13 コンピュータのソフトウェア

計算機の装置そのもの ··· ハードウェア
計算機の利用技術(特にプログラム) ··· ソフトウェア

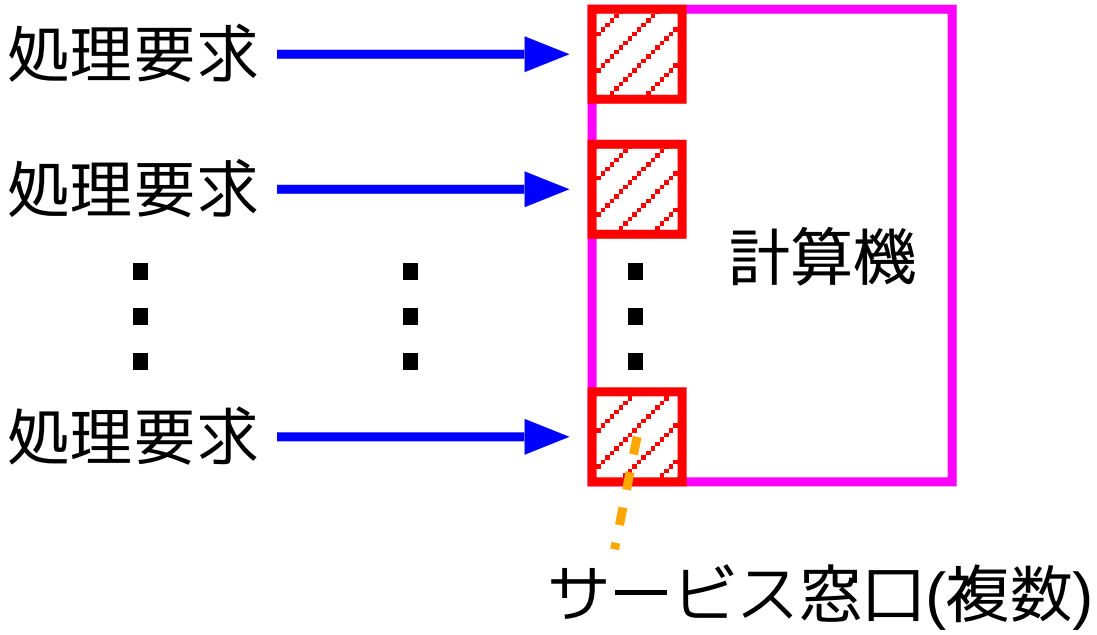
13–1 コンピュータの処理形態(利用形態)

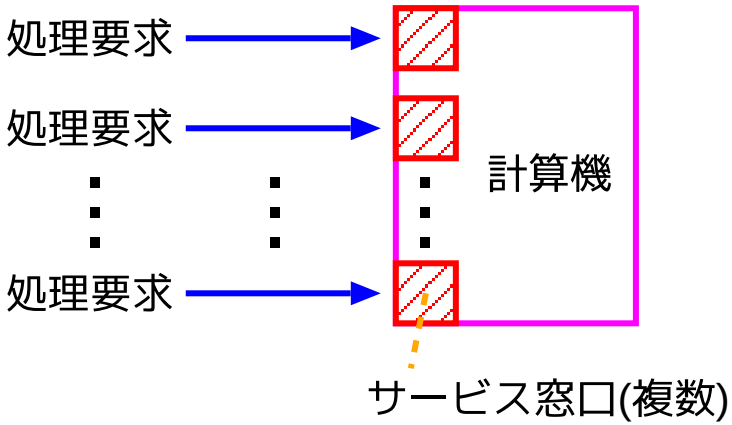
通常、我々は手元にあるコンピュータを占有して使ったり、(ネットワークを通して)共用のコンピュータを利用したりする。これら様々な利用形態に対して、サービスを提供するコンピュータ側にも色々な処理形態が用意されている。

