンクを作る：link

ファイルを共用するには link コマンドを使う．リンクはディレクトリに新しいエントリを作成して，同一の i-node 番号を共有することである．これをプログラムでシステムコールを用いる場合は link( ) を実行させる．**図 6･26** にその仕様を示す．

```
int link(opathnm,npathnm)  /* ファイル共用のリンク*/
char *opathnm;       /* 既存のパス名を指定 */
char *npathnm;       /* 新ディレクトリを指定 */
/* 成功の場合は０，失敗は -1 */
```

図 6・26　システムコール link **の仕様**

(10)　ファイルチェック：access

あるディレクトリにプログラム内からファイルを作成しようと思うとき，事前に可能か否かの判定を行わなくてはならない場合が生じる．このようなときに，アクセスのチェックを行うシステムコールが access( ) である．

仕様は**図 6･27** に示すとおりである．第 1 パラメータはパス名である．第 2 パラメータの基本は，アクセスの 9 ビットである．この場合，許可されているか否かのビット部分を指定する．そのすべてのビットで許可されているならば，返り値は 0 であり，それ以外は-1 である．

```
int access( path, pattern);
char *path;       /* 既存のパス名を指定 */
int pattern;      /* アクセス許可ビットを指定 */
/* 成功の場合は０，失敗は -1 */
```

図 6・27　システムコール access **の仕様**

■7 群 － 3 編 － 6 章

6-5 演習問題

（執筆者：吉澤康文）［2013 年 2 月 受領］

(1) UNIX のファイルには 3 種類あるがそれらを列挙し説明せよ．

(2) UNIX では stdout と stderr のデフォールトが端末になっているが，どうして 2 種類あるのか．

(3) 絶対パス名と相対パス名を設けた理由と利点を述べよ．

(4) UNIX のファイルキャッシュ（あるいはバッファキャッシュ）について以下の問に答えよ．
- 目的は何か．
- 欠点は何か．
- どのような管理を行うのか．
- 関連するシステムコールにどのようなものがあるか．

(5) UNIX のファイル入出力では入出力が完了するまでプロセスがブロックされる同期入出力であるが，その利点と欠点を述べよ．

(6) ファイル入出力の非同期処理を行うことでメリットのです応用をあげ説明せよ．

(7) 端末（stdin）から入力した文字列を自分のホームディレクトリのファイルとして格納するプログラムを作成せよ．

(8) 複数行からなるファイルとし，そのファイル名は任意とする．端末から指定．

(9) 上記作成したファイルを読み込み，端末（stdout）に出力するプログラムを作れ．

(10) 二つのファイルを作成し，一方のファイルをもう一方のファイルの最後に付加えるプログラムを作れ．

(11) 任意のディレクトリ（自分の home dir. がよい）を読込み，
- ディレクトリ内情報を表示せよ．
- stat/fstat などのシステムコールを用いてファイル情報を極力詳しく出力せよ．

(12) 10 バイト以上の容量をもつファイルが既に open されていたとする {fd=open("file_name",1);}．このとき，ファイルの最後から 10 バイトを読み込むプログラムを書け．