第2章

オペレーティングシステムの構造

＜略＞

2.7 オペレーティングシステムの構造

現代のオペレーティングシステムほどの大規模で複雑なシステムが正しく機能し，容易に変更できるためには，注意深く開発されなければならない．一般的なアプローチでは，一枚岩の（モノリシック（monolithic）な）一つのシステムにするのではなく，仕事を小さな構成要素に分割する．これらの各モジュールは，注意深く定義された入力，出力，関数を持つ，明確に定義されたシステムの一部分であるべきである．第１章でオペレーティングシステムの共通の構成要素について述べた．本節では，これらの構成要素がどのように相互に結合され，カーネルに併合されるのかについて述べる．

2.7.1 単純構造

多くの商用システムは，明確に定義された構造をもっていない．そのようなオペレーティングシステムは，しばしば小規模な単純で限定されたシステムとして始まり，元の限界を超えるほどに成長していった．MS-DOSはこのようなシステムの例である．もともと，そんなに人気が出るとは考えずに，数人によって設計され，実装された．最小限の空間で最大の機能が提供できるように書かれたため，注意深くモジュールに分けられなかった．図2.6にその構造を示す．

MS-DOSでは，インタフェースと機能のレベルはうまく分離されていない．たとえば，アプリケーションプログラムは，基本的な入出力ルーチンにアクセスして，ディスプレイやディスクドライブに直接書き込むことができる．このような自由は，間違った（または悪意のある）プログラムに対してMS-DOSを無防備な状態にしているので，ユーザプログラムが故障したとき，システム全体がクラッシュする．もちろん，MS-DOSはその時代のハードウェアにも制限されていた．対象にしていた Intel 8088にはデュアルモードやハードウェアの保護がなかったため，MS-DOSの設計者は基本のハードウェアをアクセス可能にするしか方法がなかった．

限定された構造の別の例として，最初のUNIXオペレーティングシステムがある．UNIXも，ハードウェアの機能に当初は制限を受けたシステムである．UNIXは，カーネルとシステムプログラムの二つの分離可能な部分からなる．カーネルは，さらに一連のインタフェースとデバイスドライバに分離され，これらはUNIXが長い年月を重ねて発展するにつれて，追加され拡張されてきた．伝統的なUNIXオペレーティングシステムは，図2.7に示すように，層構造と見ることができる．システムコールインタフェースよりも下で，物理ハードウェアよりも上の部分は，すべてカーネルである．カーネルはファイルシステム，CPUスケジューリング，メモリ管理，そしてその他のオペレーティングシステムの機能を，システムコールを通して提供する．総じて，一つのレベルに組み合わせるには膨大な量の機能である．このモノリシックの構造を実装し，保守することは難しかった．

