パーソナルコンピュータ用
オペレーティングシステムの現状と問題点

大城　正典　菊池　豊彦

概要

近年、ハードウェアの低価格化・高機能化の勢いは著しく、パーソナルコンピュータは非常に身近な情報機器となった。また、インターネットの普及によって、情報サービスの入り口としてのパーソナルコンピュータの役割も大きくなりつつある。しかし、パーソナルコンピュータ用のオペレーティングシステム（OS）は、多様かつ高度な機能を要求されて複雑化・肥大化しつつあり、いくつかの問題を抱えている。本稿では、パーソナルコンピュータ用OSの基本的な構造と役割を改めて見直し、現状における問題点として、セキュリティの問題、共有ライブラリに関する問題、国際化に関する問題、GUIに関する課題、処理データの増大にともなう問題、OSの複雑化・肥大化に関する問題を考察する。

１．はじめに

今日、パーソナルコンピュータは一般家庭、個人にも広く普及しており、パーソナルコンピュータ用OSは、ユーザの多様な要求に応えている。しかし、OS自体は複雑化・肥大化し、いくつかの問題も生じ始めている。本稿では、情報システムの中核を担うパーソナルコンピュータ用OSの基本構造と機能を見直して、現状における問題点を考察する。

２．OSの構造と役割の概要

OSとは、基本ソフトウェアとも呼ばれ、ハードウェアを制御し、コンピュータの基本的な動作を可能にするソフトウェアということができる。狭義のOSをカーネルと呼ぶ。図１は、この観点からコンピュータシステムの構成を見た概念図である。コンピュータシステムを構成するハードウェアは、それを制御するファームウェアと呼ばれる小規模なソフトウェアによって制御することができる。一般にファームウェアはハードウェア自体に内蔵されている。OSは、ドライバと呼ばれるソフトウェアを介してハードウェアを制御する。ドライバは、ファームウェアを介してハードウェアを制御する。

このようにすることで、OSはハードウェアを間接的に制御し、動作させて、アプリケーションソフトウェア（ユーザが直接使用するソフトウェア）を実行することのできる環境を提供している。アプリケーションソフトウェアの作成者は、ハードウェアを直接制御するための処理を記述しなくてよいので、アプリケーションソフトウェアの作成コストが低減するという利点もある。アプリケーションソフトウェアが意図的にハードウェアを制御する必要がある場合は、OSの用意した機能

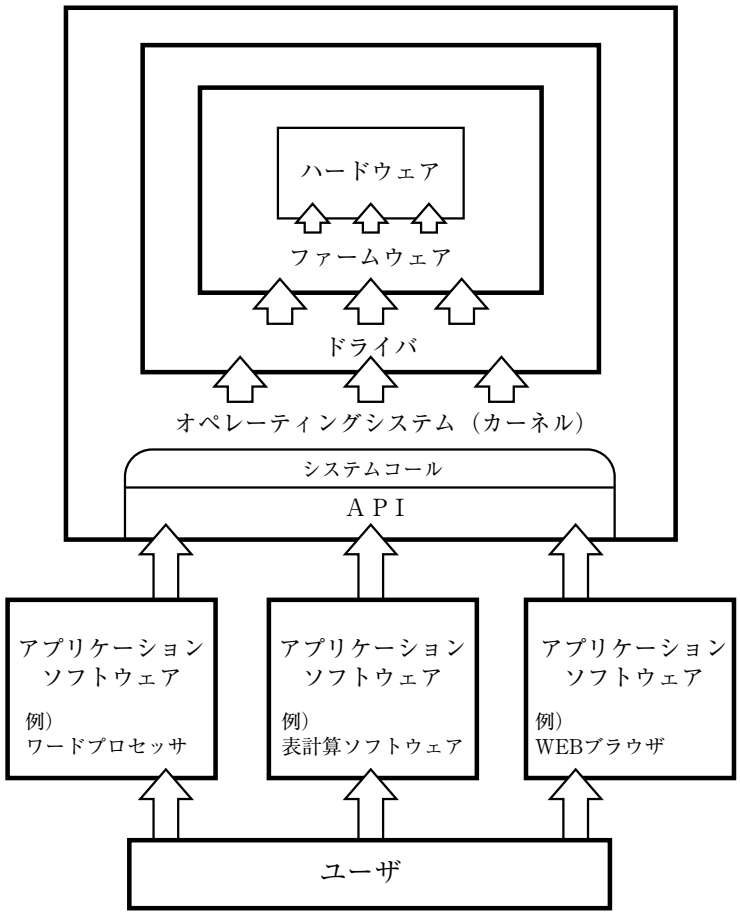


図1　コンピュータシステムの構成

を利用することができる。OSがアプリケーションソフトウェアに公開している機能を、API（Application Programming Interface）と呼ぶ。APIは、カーネルが外部に提供しているより低レベルな機能であるシステムコール（カーネルコール）をより使いやすい形にしたものである。APIはハードウェアを制御する機能の他にも、データベース処理を行う一連の機能や、テキスト処理などの高度なサービスも提供する。また、ハードウェアの制作者はファームウェアを内蔵した形で機器を出荷し、その機器を利用する複数のOSごとにドライバを用意することで、複数の異なるOSに機器を対応させることができる。

なお、ドライバはコンピュータシステム動作時には、OSに組み込まれる。また、OSのパッケージには、代表的な機器のドライバが同梱されるので、ドライバを含めて、広義のOSと考えることができる。パーソナルコンピュータにおいては、市販されている様々な外部機器を使用できることが望ましいため、コンピュータの実行中に接続された機器用のドライバを、動的に組み込む機構を備えることがOSに求められている。

その他、OSに求められる基本的役割は、コンピュータシステムに関わる各種の資源（動作時間、メモリやハードディスクなどのハードウェア資源、プログラムなどのソフトウェア資源、人的資源など）を効率よく配分、管理することであると言える。

ユーザの立場からは、パーソナルコンピュータ用OSは扱いやすく、安定して動作し、安全でセキュリティ性が高くなくてはならない。また、開発者にとっては、APIや構造が把握しやすく、プログラムの開発と保守が容易でなくてはならない。

3．OSの主要機能

本稿では、パーソナルコンピュータ用OSについての現状と問題点を考察するが、それらの問題点は当然ながらOSの主要機能に関わるものである。OSを機能的に分解すると、プログラムの実行制御、メモリ管理、ディスク管理、入出力制御、プロセス管理などに分けられる。その他、マルチユーザ管理やグラフィカルユーザインタフェイス（GUI)、多言語・国際化サービスも今日のパーソナルコンピュータ用OSには欠かせない機能である。問題点の分析に先立ち、まずこれらの各機能（ディスク管理と入出力制御を除く）について概要を説明する。

◎3.1 プログラム実行制御

OSの持つ機能のうち、もっとも重要なものがプログラムの実行制御である。プログラムをどのように作成し、実行するかを説明し、さらに動的リンクと共有ライブラリについて解説する。

○3.1.1 プログラムの作成

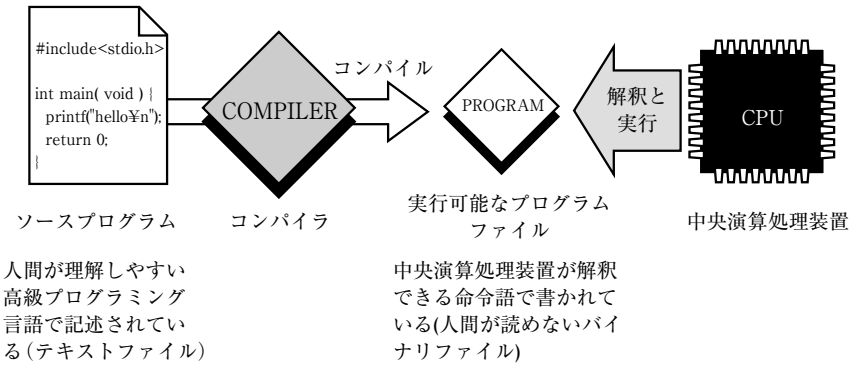
プログラムはCPU用の命令語の集まりであるが、人間が命令語（プログラミングの用語では機械語と言われる）で直接記述することは困難である。そのため、特定の文法に基づいた人間にわかる言語によってプログラムを記述するのが普通である。これをプログラミング言語と言う。命令語を人間がわかる形に直したアセンブラ言語もあるが、大きなプログラムを記述するには作成者に対する負担が大きい。そこで、高級プログラミング言語と呼ばれる、より人間の理解しやすい形で記述できるプログラミング言語が開発された。代表的な高級プログラミング言語には、C言語、C++、Java、Pascal、Fortran、Cobol、Basic、Perl、Rubyなどがあげられる。