⇨ 次に、代表的な処理形態を列挙する。

- 会話型処理:　我々は、コンピュータにコマンド(動作指示;マウスのクリックも含む)を与え、その応答を見て次のコマンドを与え、 ... という会話を通してコンピュータを利用することがほとんどである。

  この種の利用を可能にするために、コンピュータ側は、利用者との会話の窓口を開いて、そこからの指示を次々と実行する仕組みを用意している。　　特にUNIX系のコンピュータの場合には、会話の窓口を複数用意し、会話相手を切替えながら複数の利用者からの動作要求に並行して対処できる様になっている。





補足：

実際には、複数の相手との会話は次のように行われる。

◇ 計算機内に動作要求の待ち行列を保持し、基本的にはこの順番に処理を行う。

◇ 1つの要求に対する処理がある時間を越えると、この処理は途中で打ち切り待ち行列の最後に移す。

コンピュータが高価であった時代には、中央の1台の計算機を複数の利用者が同時に,そしてオンラインで会話的に利用する方式がよく用いられていた。 この処理方式を特にタイムシェアリング処理 (時分割処理,TSS処理) と呼ぶ。

- リアルタイム処理(実時間処理): 発生したデータ(または処理要求)に対して、計算機への入力,処理,出力が要求された時間(普通数秒から数十秒)内に行われる処理方式をいう。通常、計算機への入力はオンライン(i.e.処理要求発生の地点から通信回線を通して直接計算機へ入力する方式)で行われるので、オンラインリアルタイム処理(**online real-time** —)となる。

リアルタイム処理の対象分野としては、

例えば次のものがある。

① 自動機械,工程などの制御 ··· センサからの入力に応じて計算機が動作し、計算結果を基に計算機が直接全体を制御する。

② 交通の制御/管制 ··· センサや人間,他の計算機などからの入力に応じて計算機が動作し、計算結果と人間の意志を基に交通を制御/管制する。

③ 問い合わせ応答 (e.g. 座席予約,バンキング, 情報検索サービス) ··· 数多くの端末からランダムに入力される問い合わせに対して、計算機はその答を返す。一般に、中央に保持するデータベースへのアクセス結果によって問い合わせの答が決まる。

1970年代頃までは、

バッチ処理 (一括処理) と呼ばれる処理形態もあった。 これは、クローズドショップ式の利用形態 (i.e. 利用者が直接計算機を操作しない利用方式) の下で計算機の稼働率を上げるために考え出された処理方式であり、ある期間内に集められたジョブ (利用者から見た、計算機に処理させる仕事の単位) を幾つかずつまとめて処理していくというものである。 ここで、1回の処理のためにまとめられたジョブの束をバッチと呼んだ。また、計算機から離れた場所からオンライン (i.e. 通信回線を介して) でジョブの入力/結果の出力を行う場合を、特にリモートバッチ処理と呼んだ。

現在では、個々のコンピュータの処理能力が向上し、ほとんど全ての処理が手元のパソコンで賄える様になってきたし、また、手元のパソコンの手に負えないジョブを共用のスーパーコンピュータに投入する場合も、(過重負荷にならないなら) ジョブ投入後即座にジョブの実行が開始されるので、昔の、ジョブを溜めた後で複数を組み合わせて並行に実行するタイプのバッチ処理は今では全く行われていないだろう。

13–2 プログラミング言語

計算機に何らかの処理を行わせたい時には、その処理手順を計算機の理解できる形で明確に記述して計算機に教え込ま(i.e.入力し)なければならない。

一般に、処理手順の記述形式のうち計算機の理解できる様に人工的に設計されたものは、計算機と利用者の間のコミュニケーションの手段を与えるのでプログラミング言語と呼ばれる。そして、1つのプログラミング言語の下で処理手順を記述したものをプログラムという。