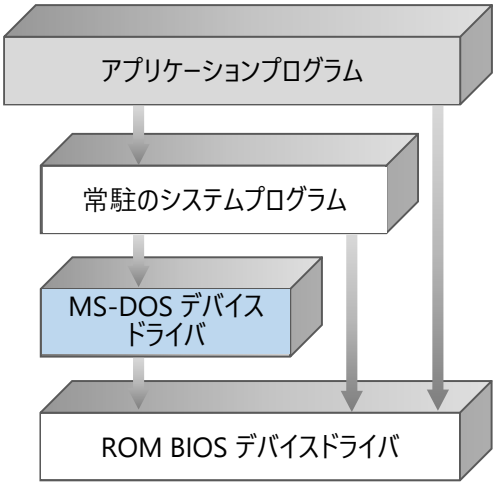

図2.6　MS-DOSの構造

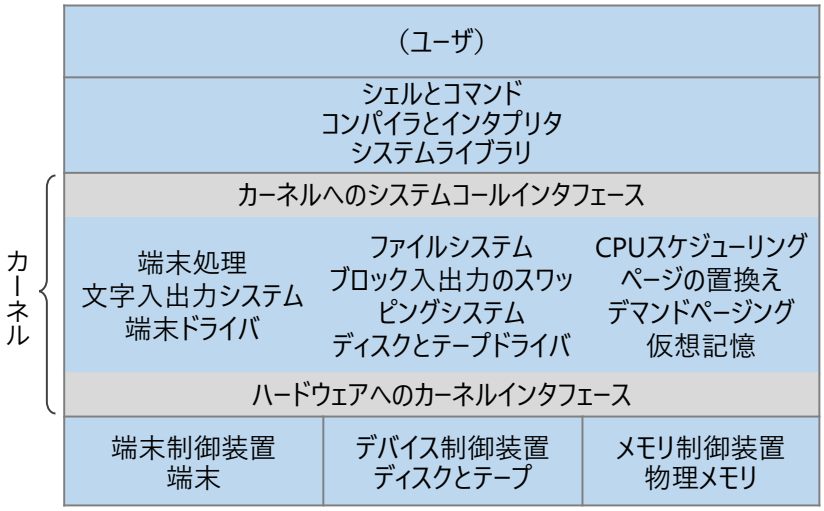

図2.7　UNIXのシステム構造

2.7.2 層アプローチ

適切なハードウェアの支援があれば，オペレーティングシステムは，最初のMS-DOSやUNIXシステムなどよりも，より小さくより適切な部品に分割することができる．オペレーティングシステムは，これによりコンピュータとコンピュータを利用するアプリケーションをより制御することができる．実装する人は，システムの内部の変更や，モジュール化されたオペレーティングシステムの作成を，より自由に行うことができる．トップダウンアプローチでは，全体的な機能や特徴が決定され，構成要素に分けられる．情報隠蔽も重要である．というのは，ルーチンの外部インタフェースが変更されず，ルーチンそのものが公開されている仕事を実行する限り，情報隠蔽により，プログラマは低レベルのルーチンを好きなように実装できるからである．

システムはいろいろな方法でモジュール化できる．一つの方法は**層アプローチ**（layered approach）であり，オペレーティングシステムをいくつかの層（レベル）に分ける．最下層（層０）はハードウェアであり，最上位（層N）はユーザインタフェースである．この層構造を図2.8に示す．

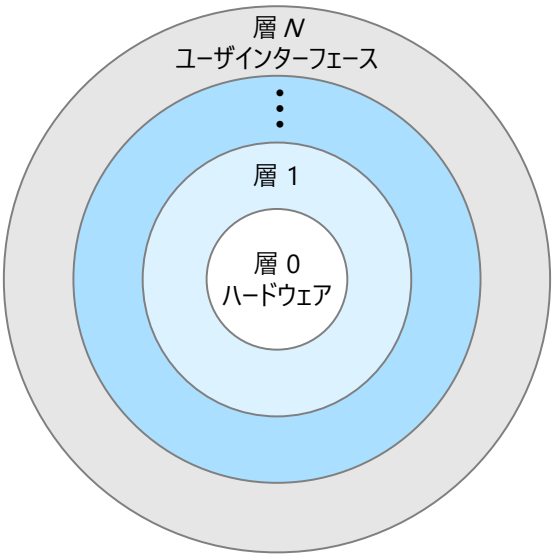

図2.8　層構造をしたオペレーティングシステム

オペレーティングシステムの層は，データと，そのデータを操作できる命令からなる抽象オブジェクトの実装である．典型的なオペレーティングシステムの層（たとえば層M）は，データ構造と，より高いレベルの層から呼び出せる一連のルーチンからなる．層Mは，それよりも低い層の命令を呼び出すことができる．

層アプローチの主要な利点は，構築とデバッギングの容易さにある．各層がより低いレベルの層にある関数（命令）やサービスだけを用いるように，各層は選ばれる．このアプローチは，デバッギングとシステムの検証を単純にする．最初の層は，システムの残りを考えずにデバッグできる．というのは，定義より，（正しいと仮定する）基本ハードウェアのみを用いて，その機能が実装されているからである．最初の層がデバッグされれば，それが正しく機能することを仮定でき，続いて第2層をデバッグでき，という具合に実装できる．ある層をデバッグしているときにエラーが見つかったら，その層よりも下はすでにデバッグされているため，エラーはその層にあるに違いない．そこで，システムの設計と実装が単純になる．