プログラミング言語で記述されたソースプログラムは、一連の命令語に変換（翻訳）され、それがCPUによって解釈・実行される。ソースプログラムから命令語への翻訳方式には、コンパイラというプログラムでソースプログラムを一度に命令語に変換（コンパイルと言う）して別のファイルとして保存するコンパイラ形式と、インタプリタと呼ばれるプログラムによって逐次ソースプログラムを機械語に変換しながら実行するインタプリタ形式がある（図２）。

以降は、コンパイラ言語を例に、プログラムの作成の概要について述べる。コンパイラ言語で書かれたソースプログラムは、前述したようにコンパイラによって命令語で構成されたプログラム、すなわち実行可能ファイルに変換される。この実行可能ファイルは、ロードモジュールとも呼ばれる。大規模なプログラムを作成する場合、ひとつのソースプログラムでそのプログラムの全てを記述するわけではない。通常、プログラムの設計段階で、見通しをよくするためにプログラム全体を機能別に分割する。この分割単位をモジュールと呼ぶ。また、単独のモジュールまたは複数個の関連するモジュールの集合をライブラリと呼ぶ。

通常、プログラム全体をモジュールに分割する際には、基本的な機能を提供する下位モジュールと、下位モジュールの機能を使って応用的な機能を提供する高位モジュールを設ける（図３）。モジュールに分割すると、機能的な面での見通しが良くなるだけでなく、開発をモジュール単位で

(1) コンパイラ言語プログラムの実行のされ方



(2) インタプリタ言語プログラムの実行のされ方

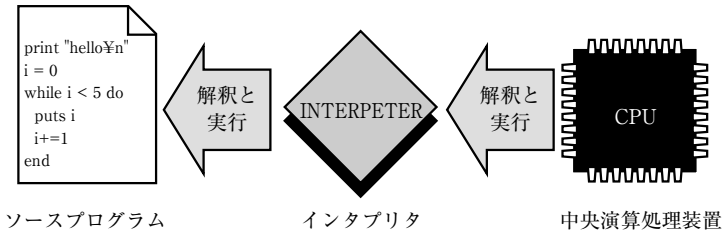


図2　コンパイラ言語とインタプリタ言語

別々に行えるという利点がある。プログラム全体が完成していなくとも、モジュール単位で動作テストなどを行える。モジュールの利用の仕方と外見的な振る舞い（モジュールインタフェイス）を定義しておけば、モジュールの中身を実際にどのように実装するかは自由である。利用者はモジュールインタフェイスだけを参考にモジュールを使用するからである。そのため、モジュールの内部は他のモジュールからは隠蔽されていることになり、モジュール間の依存性が減り、各モジュールの独立性が向上する。モジュールが十分な独立性を持ち、汎用的に設計されていれば、別のプログラムを作成する際にそのモジュールを再利用できるので、プログラムの開発コストを削減できる。

プログラミング言語を使ってプログラムを作成するときは、各モジュールに対応したソースプログラムを個々に作成する（図４）。モジュールは他のモジュールの機能やデータを利用している。たとえば図４の例では、モジュールＡはモジュールＢの機能ｆとモジュールＣのデータｄを利用している。このように、他のモジュールを参照・利用していることを外部参照と呼ぶ。それぞれのソースプログラムをコンパイルすると、オブジェクトコードと呼ばれるファイルが生成される。オブジェクトコードは、ロードモジュールを構成する部品であって、単独では実行することはできない。たとえば図４のオブジェクトモジュールＡは、機能ｆとデータｄを利用するが、これらはオブジェクトモジュールＡの中には存在せず、プログラムとして実行するには不完全な状態である。したがって、しかるべきモジュール群を結びつけ、ひとつのロードモジュールに結合する作業が必要となる。この作業を“リンクする”と言い、リンクを行うプログラムをリンカまたはリンケージエディ

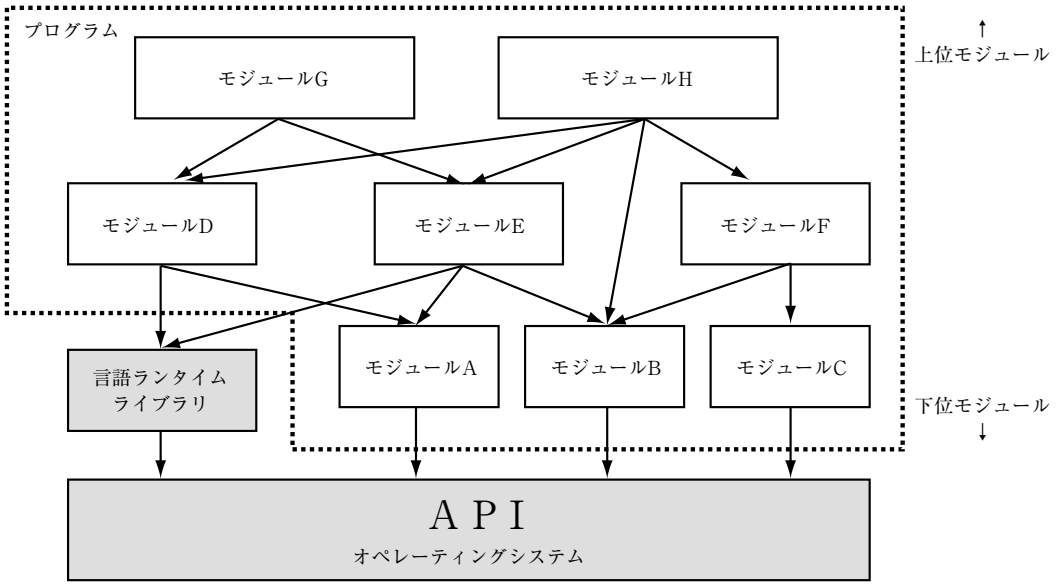


図3　モジュールの集合体としてのプログラム

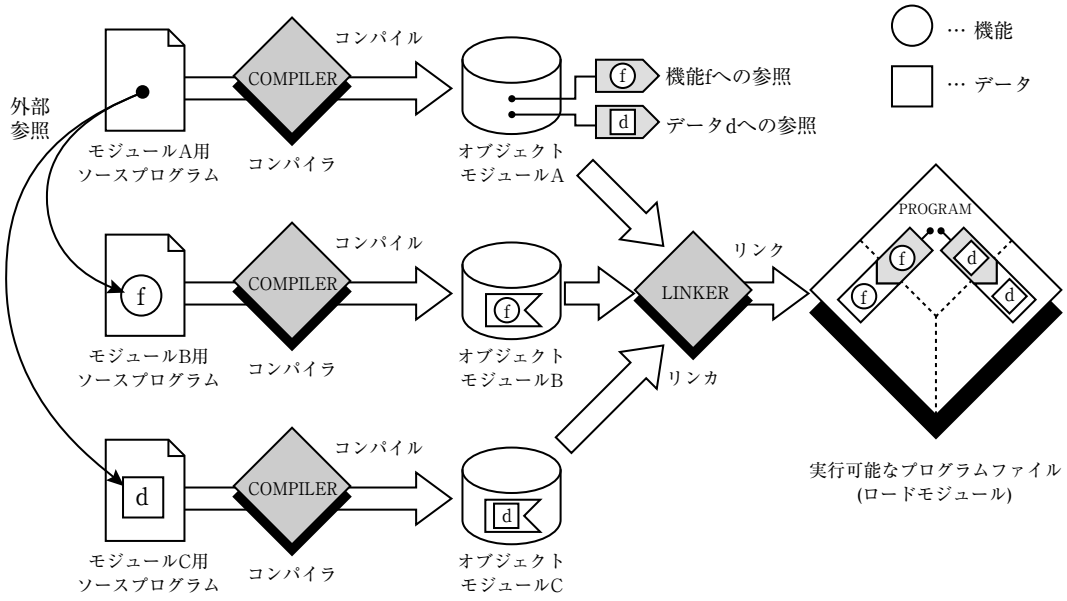


図4　外部参照とリンカの役割

タと呼ぶ。リンカは各オブジェクトモジュールの外部参照を参照先の実体がある場所を指すように調整し、オブジェクトモジュール群を単一のロードモジュール（もしくはオブジェクトモジュールの集合体であるライブラリ）に統合する。