特に裸の計算機の理解できる言語は、その計算機のアーキテクチャ(ハードウェアの論理的構造)に直接関わるので機械語と呼ばれる。機械語のプログラムは計算機の基本動作を表すビット列をそのまま並べたものであるため抽象度が低く分かりにくい。

初期の計算機では プログラミング言語としては機械語しかなく、プログラム作りは非常に困難であった。

⇨ アセンブリ言語と呼ばれる言語が次に出現した。

- プログラムの基本動作はビット列ではなく「基本動作の論理的意味を反映した記号」を基に構成される。
  $\Longrightarrow$ プログラムの書き易さ,理解し易さはある程度改善された。
- アセンブリ言語のプログラムの各ステップは計算機の基本動作とほぼ1対1に対応している。
  $\Longrightarrow$ 我々人間が簡単と感じる処理でも多くのステップが必要。
- 機械語同様計算機のアーキテクチャを反映した言語。
  $\Longrightarrow$ 1つの計算機についてアセンブリ言語を修得したとしても、それで別の計算機のプログラムを作れるとは限らない。

補足：
アセンブリ言語のプログラムは一旦機械語に翻訳されなければならない。 アセンブラ

プログラミング言語としては

アセンブリ言語よりもっと人間に近いものが望ましい。できることなら日本語,英語などの自然言語をプログラミング言語として、どんな人でもすぐにプログラミングができる様になればよい。

⇨ **FORTRAN**(1957年完成,IBM社)
(数式や日常言語に近い形でのプログラミングを目指して
最初に設計された言語)

⇨ プログラムの作成時間が大幅に短縮

⇨ 計算機の使用者がどんどんと増えた。

⇨ 日常言語に近い形でのプログラミングを目指した言語
(高水準言語または高級言語,という) が次々と開発されていった。

- 個別の計算機アーキテクチャに依存しない。
- 人間にとって比較的分かり易い。
- 処理手順の記述力も強い。

$\Longrightarrow$ プログラムの生産性保守性の点で優れている。

補足：
高水準言語のプログラムもそのままでは実行できない。
アセンブリ言語の場合と同様一旦機械語に翻訳してから実行したり、プログラムを1ステップ毎に理解/解釈しながら実行させたり、しなければならない。
コンパイラ, インタープリタ

次に、高水準言語の例をいくつか列挙する。

- **FORTRAN(FORmula TRANslator,1957~):** 最も古く、昔は(汎用機の下で)COBOLに次いで世界中でよく使われていた。元々科学技術計算用にIBM社により開発されたものであるが、FORTRAN I(1958 IV(1962),FORTRAN66(1966 年に国際的に標準化),FORTRAN77, と改良が進み、それに伴って汎用性も高くなっていった。FORTRAN は記憶容量の面でも実行時間の面でもハードウェアの性能を最大限に引き出せる様に設計されており、数値計算や統計計算などに有用な組込み関数,ライブラリが豊富に用意されている。

- **ALGOL60(ALGOrithmic Language,1958~):**
- **ALGOL68(1968~):**
- **Pascal(1968~):** プログラミングの組織的教育,計算効率の良さ,処理系の作り易さを目指して **N.Wirth** によって設計された言語であり、研究・教育用として現在よく用いられている。全般的に **ALGOL60** の骨格を採用し、系統的プログラミングに適している **(i.e.** アルゴリズムを自然に分かり易く記述する能力を持つ**)**。
- **Ada(1980~):**
- **C(1972~):** **AT&T** ベル研究所のミニコンピュータ **PDP-11** 上に開発されていた **UNIX** オペレーティングシステム(アセンブリ言語で記述されていた)を高水準言語で書き換えるために **D.Ritchie** によって設計された言語であり、その系譜を言語C←言語B(1970~)←言語BCPL(19~ ;データ型のない言語で、これを用いて英国ケンブリッジ大学では **OS-6** というオペレーティングシステムを開発した)とさかのぼることができる。改良/標準化の際には **Pascal** の影響も受けた。データの取扱いに関して「低水準言語」の性格を持っているためシステム記述言語と