各層は，より低い層が提供する命令によってのみ実装される．層は，これらの命令がどのように実装されているかを知る必要はない．これらの層が何をするかのみを知る必要がある．そこで，各層はデータ構造，命令およびハードウェアの存在を，より高いレベルの層から隠す．

層アプローチの主要な難点は，様々な層を適切に定義することである．各層は，下位レベルの層しか使用できないため，注意深い計画が必要となる．たとえば，バッキングストア（backing store．仮想記憶アルゴリズムにより使用されるディスク空間）のためのデバイスドライバは，メモリ管理ルーチンよりも低いレベルになければならない．というのは，メモリ管理はバッキングストアを使うことができなければならないからである．

他の要求はここまで明白ではないかもしれない．バッキングストアのドライバは，通常，CPUスケジューラよりも上にある．というのは，ドライバが入出力を待たなければならないかもしれず，CPUはこの間に再スケジュールできるからである．しかしながら，大規模なシステムでは，CPUスケジューラが持っているすべてのアクティブなプロセスに関する情報が，メモリに入る量よりも大きいかもしれない．したがって，この情報をメモリからスワップインしたり，スワップアウトしたりする必要があるかもしれないので，バッキングストアのドライバルーチンはCPUスケジューラよりも下に位置付けなければならない．

層の実装の最後の問題は，他の方法に比べて効率が良くない傾向があることである．たとえば，ユーザプログラムが入出力命令を実行するとき，そのプログラムは入出力層により捕獲されるシステムコールを実行する．すると，それはメモリ管理層を呼び出し，メモリ管理層がさらにCPUスケジューリング層を呼び出し，そして最後にハードウェアに渡される．各層において，パラメタが変更されるかもしれないし，データを渡す必要があるかもしれない．各層はシステムコールに対してオーバヘッドを増やす．最終結果として，システムコールは，層化されていないシステムの場合よりも時間がかかる．

これらの制約は，近年，層アプリケーションに対して小さな反発を引き起こした．より多くの機能を持つ，より少ない層を設計することで，モジュール化コードのほとんどの利点を提供しながらも，層の定義や相互作用の難しい問題を避けるのである．

2.7.3 マイクロカーネル

UNIXが拡張されるにつれ，カーネルが大きくなり，管理が難しくなったことを見てきた．1980年代半ば，カーネギーメロン大学（CMU：Carnegie Mellon University）の研究者たちが，**マイクロカーネル**（microkernel）アプローチを用いてカーネルをモジュール化したMachというオペレーティングシステムを開発した．この手法は，カーネルから必須ではないすべての構成要素を取り除き，それらをシステムやユーザレベルのプログラムとして実装することにより，オペレーティングシステムを構成した．結果としてカーネルが小さくなった．どのサービスをカーネルに残し，どれをユーザ空間で実装すべきかについての意見の一致はほとんどない．しかしながら，典型的には，マイクロカーネルは通信機能と，最低限のプロセスとメモリ管理を提供する．図2.14に典型的なマイクロカーネルの構造を示す。