○3.1.2 プログラムの実行

このように作成されたロードモジュールは、ローダと呼ばれるプログラムによってメモリ上に読み込まれ、配置される。このとき、ローダはまずロードモジュールを読み込むのに必要な領域をメモリ上に確保する。そしてその確保した領域にロードモジュールを読み込む。次にロードモジュールファイル上にあったデータやルーチンの位置関係を、メモリ上の位置関係、すなわちメモリアドレスに置き換える。また再配置可能なロードモジュールならば、ベースレジスタをロードモジュールの先頭アドレスの値に設定する。その他必要な処理を行った後、プログラムの最初の命令語のアドレスをプログラムカウンタに設定して、プログラムの実行が開始される。こうしてメモリ上に読み込まれ実行が開始されたプログラムを、プロセスと呼ぶ。

前節のリンク作業は静的リンクと呼ばれ、リンク作業によってロードモジュールファイルをあらかじめ作成しておくというものであった。リンク作業には、もうひとつ動的リンクと呼ばれる方法がある。静的リンクがあらかじめロードモジュールを作成しておくのに対し、動的リンクはプログラムを複数のモジュール群で構成する。そしてプログラムの実行時に、必要なモジュールをメモリに読み込み、そしてリンク作業を行うのである。このため、ローダがリンカの役割も担うことになる。静的リンクとはリンクの仕組みが異なるために、動的リンクではオブジェクトモジュールのかわりにロードセグメントと呼ばれる形式を使用する。つまり、単一のプログラムはロードセグメントの集合体として表現される。

動的リンク作業には、実行直前にメモリ上で全てのロードセグメントを読み込んで結合する方式と、外部参照が最初に発生したときにその外部参照の参照先の機能やデータが格納されているロードセグメントを読み込み、外部参照を解決するという方式がある（2度目以降の参照が起こった場合は既に読み込み済みであるので新たに読み込む必要はない）。後者の方式を遅延リンクと呼ぶ。遅延リンクの様子を図5に示す。この例では、ロードセグメントAとBがメモリに読み込まれており、まだ読み込まれていないロードセグメントCの中にあるデータdへの参照が行われようとすると、ロードセグメントCをメモリに読み込み（図5－(1)）、外部参照を解決する（図5－(2)）。非遅延リンク方式の場合、プログラム起動時にリンク作業が行われるので、プログラムの起動が遅い。また、プログラムの全ての部分をメモリに読み込むので、消費するメモリ空間が多くなる。一方、遅延リンク方式の場合は参照が起こるまでリンク作業が行われないのでプログラムの起動が速く、プログラムの占有するメモリ領域が平均的に少なくてすむという利点がある。しかし、処理の内容は後者の方が複雑になるので実現するのはより難しくなる。

動的リンクを用いると、外部機器の接続時にその外部機器に対応するドライバを動的にメモリ上にロードするというサービスが可能になる。コンピュータの動作中に様々な外部機器を気軽に接続できることは、パーソナルコンピュータにとっては大事な特質であり、動的リンクの仕組みはパーソナルコンピュータ用のOSにこのような要求に応えるための礎を与えている。なお、静的リンク機構と動的リンク機構はかならずしも排他的ではない。通常、OSは場合によって使い分けが出来るように両方の機構を用意している。

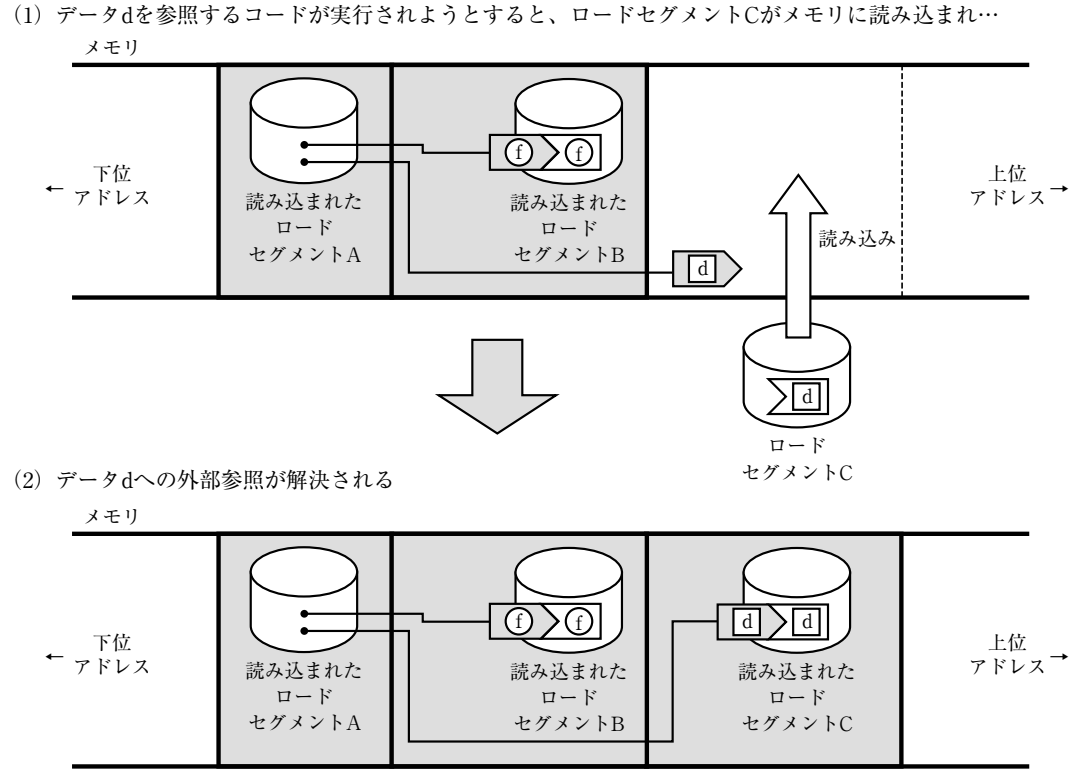


図5　動的リンク（遅延リンク）の仕組み

○3.1.3 共有ライブラリ

前述したように、静的リンクではオブジェクトモジュールを全てリンクしてロードモジュールを作成してしまう。そのため複数のプログラムが同一のオブジェクトモジュールを使用する場合、各ロードモジュールにそのオブジェクトモジュールが含まれることになるので、ディスク空間やメモリ空間の消費が大きくなる。一方、動的リンクを用いる場合はひとつのロードセグメントを複数のプログラムで共有することが可能なので、プログラムのサイズが全般的に減少し、ディスク空間やメモリ空間の消費が少なくなる。このように多くのプログラムから使用されるロードセグメントを共有ライブラリと呼ぶ。

共有ライブラリを更新することで、プログラム全体をコンパイルすること無しに複数のプログラムの機能を改善することが可能となる。このことは、共有ライブラリを用いると、特に大きなプログラムの保守が容易になることを意味する。したがって一般的には、OS自体も共有ライブラリの集合体として作成すると保守が容易になると言える。ところで、共有ライブラリを更新する際には、バージョンの違いが既存のプログラムの動作に悪影響を与えることがある。そのため、共有ライブラリのバージョン管理が重要になる。UNIXの例では、バージョンは基本的に

1.2.3

というように3つの番号をドットで区切った形で表現される。各番号はバージョンナンバで、新しい物ほど数値が大きくなる。3つの番号のうち、最初のものはメジャーバージョンナンバで、この

番号が異なるとライブラリの互換性は全く無いものとされる。2番目の数字はマイナーバージョンナンバである。同じメジャーバージョンでも、マイナーバージョンがプログラムによって要求されるものよりも小さい場合は、一般には互換性は失われる。3番目の番号はパッチナンバとして扱われ、パッチナンバのみが異なる場合には、互換性は保たれる。UNIXの共有ライブラリの名前は、ライブラリ本来の末尾にバージョンナンバをつけたものになる。例えば、ライブラリの本来の名前がlibでバージョンナンバが1.2.3なら、共有ライブラリ名は

lib.1.2.3

となる。

しかし、このようにバージョン管理の規則を決めていても、共有ライブラリの数が非常に多い場合には、互換性のテストが十分行えず、結果としてバージョン管理がずさんになり、逆に保守性を低下させている場合もあるようである。この問題点については後述する。

○3.1.4 スタックフレーム

実際のプログラムは、ルーチンを連鎖的に呼び出すことで動作する。ルーチンを呼び出す際には、メモリ上のスタックフレームと呼ばれる枠組みを用いて、連鎖的な呼び出しを実現する。図6にスタックフレームの概念図を示す。スタックフレームは定められたスタック領域にルーチンを呼び出すごとに、ルーチンに渡す引数、戻り番地、1つ手前のスタックフレームのアドレスが積まれる。ルーチンはスタック上に局所変数用の領域を確保して、処理を行う。その処理の中で別のルーチンを呼ぶ場合は、さらに同様にスタックフレームを積んで、そのルーチンを呼び出す。通常、スタックを制御するためにスタックの先頭を指すスタックポインタレジスタ、先頭のフレームの境界を示すフレームポインタレジスタを用いる。フレームポインタレジスタの指すアドレスには、1段階前のフレームへのポインタが格納されており、これによって各スタックフレームはリストとして管理されることになる。スタックは上位アドレスから下位アドレスに向かって、成長する。このスタックフレームは、セキュリティーホールの温床となっているバッファオーバフロー問題の舞台である。