いう色彩が強いが、構造的プログラミングのための道具立ても揃っているため科学技術計算のための汎用言語として使うこともできる。

- **C++(1983~):** 代表的なオブジェクト指向言語の1つ。元々は、離散事象のシミュレーションを効率的に行うために、C言語とSimula67を土台にしてAT&Tベル研究所のB.Stroustrupによって1980年頃に設計が始まった言語で、最初は「クラス付きC (C with classes)」という名前で発表された。C++と命名された1983年以降もC言語との互換性を保ちながら拡張/改良が加えられたため、従来の手続き型プログラミングにもオブジェクト指向プログラミングにも対応できる様になっている。
- **Java(1995~):** 現在最も注目を集めているオブジェクト指向言語。元々は、情報家電製品にソフトウェアを組み込んでその配付やバージョンアップを円滑に行うために、C言語とC++を土台にしてSun Microsystems社のJ.Goslingによって1991 年に設計が始まった言語で、最初はOakと名付けられていた。(開発社の部屋の窓からOakの大木が見えたから。) しかし、Oakという名前の言語が既にあることが判明し、Javaという名前になった。(開発者達が喫茶室に集まった時、コーヒー

の銘柄にちなんで提案された。)　本来のSun社のねらった情報家電製品の市場は伸びなかったけれども、その後、1993年以降のWWWの爆発的な人気により、JavaはダイナミックなWebページ作成ツールとして急速に発展・普及している。

- **Modula-2**(MODUlar LAnguage,1979~):
- **SIMULA**(1964~):
- **Smalltalk**(1972~):
- **COBOL**(COmmon Business Oriented Language,1959~):
- **PL/I**(Programming Language/one,1965~):
- **BASIC**(Beginner's All-purpose Symbolic Instruction Code,1965~):
- **APL**(A Programming Language,1962~):
- **Lisp**(LISt Processor,1958~):　非数値情報の処理のためにMITのJ.McCarthyらによって開発された関数型プログラミング言語であり、リスト (list; データの並び) の処理が得意。その出現以来、定理の証明, 数式処理,パターン認識,計算機によるゲーム,ロボットのソフトウェ

アなど、人工知能研究用の言語として主に発展し、Lisp専用の計算機(Lispマシンという)もいくつか開発されていた。 方言も多い(e.g.Inter
が、標準化を求める声の高まりによって Common Lisp が案として設計された。

- Prolog(PROgramming in LOGic,1972~): 元々自然言語の構文解析のために A.Colmerauer らによって開発されたプログラミング言語であるが、R.A.Kowalski の論文(1974)によって記号論理との関連が明らかにされてからは論理プログラミング言語として知られる様になった。そして、D.H.D.Warren らによって実用的な処理系(DEC-10 Prologと呼ばれるものでインタープリタとコンパイラを含む,1977)が作られたこと，さらには日本の第五世代コンピュータプロジェクト(1982~)の核言語として採用されたことによって、Lispと並ぶ人工知能用言語として広く普及した。ICOT(Institute for new generation COmputer Technology,新世代コンピュータ技術開発機構;第五世代コンピュータプロジェクトのために作られた研究組織,1982~)によって開発されたESP,GHCを始めPrologの改良/拡張も盛んに行われた。標準化のための国際的な活動は、事実上の標準であったDEC-

10 Prologを基礎言語として1988年に開始され、1992年の時点で最終段階に入った。

- **OPS**(Official Production System,1977~):
- **LOGO**(ギリシャ語の $\lambda o\gamma o\varsigma$ に由来,1967~):
- **SNOBOL**(StriNg-Oriented symBOlic Language,1964~):
- **GPSS**(General Purpose System Simulator,1961~):
- **QBE**(Query By Example,1975~):
- **SQL**(Structured Query Language,1974~):

オブジェクト指向プログラミング:

- 大規模なプログラミングを効率的に行うためには、個々のソフトウェアモジュールの独立性を高め、それらを再利用可能なソフトウェア部品としなければならない。

  ⇨ モジュール化、「抽象データ型」といった考え方を更に1歩進めたものがオブジェクト指向の考え方。

- オブジェクト指向プログラミングにおいては、データとそのデータに関わる操作をカプセル化したオブジェクトと呼ばれる、自律した動作主を考え、それらのオブジェクトが互いに動作依頼のメッセージを送り合いながらプログラムの実行を進めてゆく。

  ⇨ プログラムの動作を、オブジェクト同士の相互作用のシミュレーションと見ることが出来る。

どの言語を使えば良いのか?:

使用目的に合った言語を選べば、プログラムも作り易い。

例えばアセンブリ言語だと、

- ハードウェア動作を理解するのが目的なら、これが良い。
- うまく書くと高速でコンパクトなプログラムが可能。
- 入出力機器の制御といった細かなプログラミングも可能。

しかし、

- コンピュータの機種に依存したプログラムしか出来ない。(i.e. 移植性がない。)
- 大規模なプログラミングには向かない。

⇨ 応用プログラム作成にどうしても必要という場合でも、アセンブリ言語は本当に必要な箇所だけに限定すべき。

高級言語に関しては、

- 科学計算なら、**C, C++, Fortran, ...** あたり。
- **C**言語は**UNIX/Linux**に必ず付いているので、色々なコンピュータ上で実行できる。
- **Java**は実行速度が遅いので、科学計算には向かない。
- 全ての可能性について試行錯誤を繰り返したければ、**Prolog**が便利。
- 人工知能関連の処理をしたければ、**Lisp, Prolog, ...** 。
- ほとんどの言語はシステムプログラムを書くのに向かないが、**C**言語のようにシステムプログラムを書くのに適した言語もある。

例えば、
UNIXはほとんどの部分が**C**言語で書かれている。
プリンタの動作を制御するプログラムも**C**言語で書かれている。

13–3 プログラムの実行過程

前節で見た様に、高水準言語, アセンブリ言語のプログラムはそのままの形では計算機で実行できず、実行のためにはインタープリタまたはコンパイラが必要である。

インタープリタによるプログラムの実行:
プログラムを1ステップずつ理解/解釈して、そのステップでどういう影響/変化がもたらされるかを記録しながら実行手順を進めてゆく。それを行うプログラムをインタープリタと言う。
元のプログラムを (コンパイラやプリプロセッサによって) 仮想機械語や構文解析木などの中間言語に変換して、それを基にインタープリタで実行させることもある。

