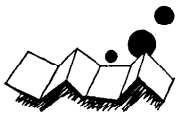
解　説



プログラム開発のための オペレーティングシステムの機能†

大和喜一†† 村井　純†† 斎藤信男††

1. はじめに

計算機システムを構成するハードウェアの能力の大小がプログラムの生産効率を決定する大きな要素であることはいうまでもないが，作業の分担やそれらの構築に関して，オペレーティングシステムや言語が提供する概念もプログラムの開発に大きく影響する．システム上での作業環境の良否は，その設計者が考案した概念の質と，それらがどのように整理されて利用者に提供されるかとにかかっている．

オペレーティングシステムが提供しなければならない基本的な要素であるタスクの提供，データの保管，入出力の処理などについては，すでに多くの場で整理され[1),2)]，いくつものシステムがそれぞれについて効率の向上や概念の統一に気を配りながら実現されてきた[3)]．

ここでは，これらの基本的な要素とプログラムとの関係を，利用者が要求する能力の提供という視点から考え，それらを実現しているオペレーティングシステムの例を示す．

1.1 プログラムに必要な機能の提供

プログラマがプログラムの設計を行うときに最初に考えることのひとつに，作業を遂行するために必要なデータ構造とそれを駆使するプログラムの流れがある．これら処理の各部を構成する要素の中には，処理手順やデータ構造を考えるときの一般的な材料となるものがいくつかある．これらのうち，一部はプログラミングに用いられる言語が提供する機構であるが，大部分は作業を行っている土台のオペレーティングシステムが利用者に与える機構である．また，言語が提供している機構のうちのいくつかについては，入出力やタスクに関する操作などオペレーティングシステムに操作を依頼するようになっているものもある．このように，プログラミングに必要とされる基本的な操作，機構などの要素の大部分はオペレーティングシステムが源となっている．したがって，これらの要素の構成次第で，プログラミングの柔軟性は大きく変わる．

一方，入出力や作業環境の設定など基本的な操作，機構でプログラミングに必要不可欠なものは数多くあるが，それらの各々について，システムごとに独自の実現方法が使われている．そのため，プログラムによっては，システムが提供する操作を利用するために多くのステップを要することもある．また，他のプログラムを作るときに用意した関数やプログラムを再度利用したいことも少なくない．これらのうち，比較的汎用のデータ構造に対する汎用の手続きを供用する方法には，ライブラリサブルーチンを使う方法が多く用いられているが，その他にもユーティリティプログラムを利用者のプログラム内からサブルーチンを使うように利用するための機能が用意されていて，これを用いて分類や探索のような操作の手続きを共用することもある．これらのシステムが提供したり利用者があらかじめ作成しておいた手続きを，核の提供する能力（システムコール）と同様に，利用できれば，利用者あるいは作業ごとにいろいろな応用に適したシステムを作り上げることができる．

それでは，このような能力を得るには，オペレーティングシステムにどのような機構が必要であろうか．以下のような機構を挙げておくことにする．

（1） 組み立てられる要素を，相互に干渉しないように働かせる機構：要素の組立て方は，それがサブルーチンか独立したユーティリティプログラムかによって異なるが，少なくとも現在実行中のタスク以外に別のタスクを駆動する機能があれば，個々の作業に分けて設計調整したプログラム群の間の干渉が減少し，分離したことによる利点が増大する．さらに，通常の実行可能プログラムだけでなく，プログラムを解釈実行するプログラム（コマンド処理言語のインタプリタ

† Operating System Functions Required in Programming by Kiichi YAMATO, Jun MURAI and Nobuo SAITO (Department of Mathematics, Faculty of Science and Technology, Keio University).