◎3.2 メモリ管理

メモリという資源に関するOSの役割は、実際にコンピュータに搭載されているメモリ（実メモリ）上でユーザから要求された処理を遂行できるように工夫することである。メモリ管理の中でも重要な機能は、メモリ不足の解消とメモリ空間の保護である。

現在のパーソナルコンピュータ用OSでは、仮想メモリという機構で、メモリ不足を解消している。仮想メモリでは、実メモリと高速な補助記憶装置（ハードディスクなど。以下ディスク）を協調して管理し、実メモリよりも大きなメモリ空間を仮想的に構築する。これが仮想メモリ空間である。この仮想メモリ空間上のメモリアドレスを仮想アドレスと呼ぶ。仮想メモリ空間は、多数のブロックに分割され、それぞれが実メモリ空間上の同じサイズのブロックもしくはディスク上の同じサイズのブロックに対応づけられている。図7にその様子を示す。この対応付けはマッピングテーブルによって行われている。マッピングテーブルにはブロックの数だけ要素が並んでおり、仮想メモリ空間上のブロックとマッピングテーブルの各要素は順序よく対応している。マッピングテーブルの各要素には、各ブロックの内容が実メモリ空間上にあるか、それともディスク上にあるかを示すフラグと、実アドレスまたはディスク上の位置（ディスクアドレス）が格納されている。CPUが実メモリ上に対応づけられている仮想メモリ上のブロックにアクセスしようとする場合はそ

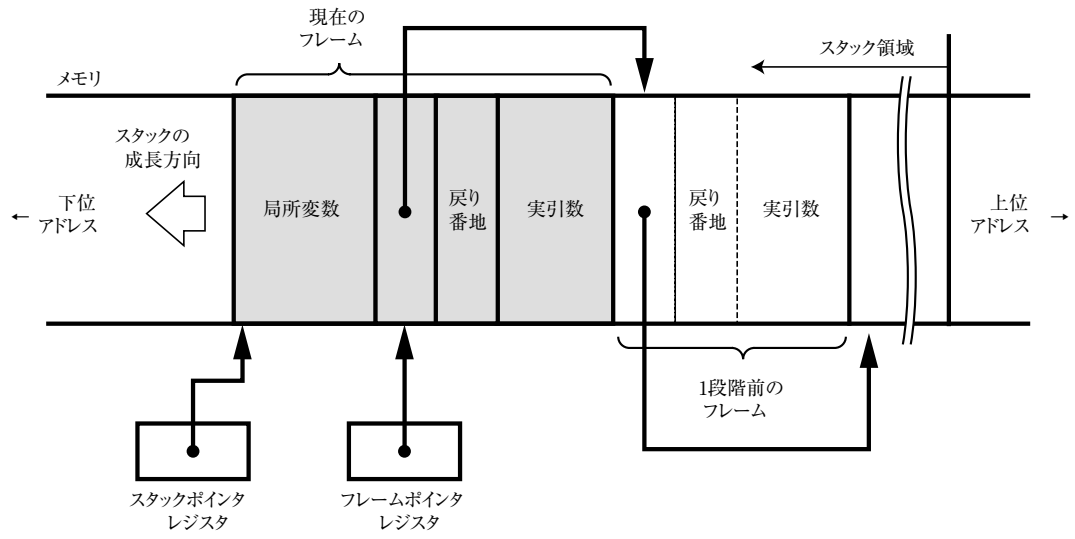


図6　スタックフレーム

のままアクセスが実行される。しかし、もしディスク空間上に対応付けされている仮想メモリ上のブロックにアクセスしようとした場合、例外が発生し、ディスク上に退避されているブロックが実メモリ空間上に読み込まれ、仮想空間上の当該ブロックは新たに読み込まれた実メモリ空間上のブロックへ対応付け直される。そして、プログラムの実行が再開される。また、実メモリ上のブロックが不足した場合には、実メモリ上の適当なブロックをディスク上へ書き出し、マッピングテーブルを書き換えて対応する仮想空間上のブロックを、書き出したディスク上のブロックへ対応づける。

図 7 の例では、仮想メモリ空間上の第 4 ブロック内へのアクセスが発生した場合、第 4 ブロックは実アドレス f にある実メモリ上のブロックへ対応づけられているので処理はそのまま継続される。しかし、仮想メモリ空間上の第 0 ブロックへのアクセスが発生すると、第 0 ブロックはディスクアドレスＢのディスク空間上のブロックへ対応づけられているので、このブロックの実メモリ上への読み込みが行われる。

仮想アドレスから実アドレスへの変換は、図 8 のように行う。図 8 の例では 1 ブロックを 4 キロバイト（4096バイト）とし、32ビット仮想アドレス（実効アドレス）のうち、20ビットがブロック番号を、残り12ビットをブロックの先頭からの位置を表している。2^{12}=4096なので、12ビットでちょうどブロック内のアドレスを表現できる。仮想アドレスから実アドレスへの変換は、ソフトウェアで行うと時間がかかりすぎるのでメモリ管理ユニット（MMU、Memory Management Unit）と呼ばれるハードウェアで行われるのが普通である。現在の代表的なパーソナルコンピュータ用のCPUには、メモリ管理ユニットが内蔵されている。マッピングテーブルは高速に参照する必要があるので、メモリ管理ユニット内の高速なキャッシュメモリに保持される。マッピングテーブルが 1 つだけだとマッピングテーブル自体のサイズが大きくなり、キャッシュの中に格納しきれなくなる。そのため、マッピングテーブルを多段にすることでマッピングテーブルのサイズを縮小する方法がよく用いられる。

仮想メモリ機構には、これまでの説明でブロックと呼んだものをどのように扱うかによって、 2

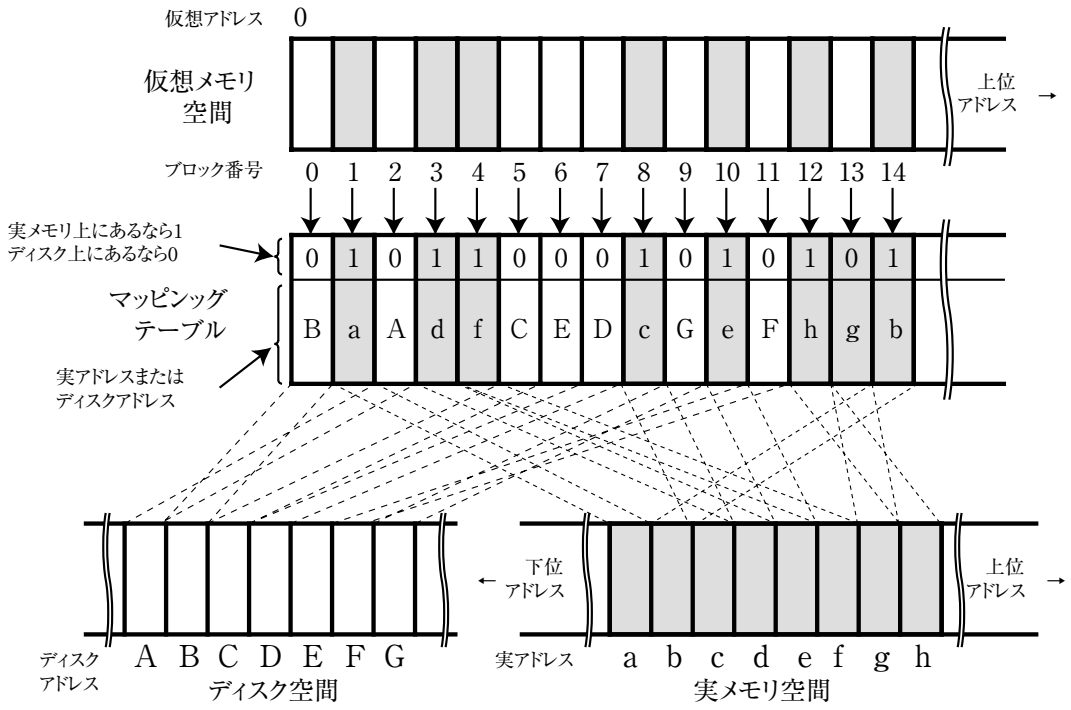


図7　仮想メモリの仕組み

つの方式がある。第1の方式はページングと呼ばれ、固定長のブロックを用いる（このブロックをページと呼ぶ）。実メモリブロックの内容をディスクに退避することをページアウト、逆の動作をページインと言う。実際に必要となったときにページングを実行する方式をオンデマンドページングと言う。第2の方式はセグメンテーションと言い、プログラム内の意味のあるひとかたまりをブロックとするものである（このブロックは可変長でセグメントと言う）。セグメンテーションはセグメントがプログラム的に意味のある単位（たとえば後述のプロセス単位）となる。プロセス単位で実メモリからディスク空間へ退避を行うことをスワップアウト、逆の動作をスワップインと言う。このようなことから、自然とセグメントのサイズが大きくなる（最小で数十キロバイト程度）。