補足：
Java, Smalltalk の場合は、通常、プログラムを一旦仮想機械語に翻訳しその結果をインタープリタで実行する。

インタープリタ方式の長所と短所は次の通りである。

長所 (1) 動的な言語機能 (e.g. 動的なデータ型, 変数や関数の動的宣言, 名前の動的有効範囲) に対処し易い,

(2) 処理系の作成が容易,

(3) プログラムの部分的実行が容易,

(4) デバッグ (i.e. プログラムの修正) が容易,

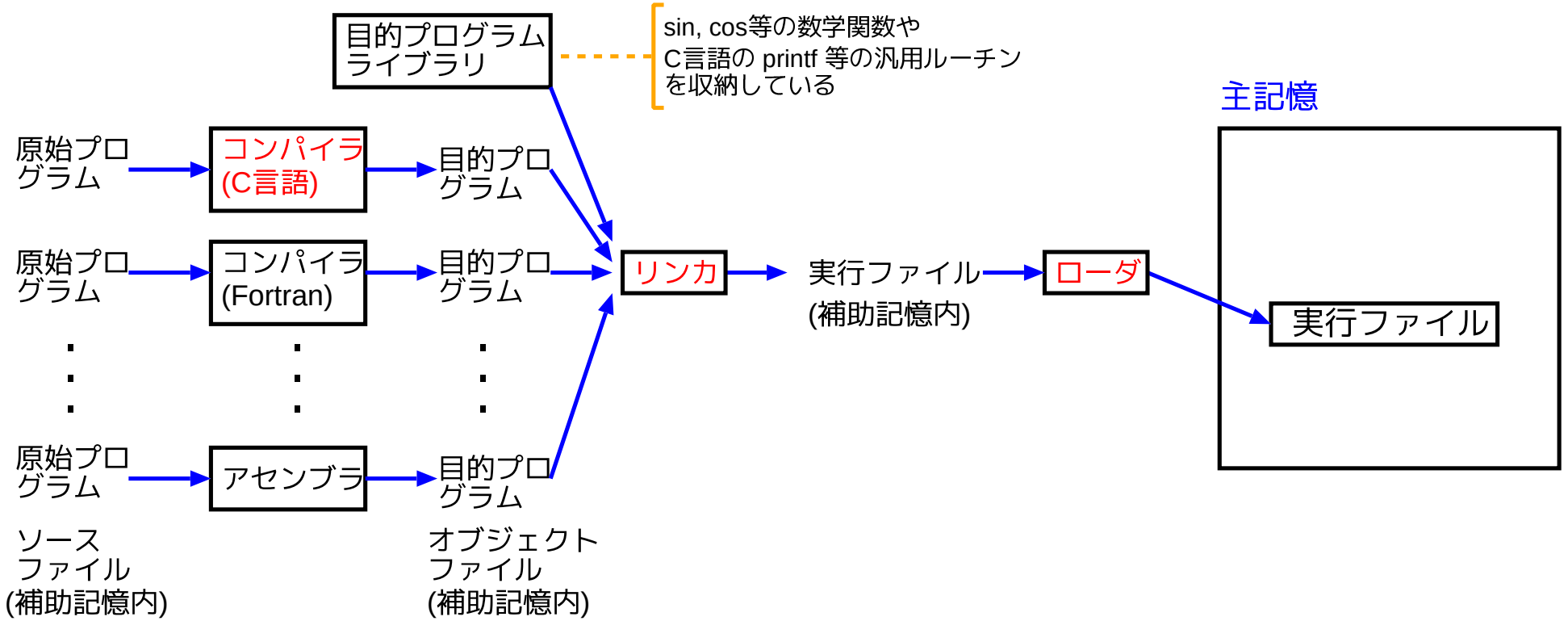
(5) 翻訳などの時間が不要になる。

短所 (1) 実行速度が遅い (翻訳されたプログラムの場合の数倍 ~ 数十倍の時間がかかる),

(2) コードの最適化に相当することができにくい。

コンパイラによるプログラムの実行:

高水準言語のプログラム (原始プログラム, と言う) は、通常、コンパイラと呼ばれるソフトウェアによって機械語または中間言語 (e.g. 仮想機械語, 構文解析木) に翻訳されてから実行される。一般には、高水準言語のプログラムは次の図の様なプロセスを経て実行される。



13–4 オペレーティングシステムとその目的

裸の (i.e. ハードウェアだけの) 計算機では

- 使い勝手が悪く、しかも
- 「利用希望者は予約を取りその時間は一人で計算機を独占する」という利用形態しか考えられない。

⇨ 計算機の有効利用は望めない。

⇨ 高水準言語

高水準言語のプログラムを実行させる場合には

- インタープリタ, またはコンパイラ,リンカ,ローダというソフトウェアが必要になる。
- 作成したプログラムを次回の使用のために磁気ディスク等に保存しておくためのソフトウェアも必要。
- 入出力装置を使う際にはそれらの装置に固有の制御を行うソフトウェアも必要。
- 通信回線を介して計算機を使う際には端末と計算機の間のデータのやり取りを行うソフトウェアも必要。