†† 慶応義塾大学理工学部数理科学科

など)の助けによって，この機構の応用範囲が広がる．

(2) 組み立てられる要素の間の相互関係を保つための通信機能と，そのためによく使われるファイルなど，入出力との間が単純に接続できる機構：(1)で述べたようにタスクを並行動作させるとき，それらの間の通信機能は，システムによって様々であるが，タスク上で動くプログラムに関係なしに整合させることのできる通信手段であることがのぞましい．

(3) プログラムが実行されているときの実行環境をプログラム中で認識できる機構，およびそれらを設定，変更できる機構：複数のタスクの各々で実行されるプログラムの間で，実行環境に関する共通性を得るためには，環境を知らなければならない．同様に，別別の環境で動いているプログラム間で，環境を一致させるためには，環境を設定，変更する機能がいる．

このような点を重視したシステムがいくつか実現されている．まず，ミニコン用として有名な UNIX[11] がある．これには，複数のタスクを自在に操るための道具が整っていて，操作を組み合わせる能力を持っている．次に，仮想計算機の概念を中心に構成された CMS[15] がある．これは，仮想計算機の各々で，それぞれプログラムを駆動して，それらの間の通信によって全体のプログラムを構成している．これらについては後の章で詳しく述べる．また，VOS (Virtual Operating System)[7] とよばれるオペレーティングシステムは，前に述べた UNIX が持つシステムコールやユーティリティの類のいくつかを別のオペレーティングシステム上で実現して，UNIX と同じ使い勝手を得ようとするものである[8]．このような他のオペレーティングシステムで新しいシステムを構成することで，目的に応じた操作能力を持つ要素を作り出す方法もある．

2. プログラム開発とシステムの機能

プログラミング効率の高い開発環境を求めて数多くのシステムが実現されてきたが，それらの具体的な目標は主に，適切な環境の設定，変更と状態の確認，効率が良く簡単な入出力，複数のタスクの効果的な制御などに向けられている．現実のシステムは広汎な作業に対してある水準を保って適応しなければならないので，これらの目標を達成するのに何通りもの策を講じることが多いが，システムに要求される基本的な機能はほぼ共通である．

2.1 環境の設定

プログラムの実行環境にはプログラマによって状況に合致するように変更できなければならないものがある．たとえば，時分割システムでは遠隔地にあって電話回線で接続されている利用者も，センタにあって直結されている端末を使う人と同様の高い操作性を得ることができるように，端末についての環境設定ができなければならない．また，システムのハードウェア構成が異なっても稼動できるプログラムを作成するには，ハードウェアの状態を含めた実行環境を自由に設定できる能力がシステムに要求される．

プログラムの実行環境を分類すると，次のようになる．

(1) ソフトウェア，ハードウェアを含めた，必要とされるシステムの構成．

(2) 実際に利用している装置，時刻など使用状況に応じて物理的に決定している状態．

(3) データやプログラムなど利用者ごとに区別され与えられる情報．

(4) ファイルの構造など遂行する作業に依存する情報．

これらの環境をプログラムの利用状況や目的に応じて制御する能力が大きいシステムでは，その上で開発可能なプログラムの幅が広がり，環境の変化に適応できるプログラムの設計が可能となる．

2.2 プログラムの入出力とファイルシステム

プログラムの開発時，デバッグのためのデータを与えるのに，本来特殊な入力装置から入力されることになっているのを，テストデータを保持するファイルを作ってそこから入力したり，端末から適当に与えてプログラムの挙動を確認することが多い．このように，1つのプログラムに対する入出力が実行状況に応じて変わっても，任意の装置やファイルから一様に入出力可能とするために，インタフェースを定めてそれらの切換えを行っている．これによって，実行状況を意識せずに，プログラム開発ができる．このように，切換えのために特にプログラムに注意を払う必要がないことが，プログラムの接続を容易にするために重要である．

2.3 多重タスクの利用と支援

複雑なプログラムの多くは，複数の分離できる作業が合成されて作られている．逆に，簡単なプログラムの追加によって，元のプログラムの機能を拡張したり，いくつかのプログラムの組合わせによって特別な