セグメンテーションでは、プログラムの都合にあわせてセグメントの読み込みと書き出しが行われるため、処理内容は比較的簡単である。一方、ページングではページは固定長で小さい（数キロバイト程度）ので、マッピングテーブルは大きくなり、ページの読み込みと書き出しはプログラムの都合とは関係なく頻繁に行われる。そのため、処理は複雑かつ高速性を要求され、メモリ管理ユニットが必要になる。しかし、セグメンテーションでは実メモリのサイズより大きいプログラムを動作させることはできないが、ページングではそれが可能になる。そのため、現在のパーソナルコンピュータ用OSでは主にページングが用いられる。リンカとローダはページの境界にうまくコードの固まりを配置するようにして、ページインとページアウトが頻発しないように工夫する必要がある。なお、共有ライブラリのコードとデータは、複数のプロセスで共有されるページに配置する。

ところで、一般にプログラムは実行中にメモリを必要に応じて動的に確保し、割り当てる。この動的に確保するためのメモリ領域をヒープ領域という。通常、ヒープ領域は仮想メモリ空間の下位

※マッピングテーブルが1段の例

図8　仮想アドレスから実アドレスへの変換

アドレス部分にあり、上位アドレス方向に向かってメモリを割り当てていく。一方、スタック領域は最上位アドレス付近から下位アドレスに成長するようになっており（図9)、メモリ空間を効率的に使用できるようになっている。ヒープの使用領域がスタックと衝突した場合には、プログラムは正常に動作できなくなる。

◎3.3 多重プログラミング

もともと中・大型のコンピュータのOSは、高価な計算機資源を有効に使うために、並行してプログラムを動作させる多重プログラミングや、同時に複数のユーザが利用できるマルチユーザ環境を提供していた。しかし、その機能を実現するための負荷が大きかったために性能の低かったパーソナルコンピュータ用のOSはそのような機能を持っていなかったし、また必要ともされなかった。しかし今日では、パーソナルコンピュータの低価格化・高機能化により、パーソナルコンピュータ用のOSにも、多重プログラミング機能やマルチユーザ環境が必要とされている。ここではまず、多重プログラミングのメカニズムについて説明する。

○3.3.1 プロセス

プロセス（タスク）とは、プログラムがメモリ上に読み込まれて、実行可能になった処理単位である。OSはプロセスごとに様々な情報を管理し、操作し、計算機資源を配分する。

多重プログラミング環境において、複数のプロセスが並行動作している様子を図10に示す。CPUは、数十ミリ秒程度の間隔でプロセスを切り替えながら解釈・実行する。この方式をタイムシェアリングシステム（TSS、Time - Sharing System）と呼ぶ。図10の例では、期間T_0の間はプロセス１が実行されていて、他のプロセスは実行が中断されている。また期間T_1の間はプロセス３が実行されていて、他のプロセスの実行は中断されている。切り替えの期間が非常に短いので、外見上は複数のプロセスが同時に実行されているように見える。プロセスの切り替えはOSのディスパッチャによって行われる。ディスパッチャは実行中のプロセスの実行環境（プログラムカウンタの値、レジスタの値など）をPCBと呼ばれるプロセス管理ブロックに退避してプロセスからCPUを剥奪し、次に実行すべきプロセスの実行環境をPCBより再構築してそのプロセスに実行制御を渡す。このように並行に実行されるプロセスが共通のリソースに同時にアクセスし、なんらかの処理を加えようとすると、処理が互いに干渉して正常に行われないという現象が起こる。このような現象を防ぐために、排他処理が必要不可欠である。

プロセスは、図11に示した状態を遷移する。プロセスはまず生成されるとREADY状態になる。OSはREADY状態のプロセスの中から適切なものを選んで、CPUを割り当ててプロセスを実行する。

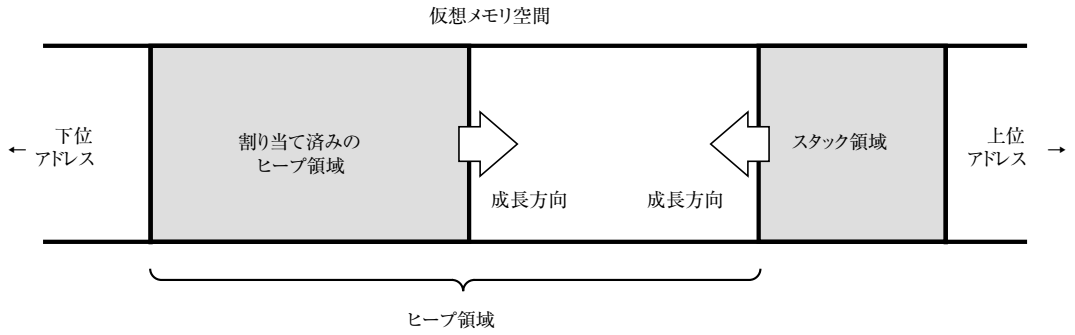


図9　ヒープ領域とスタック領域の位置関係

このときプロセスの状態はRUN状態になる。RUN状態のプロセスが処理を完了するとプロセスはそのまま終了する。もしRUN状態のプロセスが入出力処理を行うと、入出力機器からの終了報告（イベント）を待つために、自らWAIT状態になる。入出力機器からの終了報告をOSが関知すると、OSはイベント待ちしていたプロセスの状態をREADY状態に設定しなおす。WAITまたはREADY状態のプロセスは、他のプロセスにより強制停止され、SUSPEND状態になることがある。強制停止の解除も他のプロセスによって実行され、プロセスの状態は元に戻る。

OSがRUN状態のプロセスからCPUを剥奪する場合もあり、この場合はプロセスの状態はREADY状態に逆戻りする。READY状態とRUN状態を相互に遷移させるのはOSの働きである。READY状態にある複数のプロセスからひとつを選んでCPUを割り当てRUN状態にするOSの働きを、プロセススケジューリングと呼ぶ。また、このプロセススケジューリングを行う部分をスケジューラと言う。プロセスには早急に実行しなければならないものがある一方で、それほど緊急性を要しないものもある。そこで、通常はプロセスごとに優先度が割り振られている。一般にスケジューラは、優先度が一番高いプロセスを選んでCPUを割り当てる。同じ優先順位のREADY状態プロセスが複数有った場合は、一番早くREADY状態になったプロセスを選んでCPUを割り当てる。

OSがRUN状態のプロセスからCPUを剥奪してREADY状態に戻すことがあるのは、いつまでも特定のプロセスがCPUを独占しないようにするためである。その方法としては、タイムスライスとプリエンプションがある。タイムスライス機能は、プロセスが連続してCPUを占有することができる時間を決めておき、その時間の経過後にプロセスをREADY状態に戻す。互いに同等の優先度を持つプロセスが多い場合に有効である。プリエンプションは、RUN状態のプロセスよりも優先度の高いプロセスがREADY状態になったときに、RUN状態のプロセスをREADY状態に、より高い優先度のREADY状態プロセスをRUN状態にする。

○3.3.2 プロセスと仮想メモリ空間

プロセスへの仮想メモリ空間の割り当て方には 2 種類有る。ひとつの仮想メモリ空間を複数のプロセスで共有する単一仮想メモリ空間と、プロセスごとに独立した仮想メモリ空間を割り当てる多重仮想メモリ空間である。通常、多重仮想メモリ空間の方が処理が簡潔になる。例えば、各プロセスを特定の固定アドレスにロードすることができるので、リンクとロードの処理が単純になる。そのため、現在のパーソナルコンピュータ用OSでは、多重仮想メモリ空間を用いるのが主流である。