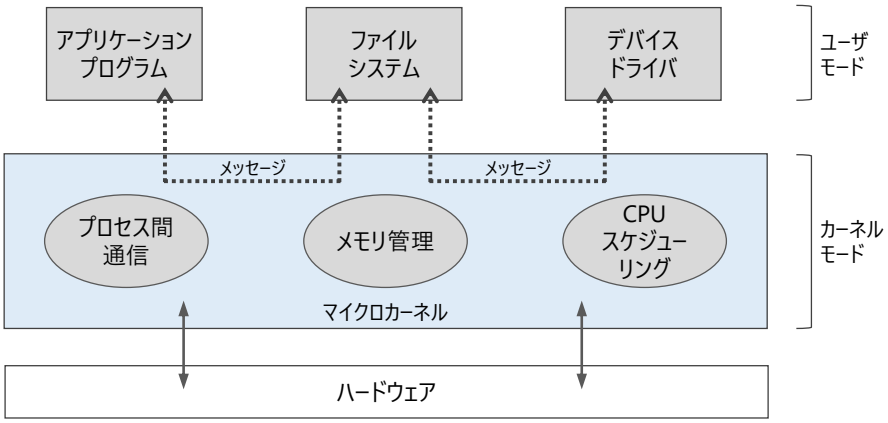

図2.14　典型的なマイクロカーネルの構造

マイクロカーネルの主要な機能は，クライアントプログラムとユーザ空間で実行している様々なサービスとの間の通信メカニズムを提供することである．通信は**メッセージパッシング**（message passing）によって提供される．これについては2.4.5項で述べた．たとえば，クライアントプログラムがファイルにアクセスしたい場合，ファイルサーバと相互に作用しなければならない．クライアントプログラムとサービスとが，直接，相互作用することは決してない．マイクロカーネルとメッセージを交換することにより，間接的に通信するのである．

マイクロカーネルアプローチの一つの利点は，オペレーティングシステムの拡張のしやすさである．すべての新しいサービスはユーザ空間に加えられるので，その結果，カーネルは変更する必要

がない．カーネルを変更しなければならないとしても，マイクロカーネルはより小さなカーネルであるため，変更は少ない傾向にある．結果として，設計が異なるハードウェア間でオペレーティングシステムを移植することが容易になる．ほとんどのサービスが，カーネルではなくユーザプロセスとして実行されるため，マイクロカーネルはまた，セキュリティと信頼性がより高い．あるサービスができなくなっても，オペレーティングシステムの残りの部分は無傷のままである．

現代のオペレーティングシステムのいくつかは，マイクロカーネルアプローチをとっている．Tru64 UNIX（以前のDigital UNIX）は，ユーザにUNIXのインタフェースを提供するが，Machカーネルで実装されている．Machカーネルは，UNIXのシステムコールを適切なユーザレベルサービスへのメッセージに変換する．

別の例として，QNXがある．QNXはリアルタイムオペレーティングシステムであり，これもマイクロカーネル設計に基づいている．QNXのマイクロカーネルは，メッセージパッシングとプロセススケジューリングのためのサービスを提供する．また，低レベルのネットワーク通信やハードウェアの割込みも扱う．他のすべてのQNXのサービスは，カーネルの外でユーザモードで実行する標準的なプロセスによって提供される．

残念ながら，システム機能のオーバヘッドが増えるので，マイクロカーネルは性能が低下するかもしれない．Windows NTの歴史を考える．最初のリリースは，層構造のマイクロカーネルであった．しかしながら，このバージョンはWindows 95に比べて性能が低かった．Windows NT 4.0は，いくつかの層をユーザ空間からカーネル空間に移し，それらをより緊密に結合することにより，性能の問題を部分的に改善した．Windows XPが設計されたころには，そのアーキテクチャはマイクロカーネルというよりはモノリシックなものであった．

2.7.4 モジュール

おそらく，オペレーティングシステムの設計のための最善の方法論は，オブジェクト指向プログラミングの手法を用いて，モジュール化されたカーネルを作成することである．ここで，カーネルは一連の中核的な構成要素を持ち，ブート時または実行時に追加のサービスへ動的にリンクする．動的にロード可能なモジュールを利用するこのような戦略は，Solaris，Linux，Mac OSといったUNIXの現代の実装で一般的である．

＜以下略＞

注意
原文での「システム呼出し」を「システムコール」としています。「一枚岩の」などは「モノリシックな」などとしています。
また、原著第 9 版の対応部分にある図 2.14 を引用して追加しています。