働きをする新しいプログラムを生み出すことができる．このように，小さなプログラムの集積でプログラムを作っていくとき，プログラムの組合わせを複数のタスクの並列利用に置き換えて考えると，既存のプログラムをそのまま変更せずに，単に組み合わせるだけで新しいプログラムを作ることができる．既存のプログラムに不足している機能を加えるときも，足りない部分を作って別のタスクでそれを実行する方法を採ることもできる．また，複数のタスクの上で簡単な作業するプログラムを動かし，相互に結合する方法を用いると，それらの間の相互関係のプログラミングによって，大規模で複雑なプログラムを容易に作ることができる．

3. 小型機におけるプログラミング環境

小型機の能力は，ここ数年で飛躍的に向上し，同時に利用者人口の増加と応用範囲の拡大が顕著であった．それにともない，プログラムの構成要素の設定やタスクの操作などの概念を持ったオペレーティングシステムが何種類か登場してきている．ここでは，時分割システム UNIX で実現されている環境設定の機能，入出力の方法，タスク（UNIX ではプロセスとよぶ）とその利用について解説する．

3.1 UNIX

UNIX は汎用の時分割システムとして，1969年ベル研究所で PDP-7 上にその原型の開発が始まって以来，現在に至るまで開発，変更が継続しているオペレーティングシステムである．PDP-11 の他，VAX-11，Interdata 8/32 に移植され，次々と拡張されて現在では，数千に及ぶ計算機上で稼動している．また，いろいろなマイクロコンピュータ上で UNIX またはそれに似たシステムが稼動している．

現在入手可能な UNIX システムには PDP-11 用として1975年に発表された第6版および PWB/UNIX と1978年の第7版，VAX-11 用としてベル研究所の UNIX/32 V 第1版 (1979) とカリフォルニア大学(バークレー)の 4.1 BSD (1981) などがある（バークレー版 UNIX は1982年7月に 4.2 BSD に改訂される）．また，UNIX SYSTEM III が1981年にベル研究所から発表されている．

UNIX の基本的な特徴は次のような点にある[11]．

（1） ファイル構造が階層化されている．

（2） ファイル，入出力装置上のデータ，プロセス間通信が同じ概念で統一されている．

（3） 非同期プロセスの生成，利用が自由である．

（4） コマンドの解釈実行プログラム (shell) が強力で，その取り換えも自由である．

（5） 言語プロセッサやユーティリティが豊富である．

（6） ポータビリティが高い．

これらの特徴を生み出す概念のうちのいくつかは，Multics[3] で実現されていたものである．

ユーティリティやライブラリを含めたシステムの大部分は言語C[12]で書かれていて，ポータビリティを高めシステムの修正を容易にしている．また，実現にあたって特別なハードウェアの補助を受けていないこともポータビリティを向上させることに役立っている[13]．

3.2 環境の設定

UNIX は次のような環境の設定能力を持っている．

・ ジョブで実行されるプログラムを指定する能力：UNIX では1つのジョブで実行されるプログラムは基本的に1つであるが，いくつものプログラムを次々と実行できるようにこの1つのプログラムをコマンドの解釈実行に用いるのが普通である（このプログラムを shell とよぶ）．この shell を利用者名に対応させて自由にとり換えることができる．

・ 利用者の作業範囲を限定する能力：他の利用者と隔絶した環境を作り，独自の作業場所を設定するために，利用者番号，グループ番号（その利用者が属する共通環境を持つ集団の識別子），ホームディレクトリ（各利用者の標準的な作業場所）が利用者ごとに与えられる．これによって，ファイルの参照範囲や，コマンドの実行範囲が限定される．

・ 利用する端末の能力に応じた状態を設定する：利用者が使っている端末装置の速度や機能に応じて，適正な動作をするようにあらかじめシステムに情報を与える．

これらは UNIX のジョブに関する環境設定の機能であるが，そこで実行される shell において，さらに詳細なレベルの環境設定が行われる．また，別にタスクを発生させて，その上で shell を動かすことによって，元の環境を保持したまま別の環境に移って作業を続けたり，複数の環境を同時に出現させてそれらを組み合わせて利用することができる．