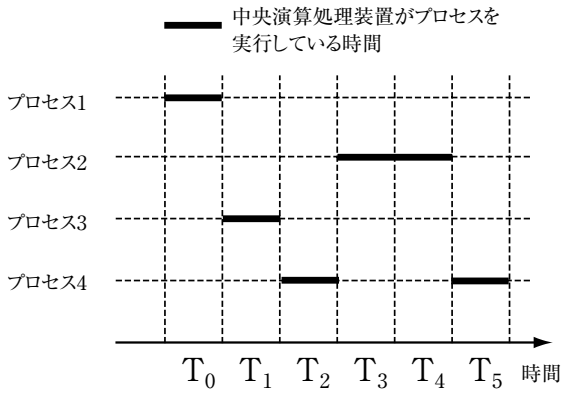


図10　複数のプロセスが並行動作する様子

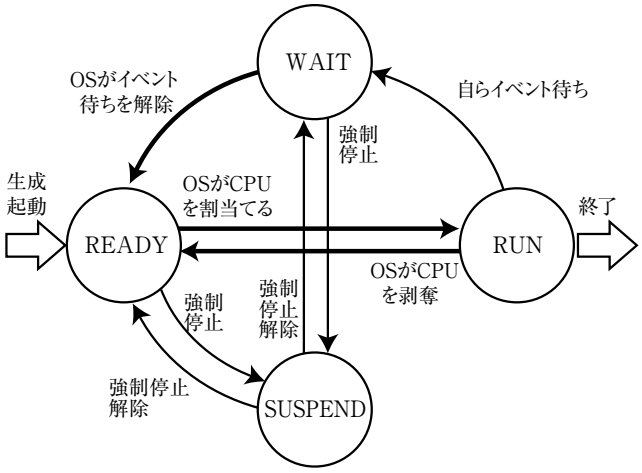


図11　プロセスの状態遷移

○3.3.3 スレッド

プロセスの内部で複数の小さな処理単位を並列で動作させることもできる。この処理単位をスレッドと呼ぶ。スレッドの利点としては、プロセスの切り替えには大きなコストがかかるが、スレッドは同一プロセス内に存在するので、スレッドの切り替えは非常に高速に実行できるということがあげられる。また、プロセスの中でユーザインタフェースの応答処理とデータ処理を別のスレッドで実行させることで、ユーザに待ち時間を感じさせないスムースな処理を実現できる。これは、パーソナルコンピュータのようにユーザがシステムに即応性を要求する環境では、必須の機能といえる。

◎3.4 マルチユーザ管理

パーソナルコンピュータの高機能化にともない、パーソナルコンピュータを多人数環境のオフィスや学校で共用使用するケースが増えてきている。その際に重要なのがマルチユーザサポートであ

る。ユーザごとにユーザ識別名とパスワードを設定し、コンピュータの使用時にはログインを行わせる。システム全体を管理するために強い権限を持ったスーパーユーザがおり、システムの設定はユーザごとに行えるようにする。ファイルやディレクトリには、ユーザごとに所有者を設定し、所有者とその他のユーザに対して、読み込み、書き込み、実行（ディレクトリの場合はディレクトリへの移動）の可否を設定できるようにする。またプロセスにも所有者という概念を持たせ、プロセス起動者の権限でプロセスを実行するようにする。こうすることで、プロセスはシステム上の資源に対して権限の無い処理は実行できなくなる。

◎3.5 GUI

パーソナルコンピュータ用のOSで特徴的なのは、洗練されたグラフィカルユーザインタフェイス（GUI）の発達である。Microsoft社のWindows OSやApple社のMacOSでは、ビットマップディスプレイ上に描画された直感的に理解しやすいウィンドウやメニュー、スクロールバー、ボタン、ラジオボタン、チェックボックス、アイコンなどの操作部品が表示され、マウスなどのポインティングデバイスでそれらを操作することが可能になっている。特にウインドウによって、プロセスを可視化することに成功している点は重要である。そのおかげで、各プロセスに対応するウインドウを同時に表示しながら、それらのプロセスの機能を組み合わせた作業が可能になっている。また、GUIのおかげでユーザは直感的に操作できるので、容易にプログラムの操作に習熟できる。操作が難しいところでは、グラフィカルなヘルプ機能によって、ユーザをガイドすることも可能である。

また、GUIの目標のひとつとしてWYSIWYG（「ウィジウィグ」と読む）があげられる。WYSIWYGは、"What You See Is What You Get"の略で、ディスプレイ上に表示されたものがそのままの形で印刷されるということである。書類作成や電子出版（DTP）などに用いられるパーソナルコンピュータにとっては、WYSIWYGは重要なファクターである。WYSIWYSの観点から最近注目されているのは、スキャナ、デジタルカメラ、ディスプレイ、プリンタなどの画像入出力機器間における色再現性である。たとえば、MacOSではColorSyncと呼ばれる機能で色再現をサポートしている。

◎3.6 多言語・国際化サービス

パーソナルコンピュータやインターネットの普及により、パーソナルコンピュータ用OSの市場は世界中に広がっている。そこで問題になってくるのが、文字をはじめとする文化の違いにいかに対応するかという点である。

基本的に、コンピュータで扱う文字は、表現したい文字集合を規定し、その中の特定の文字を特定の符号なし整数値によって表す。この符号なし整数と文字との１対１写像を、文字コードと呼ぶ。代表的な文字コードには7bitで欧米の文字集合（制御文字、記号、数字を含む）を表すASCIIがある。漢字だけで数万個の文字がある日本語を表すために、主にキャラクタ通信に用いられるJIS、主にパーソナルコンピュータ用の文字コードとして用いられるシフトJIS、主にUNIX系のOSに用いられる日本語EUCなどがある。いずれもASCII文字コードと互換性をもちながら、日本語の１文字を２バイトで表現するように拡張したもので、相互の互換性はない。また、中国語や韓国語をはじめとする各国でも自国の地域文化で使用する文字を独自にコード化したものが用いられてきた。このように相互に互換性のない文字コードが各国で使用されていると、作成された文書自体の互換性が失われるだけでなく、ソフトウェアの一元化された開発が困難である。そこで、全世界で使用

されている文字を含めた文字集合を規定し、それに含まれる文字を統一された文字コードで表現することを意図したUnicodeが定められている［1］［2］［3］。しかし、日本市場を対象としたパーソナルコンピュータ用のOSでは、未だシフトJISが主に用いられているのが現状である。

当然ながら、国や地域によって違うのは文字だけではなく、その他にも数値や日付の表現や通貨などがあげられる。現在では、これらの点や、文字コードをそれぞれの文化に合わせてローカライズされたOSが販売されている。しかし、各国・各地域におけるローカライズは別々に行われてきたので、同じOS・アプリケーションソフトでも各国・各地域用のバージョン間ではソフトウェア的に互換性が無いことがほとんどであった。だが、インターネットでは多くの国・地域のクライアントが同一のサーバへアクセスすることがありえる。また、それぞれの国や地域のためのローカライズ作業は開発者にとっては大きな負担である。そこでUnicodeや、置き換え可能なロケール（各国・地域に文化的に依存する振る舞いを定義したもの）などを採用することで、最初から国際化されたソフトウェアを開発しようとする流れが近年本格化しつつある［2］。国際化アプリケーションソフトを実現するには、もちろんOS自体が国際化されていなければならない。OSメーカーも国際化を視野に入れ、文字列の内部表現やディスク上のファイル名記録などにUnicodeを用いるといった動きを見せている。

4．現状の問題点と今後の課題

以上をふまえて最後に、パーソナルコンピュータ用OSの現状と問題点について述べる。

○4.1 セキュリティ問題

パーソナルコンピュータはもはや、インターネットに繋いだ状態で使用されるのが普通となりつつある。よって、セキュリティ性の高さがパーソナルコンピュータ用OSに求められる。しかし、現行のOS上でセキュリティホールが頻繁に見つかる事実は、OSの開発が機能偏重の傾向にあることを物語っているようである。

セキュリティホールとなる脆弱性にも、ユーザ管理機能の弱さに根ざすものや、外部のユーザに対して提供してはならないサービスを提供してしまっていた、というような単純な誤り、そして、ネットワークからの積極的な攻撃によってプログラムの制御権を奪われてしまうというものまで様々である。