⇨ 高水準言語のプログラムとそれを実行する計算機ハードウェアの間には大きなギャップがあり、これを埋める必要がある。

そのために用意するソフトウェアの機能は
プログラミングの際に用いた言語に依らず、汎用性の高いものでなければならない。

⇨ 高水準言語のプログラムとそれを実行する計算機ハードウェアの間には大きなギャップがあり、これを埋める必要がある。

⇨ 利用者と計算機ハードウェアの間にオペレーティングシステム (**OS**,基本ソフト) と呼ばれるソフトウェアを設け、計算機システムの運用に関して次の様な目標の達成 (または性能の向上) を図らせる。

- (1) 計算機システムの有効利用
- (2) 快適な使用環境 (ユーザインターフェース) の実現

(1) 計算機システムの有効利用:

初期の頃は計算機が高価であったためこの目標を達成することが最も重要な課題であり、これがOS誕生の動機となった。

この目標の達成度の具体的目安としては

スループット (単位時間内に処理されるジョブの量, 件数)

が一般に使われる。

スループットの向上のためには、

OSはシステムの各資源 (e.g. CPU, 主記憶, 補助記憶, 入出力装置, システムソフトウェア) が最大限にバランス良く利用される様にしなければならない。これらの処理のためにOS自身がCPUを使うこともあり、このための時間をオーバーヘッドという。

(2) 快適な使用環境 (ユーザインターフェース) の実現:

①汎用性の向上

- 会話型処理, リアルタイム処理など様々な処理形態を (同時に) 受け入れる。

②応答時間の短縮

③使い易さの向上

- 一般ユーザが快適にコンピュータを使える環境を提供する。

例えば、
マウス, アイコン, ウィンドウシステム, ... による GUI。

- 実際のハードウェア環境への依存性をできるだけ少なくする。すなわち、計算機内部について詳細に知らなくてもよい様にする。

例えば、
プログラムの中で入出力を容易に実行できる。

補足：
入出力機器は機器毎に制御の仕方が違う。
$\Longrightarrow$ 入出力機器の細かな制御も個々のシステムプログラマに任せてしまうというのでは、プログラマの負担が大きくなる。
$\Longrightarrow$ 入出力などの共通機能を容易に実行できる環境をハードウェアに付随して提供すれば、プログラマの生産性向上につながる。 これが初期の OS の目的であった。

- マニュアル (i.e. 取扱い説明書) なしでもシステムからの指示によって容易に使いこなせる様にする。
- 豊富な種類のプログラミング言語が用意されている。

- 希望者には、ジョブの実行を自分で管理する環境が設定される。
- 利用者の身近な場所からシステムを自由に利用できる。
- 運用の自動化・省力化が図られている。オペレーションが容易な様にする。
- プログラムの開発を支援する機能がある。
- 共同作業が容易にできる様になっている。

④拡張性の向上

- 機能の拡張がシステム全体に影響することなく局所的に容易に行える様にする。

⑤**RASIS**の向上

- **Reliability**(信頼性) ··· 故障しにくい。
- **Availability**(可用性) ··· 正常に稼働している時間の割合が高い。
  [故障しにくくても修理に時間がかかれば問題。]
- **Serviceability**(保守性) ··· 故障の際、修理/復旧が容易。
- **Integrity**(保全性) ··· 誤動作やプログラムの誤り等によってシステムが簡単に破壊される様なことがない。
- **Security**(機密性) ··· ソフトウェアやデータを本来の利用者以外か

