第2章

オペレーティングシステムの構造

＜略＞

2.3　システムコール

システムコール（system call）は，オペレーティングシステムにより利用可能となったサービスへのインタフェースを提供する．これらの呼び出しは，C や C++ で書かれたルーチンとして一般的に利用できるが，いくつかの低レベルの作業（たとえば，ハードウェアに直接アクセスしなければならない作業）はアセンブリ言語命令で書かなければならないかもしれない．

オペレーティングシステムがシステムコールを利用可能にする方法について述べる前に，まず，システムコールがどのように利用されるのかを例で示す．例は，あるファイル中のデータを読み込み，それを別のファイルにコピーするという単純なプログラムを書くというものである．プログラムが必要とする最初の入力は，二つのファイル（入力ファイルと出力ファイル）の名前である．これらの名前は，オペレーティングシステムの設計により，いろいろな方法で指定できる．一つは，プログラムが二つのファイルの名前をユーザに聞く方法である．インタラクティブシステムでは，この方法は一連のシステムコールを必要とする．まず，画面上に入力を促すメッセージを書き込み，次に二つのファイルを定義する文字をキーボードから読み込む．マウスやアイコンに基づいたシステムでは，ファイル名からなるメニューが，通常，ウィンドウに表示される．そして，ユーザはマウスを使ってコピー元のファイル名を選び，さらにコピー先の名前を指定するためにウィンドウを開く．この一連の操作は多くの入出力システムコールを必要とする．

いったん二つのファイル名を取得したら，プログラムは入力ファイルを開き，出力ファイルを生成しなければならない．それぞれの操作は，別のシステムコールを必要とする．各操作について，エラーを引き起こす条件もあるかもしれない．入力ファイルをオープンしようとするとき，そのような名前のファイルがないとか，そのファイルがアクセスから保護されていることがわかるとかするかもしれない．これらの場合，プログラムはコンソール上にメッセージを出力し（別の一連のシステムコール），それから異常状態で終了（これもまた別のシステムコール）すべきである．入力ファイルが存在すれば，新しい出力ファイルを生成しなければならない．同じ名前の出力ファイルがすでにあることがわかるかもしれない．この場合，プログラムを異常終了（システムコール）するか，あるいは現在あるファイルを削除（別のシステムコール）して新しいファイルを生成（別のシステムコール）するかもしれない．もう一つの選択肢として，インタラクティブシステムでは，ユーザに（入力を促すメッセージを出力し，端末から返答を読み込む一連のシステムコールを通して）現在あるファイルを置き換えるか，プログラムを異常終了させるか聞くことができる．

ここで両ファイルが準備されたので，入力ファイルから読み込み（システムコール），出力ファイルに書き込む（別のシステムコール）というループに入る．それぞれの読み書きは，様々なあ

りうるエラーの条件についての現状の情報を返さなければならない．入力では，プログラムはファイルの最後に達したり，読込みの際（パリティエラーなど）ハードウェアの故障があったりしたことを検出するかもしれない．書込みの操作では，出力デバイスによって，（ディスク容量不足，プリンタ用紙の不足など）様々なエラーに遭うかもしれない．

最後に，ファイル全体がコピーされたら，プログラムは両方のファイルを閉じ（別のシステムコール），コンソールまたはウィンドウへメッセージを書き出し（また別のシステムコール），そして最後に正常終了（最後のシステムコール）する．見てのとおり，簡単なプログラムでもオペレーティングシステムを頻繁に使用する．しばしば，システムは1秒に何千ものシステムコールを実行する．この一連のシステムコールを図2.1に示す．