ユーザ管理機能が弱いパーソナルコンピュータ用OSとしては、Apple社のMacintoshシリーズ用の旧OSであるMacOS 9があげられる。MacOS 9は基本的にシングルユーザOSであり、またプログラムはすべてスーパバイザモードで動作するため、機構上はセキュリティ上の問題が大きかった。MacOSの最新バージョンであるMacOS Xでは、UNIXをベースにしたシステムに変更されたため、この点については大きく改善された。一方、Windows NT系OSでは、ユーザ管理機構はしっかりしていたが、インストール方法によってはユーザが管理者として日常の操作をすることが可能な状態になるなどの問題があった。パーソナルコンピュータは本来、個人による使用を前提にしたものである。個人で使用するには、システム上のリソースを全て無制限に扱える方が使いやすい。しかし、コンピュータの高機能化とネットワークの普及によって、常に多人数による使用環境を考慮しなければならなくなっている。したがって、ユーザ管理方法を工夫して、個人用途でも使いやすい環境を構築することがパーソナルコンピュータ用OSに求められていると言える。MacOS Xは前述

のようにUNIXベースのOSであるが、通常は一般ユーザとしてログインし、管理者として必要な操作をするときにだけ管理者パスワードの入力を要求するようにして、使い勝手を向上させている。

システムへの不法侵入の方法として、ネットワークなどを通じた攻撃によってプログラムの実行権を奪うというものがある。実行権を奪われたプログラムが管理者権限で動作している場合、侵入者はシステム上のあらゆる実行権を握ることになる。このような攻撃の対象となるセキュリティーホールは主にプログラムのバグで、代表的なものはバッファオーバーフローと呼ばれるものがあげられる。通信などでデータを受信するとき、送られてくるデータのサイズをチェックせず、かつ受信データのバッファとして局所変数を使用すると、この問題が起きる。プログラムは各ルーチンを呼び出す直前に、スタックフレームに戻り番地を記録しておくが、局所変数もスタックフレームに置かれる（図６）。事前に予想されるサイズよりも大きいデータが送信されてくると、データサイズがチェックされていない場合は、データはスタックフレーム上のバッファを通り越して、戻り番地を上書きする。ルーチンの処理が終了すると制御は戻り番地にある命令に戻ろうとするが、戻り番地が正常でないので、通常はそのプログラムは例外を起こして停止する。しかし、データ内容に一連のコードとその先頭アドレスを戻り番地の位置に配置すれば、そのプログラムの制御を奪うことができる。バッファオーバーフローを防ぐ第１の方法は、プログラミングの段階で注意することであるが、プログラマの質に依存するため完全ではない。したがって、システムの方で統一的な予防策をとれることが望ましい。考えられる方法の一つは、戻り番地と局所変数を別のスタックフレームに置くというものだろう。しかし、単一の仮想メモリ空間に複数のスタックフレームが存在する場合、ヒープ領域とのかねあいも含めて実行環境を複雑にしてしまう可能性がある。

○4.2 共有ライブラリの互換性問題

前述したように、動的リンクを使った共有ライブラリは保守性の向上などをもたらしてはいるが、OSによっては、共有ライブラリのバージョンが適切に管理されておらず、古いソフトウェアをインストールする際に、新しい共有ライブラリを古い共有ライブラリで上書きしてしまう、ということが起こったり、新しい共有ライブラリに更新すると古い共有ライブラリに依存しているプログラムが動作しなくなるなどの問題が起きている［４］。

このような問題の解決方法として、以下のような方策が考えられる。まず、共有ライブラリを使用するモジュールの方では、ライブラリバージョン番号付きのシンボルを参照するようにする。共有ライブラリの方は、オブジェクト指向を取り入れた形でバージョンアップを行っていく。f()というルーチンを含むライブラリALibを擬似的にC++で記述した例を図12に示す。まず、抽象クラスALibの中で、f()の骨組みを仮想関数として定義する。図12では、f()はまず前処理のための仮想関数init()を呼び出して、次に処理の本体である仮想関数body()を呼び出し、最後に後始末を行う仮想関数fin()を呼び出す。そして、ALibの実質的な実装はAlibの派生クラスで行う。図12の例では、最初のバージョン01.00.00が、Alibの直接基底クラスALib_01_00_00として定義されていて、その中でbody()がオーバライドされている（このように、基底クラスで処理の骨組みを定義し、細部を派生クラスでオーバライドする手法を、デザインパターンの用語ではTemplate Methodパターンと呼ぶ［５］）。

以降、バージョンアップは直前のバージョンに対応するクラスの派生クラスを定義することで行えばよい。図12の例では、パッチバージョンクラスALib_01_00_01がクラスALib_01_00_00の派生クラスとして定義されている。このパッチの例では、f()の実引数値の範囲チェックを追加するだ

```
// Template Methodパターンを使用してライブラリのAPIを
// 抽象クラスとして定義する
class ALib {
  protected :
    virtual void init( T & a ) = 0;
    virtual void body( T & a ) = 0;
    virtual void  fin( T & a ) = 0;
  public :
    virtual void f( T & a ) {
      init( a );
      body( a );
      fin( a );
    }
};

// 最初のバージョン 01.00.00 の実装
class ALib_01_00_00 : public ALib {
  protected :
    void init( T & a ) {} // 実装は空
    void body( T & a ) { /* 適切な処理を記述 */ }
    void  fin( T & a ) {} // 実装は空
};

// パッチバージョンアップ 01.00.01 の実装
class ALib_01_00_01 : public ALib_01_00_00 {
  protected :
    void init( T & a ) { /* 引数の範囲チェック */ }
};

// マイナーバージョンアップ 01.01.00 の実装
class ALib_01_01_00 : public ALib_01_00_01 {
  protected :
    void  body( T & a ) { /* 処理 */ ALib_01_00_01::body( a ); /*   処理 */ }
};

// メジャーバージョンアップ 02.00.00 の実装
class ALib_02_00_00 : public ALib_01_01_00 {
  protected :
    void  init( T & a ) {} // 実装は空
    void  body( T & a ) { /* 適切な処理を改めて記述 */ }
    void   fin( T & a ) {} // 実装は空
};

// メジャーバージョンアップ 03.00.00 の実装
class ALib_03_00_00 : public ALib_02_00_00 {
    static T d;
  protected :
    void  body2( T & a ) { d = a; }
  public :
    void f( T & a ) { // f()をオーバライドして新しい骨組みを作成
    init( a );
    body( a ); body2( a );
    fin( a );
  }
};
```

図12　擬似的なC++コードで表した拡張された共有ライブラリ

けなので、仮想関数init()をオーバライドして、本処理body()の実行前に引数値の範囲チェックが実行されるようにしている。

このような範囲チェックはf()の本来の動作意味には大きな影響は及ぼさないが、バッファオーバーフローの防止などには有効である。図12では、マイナーバージョンアップであるクラスALib_01_01_00を定義している。この場合、本処理body()の機能を拡張するためにオーバライドしているが、その中で前のバージョンのbody()を再利用している。メジャーバージョンアップの場合には、互換性が失われるほどの変更にあたるので、全ての仮想関数を書き直したり（図12のクラスALib_02_00_00)、f()をオーバライドして新しい骨組みを作成（図12のクラスALib_03_00_00）したりすることになると思われる。

このような各バージョンのクラス群から、共有ライブラリを作成する。特徴的なのは、全てのバージョンのコードとデータが単一の共有ライブラリに含まれることである。そのため、新しいバージョンの共有ライブラリをインストールする場合には、既存の共有ライブラリを置き換えるのではなく、既存の共有ライブラリに新しいバージョンの部分を追加する形を取る。このため、新しい共有ライブラリを古い共有ライブラリで書き換えてしまうという単純なインストーラーが行いがちな誤りは根本的に解消される。そして、実際にリンクを行う場合は、参照する側が指定するバージョンに対応したクラスのインスタンスが生成されてリンクされる。たとえば、図12の共有ライブラリALibに含まれるバージョン02.00.00のf()を使用するモジュールに対しては、対応するALib_02_00_00のインスタンスが生成されてリンクされる。もし、マイナーバージョンの上がったクラスALib_02_01_00が存在すれば、そちらのインスタンスが自動的に選ばれてリンクされる。また、バージョン01.00.00のf()を使用するモジュールには、パッチレベルの最も高いALib_01_00_01のインスタンスが自動的にリンクされる。この方式では、参照する側も使用するバージョンを表明しているので、バージョン管理は比較的やりやすくなると思われる。

逆に、この方式の問題点としては次のような点が考えられる。まず、全てのバージョンのコードとデータを保持しているので、ディスク容量とメモリ容量を余分に消費する。しかし、共有ライブラリを利用するプログラムが多ければ、全体的なスペース効率は向上する。また、図12のALib_01_01_00::body()が基底クラスのALib_01_00_01::body()を再利用しているように、新しいバージョンが古いバージョンの一部を利用することで、共有ライブラリのサイズを小さくすることができると思われる。その他、リンクの処理が複雑になることと、仮想関数を使用することによって、ルーチンの呼び出し時に若干のオーバヘッドが生じることが予想できるが、バージョンアップを頻繁に行うルーチンは、比較的高級な処理を行っていると考えられ、このオーバヘッドが大きな問題になることは少ないと予想できる。なお、C言語のような非オブジェクト指向の言語から図12に示したオブジェクト指向的な構造を持つ共有ライブラリをどのように自動的に生成するかという問題や、実際に実装しての検証は今後の課題である。