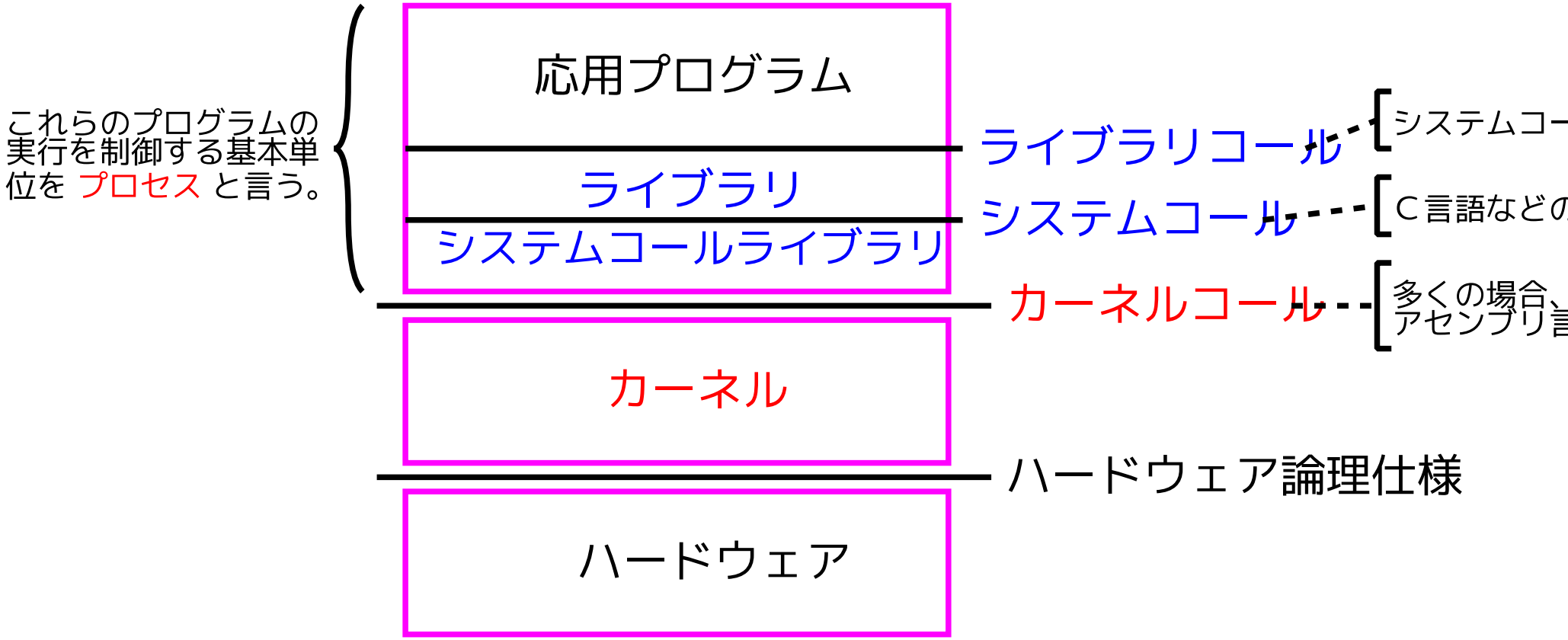
ら保護する。すなわち、盗用されたり破壊されたりしない様にする。

これらの目標の中には相反するものもあるが、これらは利用者の期待を全て満たす様にバランス良く達成されなければならない。

例えば、
使い易さや信頼性を十分に確保しようとすれば、その分だけシステムが重くなることがある。

13–5 オペレーティングシステムの構成

ソフトウェアの階層：



カーネルの機能：カーネルは次の4つの機能ブロックから成る。

- (1) システム制御 ・・・
  - 開始処理,
  - 終了処理,
  - 装置管理,
  - 障害の管理
- (2) 実行管理 ・・・
  - プロセス管理,
  - プロセス間通信管理,
  - メモリ管理,
  - 割込み制御,
  - プログラム管理,
  - 共通処理
- (3) 入出力制御 ・・・
  - 周辺入出力制御,
  - 通信制御
- (4) ファイル管理 ・・・
  - 入出力効率化のためのアクセス制御,
  - 外部記憶装置の領域管理,
  - ディレクトリ処理,
  - ファイル操作の機能を提供