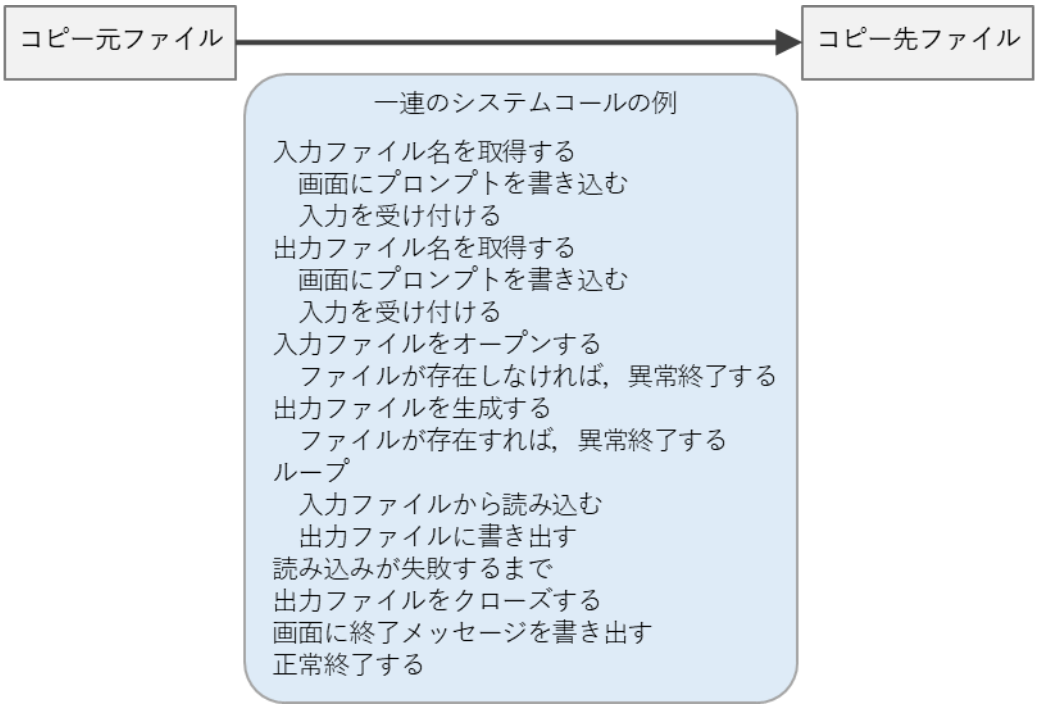


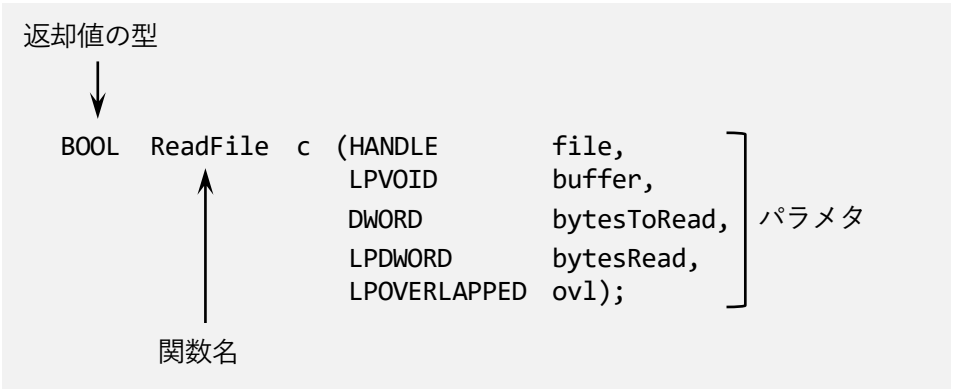
図2.1　システムコールの使用例

しかしながら，ほとんどのプログラマはこのレベルまでの詳細を見ることはない．一般に，アプリケーションの開発者は**アプリケーションプログラミングインタフェース**（API：Application Programming Interface）に基づいてプログラムを設計する．APIは，アプリケーションプログラマが利用できる関数の集合を指定する．これには，各関数へ渡されるパラメタと，プログラマが期待できる返戻値などが含まれる．アプリケーションプログラマが最もよく使う三つのAPIはWindowsシステム用のWin32 API，（UNIX，Linux，Mac OS Xのほとんどすべてのバージョンを含む）POSIXに基づいたシステムのための POSIX API，そして Java仮想機械上で実行するプログラムを設計するための Java APIである．

本書で用いるシステムコールの名前は汎用的な例であることに注意しよう．各々のオペレーティングシステムは，各々のシステムコールに対して独自の名前を付けている．

コラム － 標準的な API の例

標準的な API の例として，Win32 API における `ReadFile()` 関数（ファイルから読み込むための関数）を考える．この関数の API を下図に示す．



`ReadFile()` 関数の API

`ReadFile()` に渡されるパラメタは次のとおりである．

- `HANDLE file`：読み込まれるファイル．
- `LPVOID buffer`：バッファであり，そこにデータが読み込まれ，そこから書き出される．
- `DWORD bytesToRead`：バッファに読み込まれるバイト数．
- `LPDWORD bytesRead`：最後の読込みで読み込まれたバイト数．
- `LPOVERLAPPED ovl`：オーバラップされた入出力が使われているかを示す．

API を構成する関数は，アプリケーションプログラマの代わりに，実際のシステムコールを通常呼び出す．たとえば，Win32 の関数 `CreateProcess()`（これは，予想どおり新しいプロセスを生成するために用いられる）は，実際は，Windows カーネル内にある `NTCreateProcess()` システムコールを呼び出す．なぜアプリケーションプログラマは，実際のシステムコールを呼び出すのではなく，API に基づいてプログラムすることを好むのか？ これにはいくつかの理由がある．API に基づいたプログラミングの一つの利点はプログラムの移植性である．API を用いてプログラムを設計するアプリケーションプログラマは，同じ API を支援しているどのようなシステムにおいても，そのプログラムをコンパイルし実行できると期待できる（ただし，実際のところ，アーキテクチャの違いが見かけよりももっと難しくしている）．さらに，実際のシステムコールは，アプリケーションプログラマが利用できる API よりも細かく，しばしば使うのが難しい．いずれにしても，API の関数の呼出しと，カーネル内の関連付けられたシステムコールの間には，強い相関関係がよく存在する．実際，多くの POSIX と Win32 の API は，UNIX，Linux，Windows オペレーティングシステムが提供するもともとのシステムコールに似ている．