○4.3 国際化に関する問題

現在普及しているパーソナルコンピュータ用OSは、高度な国際化機能を搭載しており、同一のコンピュータで複数の言語を同時に使用することも可能になっている。しかし、蓄積された過去の資産を無視することはできず、アプリケーションソフト、ユーザのレベルでは本格的なUnicode環境への移行は始まっていない。現在では、多くの書類上のテキストデータはいまだにシフトJISで表現されている。しかし、Unicodeの使用がアプリケーション、ユーザレベルで本格化すると、ひ

とつのシステム内にUnicodeとシフトJISコードの文字表現が混在している状態になり、混乱が予想される。特に、プレーンなテキストファイルは、外部から文字コードの種類を判別する方法はなく、テキストデータの内部をいくらかスキャンして、自動的に判別することになる。

また、Unicode自体が扱いにくいという側面もある。例えば、Unicodeは中国、日本、韓国などで使用される共通の漢字をひとつのコードに中で表現しているので、これらの漢字文字を整列するには、単にコードを数値として整列してもだめで、変換テーブルを使用して順序を変換してから並べ直す必要がある。その他、Unicodeは“正規化”された状態だと、「が」を「か」と濁点の「゛」の２文字の組で表すようになっている。これは、Unicodeでは１文字を可変長で表すと言うことであり、文字列の比較などのテキスト処理を複雑にしてしまう。

これらの問題を解決するには、様々な方面から多くの対策を取らなければならないだろう。たとえば、アプリケーションの新しいバージョンではデータの文字コードをUnicode中心にして、Unicodeへの移行を促進する。また、開発者がUnicodeを無意識的に扱えるように、Unicode用の文字列比較関数などのテキストAPIを充実させなくてはならないだろう。また、開発者やユーザがシフトJISとUnicodeの違いを意識しなくてもよいように、場合によっては適宜、シフトJISとUnicodeの変換を自動で行うようにする。

○4.4 GUI の課題

GUIについては現状で特に大きな問題になっている部分はないが、まだ改善の余地が残っているように思われる。例えば、GUIの応答速度の改善があげられる。GUIの応答速度が遅くなると、ユーザの作業効率を低下させてしまう。アプリケーションによっては、GUIの応答速度が遅いものも見受けられる。応答速度を向上させるには、ユーザが操作中のアプリケーションのプロセス優先度をOSが動的に上げる、ということが考えられる。また、ユーザの操作をすぐに関知できるように、GUI操作に対する応答ルーチンをスレッド化したアプリケーションフレームワークを開発者に提供するのも効果があると思われる。

○4.5 処理データの増大にともなう問題

ここ数年、パーソナルコンピュータが高機能化してきたため、また、ディスクやメモリが安価になったこともあって、パーソナルコンピュータ上で処理されるデータのサイズは、急激に大きくなっている。デジタルビデオデータはその代表例である。そのため、OSは拡張され続け、使用可能なディスク容量やメモリ容量はこの数年で飛躍的に増大した。しかし、特に使用可能なメモリ空間は32ビットアーキテクチャの制限によって、限界に達しようとしている。例えば、32ビットアーキテクチャ上のMacOS Xでは、アドレス可能なメモリ空間は、その仮想メモリの管理方法から４ギガバイトに制限される。４ギガバイトでは、長時間のデジタルビデオデータを全て仮想メモリ空間上において編集することは不可能であり、アプリケーション側で特別なキャッシュ処理をしなくてはならない。しかし、このようなアプリケーション独自のキャッシュ処理は、OSの仮想メモリメカニズムと干渉することもあり、効率的なものを実現するのは難しい。この問題を解決する根本的な方法は、64ビットアーキテクチャへの移行である。

○4.6 OSの複雑化と肥大化の問題

多くのユーザの様々な要求に応えるため、また商品としての魅力を維持するために、現在のパー

ソナルコンピュータのOSは非常に巨大で複雑なものになっている。このようなOSでは、動作不良（バグ）の発見が継続して報告されている。特定のソフトウェアの組み合わせによって不都合が起きるケースも散見される。アプリケーションソフトウェアの作成も、APIの増加・複雑化でだんだんと難しくなってきている。ここまで列挙してきた問題点の多くは、結局の所、OSの複雑化・肥大化に起因していると言える。

ソフトウェアの構造の複雑性を低減する基本的な方法は、モジュールごとの独立性を高めることである。しかし、一般的にモジュールの独立性を高めると、オーバヘッドが生じる。一方でOSは高い実行効率を求められるので、モジュールの独立性を犠牲にして、各モジュールの実装に相互に依存したコーディングを行ってしまう傾向があると思われる。しかし、パーソナルコンピュータの高機能化が著しい今日では、モジュールの独立性を重視するメリットの方が高くなってきているのではないだろうか。また、現在のいくつかのパーソナルコンピュータ用OSの構造を複雑にしている原因のひとつとして、過去のバージョンとの互換性を保持していることがあげられる。ひとつの解決策として、大規模なバージョンアップの際、OS自体は互換性をある程度切り捨て、過去の特定のバージョンの環境を提供するエミュレータを用意するという方法がある。例えば、MacOS XはUNIXベースの全く新しいOSだが、MacOS 9の環境を提供するエミュレータを用意している。このようにすることで、ユーザの互換性に対する要求に応えながら、大規模なバージョンアップを成功させている。

現在のパーソナルコンピュータ用OSは多様な機能を要求され、肥大化している。だが、OSに用意されている機能が、全てのユーザに利用されているわけではない。そこで、モジュールの独立性を高めて、必要に応じてモジュールをネットワーク経由でダウンロードし、カーネルの拡張部分として動的に組み込む方法が考えられる。実際に、ドライバに関してはそのような動的な組み込みが実現されている。しかし、OSのモジュールのように比較的大きなサイズのモジュールをダウンロードするには、より高速なネットワークとサーバシステムが必要になると思われる。

5．まとめ

現在の主要なパーソナルコンピュータ用OSは複雑化、肥大化しすぎている。現状のOSの抱える様々な問題を解決するためには、過去の資産にこだわることなく現在のOSの構造を大幅に見直し、保守可能な程度に複雑性を抑えるようにする必要があると思われる。機能の増加よりも、バグやセキュリティホールの減少を優先させて開発していくことがユーザにとって重要になってきている。ユーザとしては、セキュリティホールなどに常に注意を払い、システムアップデートを逐次行い、余分な機能やソフトをインストールしないなどの防衛策をとる必要がある。そのためにもコンピュータの基本構成やOSに対する基本知識の修得と問題点の理解は、一般ユーザにとっても重要になってきていると言える。

参考文献

［1］The Unicode Consortium編,"The Unicode Standard,Version 3.0",Addison‐Wesley,2000
［2］清兼義弘&末廣洋一,"国際化プログラミング I18N ハンドブック",共立出版,1998
［3］Ken Lunde,"CJKV Information Processing",O'REILLY,1999
［4］John R.Levine,"Linkers &Loaders"（榊原一矢監訳、ポジティブエッジ訳）,オーム社,2001
［5］Erich Gamma,Richard Helm,Ralph Johnson &John Vilssides,"オブジェクト指向における再利用のためのデザインパターン改訂版"（本位田信一,吉田和樹訳）,ソフトバンクパブリッシング,1999
［6］菊池豊彦,"経営情報系の情報科学II‐情報処理編‐",コロナ社,1990
［7］Shlomo Weiss&James E.Smith,"PowerPC解説POWERからPowerPCへ",日本アイ・ビー・エム株式会社監訳,1995
［8］谷口秀夫,"オペレーティングシステム概説その概念と構造",サイエンス社,2000
［9］谷口秀夫,"オペレーティングシステム",昭晃堂,1995
［10］前川守,"岩波講座ソフトウェア科学6 オペレーティングシステム",岩波書店,1988
［11］久保秀士,"OS概論",共立出版,1988
［12］脇英世,"Windows 入門‐‐新しい知的ツール‐",岩波書店,1995
［13］Bil Lewis &Daniel J.Berg,"マルチスレッドプログラミング入門"（岩本信一訳）,アスキー出版局,1996
［14］Steve Kleiman,Devang Sash &Bart Smaalders,"実践マルチスレッドプログラミング"（岩本信一訳）,アスキー出版局,1998