これらの環境設定を達成するために，システムはいくつかのファイルを情報の保持のために用意している．

・ 利用者に関する情報：各利用者の環境の初期化の情報は，ほとんどパスワードファイルに保存されている．

・ 端末に関する情報：各端末の状態に関する情報は，ttys (/etc/ttys) に集まっていて，システムの初期化時に参照される．その他，端末の機能に関する情報は termcap, ttytype などに集められている．

・ shell やその他いくつかのプログラムが使う環境設定用のファイル：各利用者のホームディレクトリの下に .profile .login, .cshrc, .exrc, .mailrc などがあり，作業環境を整えるために shell などの実行開始時に用いられている．システムにおけるいろいろなソフトウェアの環境を変更するには，これらのファイルを修正すればよいことになっている．

shell の環境の設定は，タスクごとに存在する environ とよばれる，プログラムで変更可能なデータを利用して行われる．これらのデータは，プログラムの実行中または shell 自身によるコマンドの解釈実行中に参照され，それによって環境の状態が各々のプログラムに伝わる．

3.3 ファイルの構造と参照

UNIX は Multics 同様，木構造のファイル構造を採用している[10]．通常のファイルやディレクトリの参照はこの木をたどって行われるが，別の媒体上にあるファイルの参照をも同じ形式で処理するために木の接続が行われる．この mount とよばれる操作によって，2つの媒体上の2本の木が1つになり，別の媒体上のファイルも接続点を経由して参照できる．

すべてのファイルは同じ構造で，生成，読取り，更新，終端への追加は自由であるが，途中への挿入はできない．ファイルは機能の面から見て，次の3種類がある．

・ ディレクトリファイル：他のファイルを管理するためのファイル．

・ 通常のファイル (ordinary file)：プログラムやデータを保持するためのファイル．

・ 特殊ファイル (special file)：入出力装置やメモリなどを参照する名前を提供するためのファイル．重要なことは，特殊ファイルが通常のファイルと全く同じ操作で読み書き可能なことである．すべてのファイルは見かけ上あたかも一次元配列のように先頭から終端まで一様につながっていて，どの部分でも何バイト単位でも読み書き可能である．この単純さのためにプロセス間通信や端末を対象とする入出力とファイルの入出力が同じ操作で実行できるようになっている．

ディスク上のデータは，木構造をたどるか，特殊ファイルを使うかのいずれかで参照されるが，どちらで参照しても，相当するファイル名から得られる情報によって自動的に参照方法が切り換わるので，プログラマは対象となるファイルについて意識しなくてもよい．

入力と出力を処理データの流れ (stream とよぶ) と考えるとき，プログラムをデータを変形するフィルタと考えると UNIX のファイルシステムを有効に生かすことができる．プログラムに対する標準的なデータの入口，出口 (UNIX ではそれぞれ標準入力，標準出力とよばれている) にファイルや他のプロセスの stream を接続することで，いくつもの別々に作ったプログラムを組み立てることができる．shell は，これを利用して，1行のコマンドで何段ものフィルタを重ねて用いることを可能にしていて，これがコマンドの記述能力を強力なものとするのに役立っている．標準入出力は各タスクに1つずつであるが，入出力は必ずしも1つずつとは限らないので，標準入出力以外の stream を処理するためのライブラリが完備している．UNIX の標準 shell は複数の stream の扱いには向いていないが，それを拡張することは容易である．