ほとんどのプログラミング言語のための実行時支援システム（コンパイラに含まれているライブラリの組込み関数群）は，オペレーティングシステムにより利用可能となっているシステムコールへのリンクとして働く**システムコールインタフェース**（system-call interface）を提供する．システムコールインタフェースは API 内の関数呼出しを捉え，オペレーティングシステム内の必要なシステムコールを呼び出す．通常，各システムコールには数字が割り当てられており，システムコールインタフェースはこの数字で索引付けを行ったテーブルを持っている．そして，シス

テムコールインタフェースは，オペレーティングシステムカーネル内で意図されたシステムコールを呼び出し，システムコールの状況と返戻値を返す．

呼び出し側は，システムコールがどのように実装されているかとか，実行中に何をするかとかは知る必要がない．API に従い，そのシステムコールを実行した結果，オペレーティングシステムが何をするのかを理解するだけでよい．そこで，オペレーティングシステムインタフェースのほとんどの詳細は，API によりプログラマから隠されており，実行時支援ライブラリにより管理されている．API，システムコールインタフェース，オペレーティングシステムの間の関係を図 2.2 に示す． open() システムコールを呼び出しているユーザアプリケーションをオペレーティングシステムがどのように扱っているかを示している．

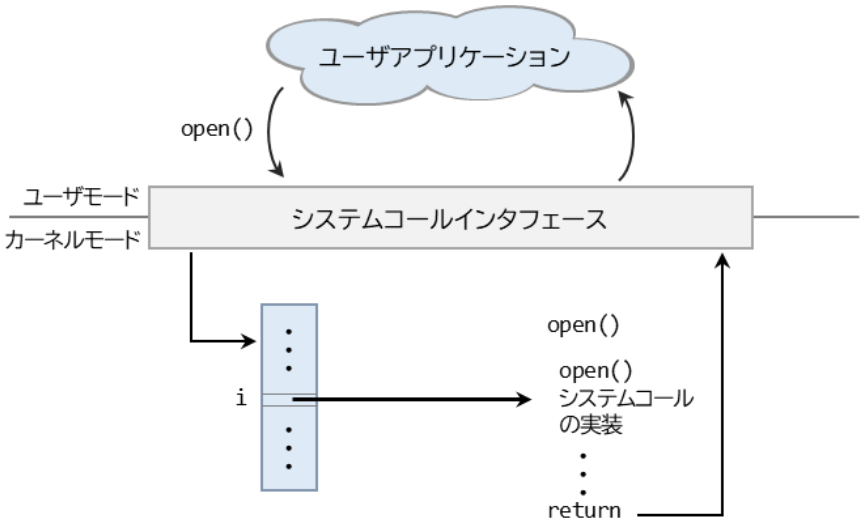


図 2.2　open() システムコールを呼び出しているユーザアプリケーションの扱い

システムコールは，利用しているコンピュータにより，いろいろな方法で生じる．望んでいるシステムコールの識別子よりも多くの情報がしばしば必要となる．情報の厳密な種類と量は，特定のオペレーティングシステムと呼出しにより異なる．たとえば，入力を得るためには，入力元として使用するファイルやデバイスを指定しなければならず，さらに入力が読み込まれるメモリバッファのアドレスと長さも必要になるかもしれない．もちろん，デバイスやファイル，そして長さは呼出しで暗黙的に決まっているかもしれない．

補足

コラムの標準 API の例は Win32 API で身近なものではないかもしれませんので、参考書原著第 9 版（参考書は第 7 版の翻訳）での UNIX や Linux に対応する記述を挙げておきます

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value　function name　parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

また、講義資料 p.13 のライブラリ関数の説明で「C言語のライブラリ関数 fopen, fread などは内部で open, read などを呼出す」がありますが、この説明の図として、参考書の別コラムを以下に挙げます。

標準 C ライブラリは，UNIX と Linux の多くのバージョンのためのシステムコールインタフェースの一部を提供する．例として，C プログラムが `printf()` 文を呼び出すとしよう．C ライブラリはこの呼出しを捕捉し，オペレーティングシステム内の必要なシステムコール（この場合，`write()` システムコール）を呼び出す．C ライブラリは `write()` により返された値を受け取り，ユーザプログラムに渡す．これを下図に示す．