3.4 プロセス

システムの稼動や利用者の作業を遂行するために，常に次のようなプロセスが存在する．

・ プロセスのスケジューリングを行うプロセス．

・ 端末を利用可能にする初期化 (init) プロセス．

・ 端末ごとの，コマンド処理用プロセス．非使用時には，ジョブの受付け (login) が実行されている．

・ コマンドの実行や利用者のプログラムの実行によって生成されたプロセス．

・ デーモン (daemon) とよばれる，spool やファイルシステムの更新などのために常時存在するプロセス群．

図-1 にこれらのプロセスの関係を示す．プロセスの生成は複製を作る操作 (fork) によって行われ，どのプロセスも平等に新しいプロセスの生成ができる．作る指令を出したプロセスを親プロセス，作られたプロセスを子プロセスとよぶ．

プロセスに関する機能の中で，プロセス間通信の機能の良否はプログラムの記述に強く影響する．

pipe：待ち行列の入口と出口を別個のプロセスの出力と入力にそれぞれ接続して使われ，満杯になると出

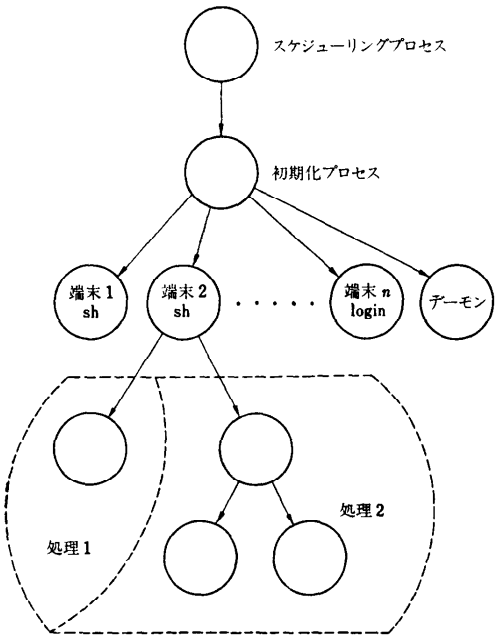


図-1　プロセスの生成関係

```
char buffer[10];
main()
{
    int p[2],pid,status;

    pid = fork();
    if(pid == 0){ /* child process */
        syori1();
        write(p[1],buffer,10); ------①
        exit(0);
    } else { /* parent process */
        read(p[0],buffer,10); ------②
        syori2();
        wait(&status);
    }
}
```

図-2　pipe による通信と同期

力側のプロセスが一時停止し，空になると入力側のプロセスが一時停止する．図-2 に pipe を用いたプロセス間の通信を行うプログラムを示す．このプログラムは，子プロセス（fork の値が 0 の方のプロセス）で syori 1 の実行によってつくられた結果が buffer に入っていて，それを親プロセスに pipe で渡し，syori 2 で使うというものである．①と②が pipe に対する出力と入力で，同期の機能により，①によって pipe が空から 10 文字満たされるまで②は完了しない．したがって syori 2 は syori 1 の後で実行される．

signal：プログラムの誤りにともなう割込みや端末からの割込みをプロセスに通知するために利用されるが，既知のプロセス間における信号として使われることも多い．プロセスに signal が到着したときに実行する手続きをあらかじめ定義しておけば，非同期の手続きの実行が可能となる．

プロセスの持つメモリ空間の内容は，生成時には親プロセスの複製となっているので，子プロセスで全く別のプログラムを実行するにはその手続きを空間にロードする機能が必要である．そのために，UNIX には exec というシステムコールが用意されている．プログラムの実行は fork とこの exec によって行われ

```
Program a.c

main()
{
    int status;

    if(fork()==0)
        execl("/bin/sh","sh","b",0);
    else
        wait(&status);
}

Program b

sh c | tee f | tr "a-z" "A-Z"


Program c

echo before e
e
echo after e

Program e.c

main()
 {
    printf("Program e\n");
 }


Program main.c

main()
{
    int status;

    if(fork()==0)
        execl("a","a",0);
    else
        wait(&status);
}

$ main
BEFORE E
PROGRAM E
AFTER E
$ cat f
before e
Program e
after e
$
```

実行結果

fの内容

図-3　コマンドと shell によるプログラムと実行結果

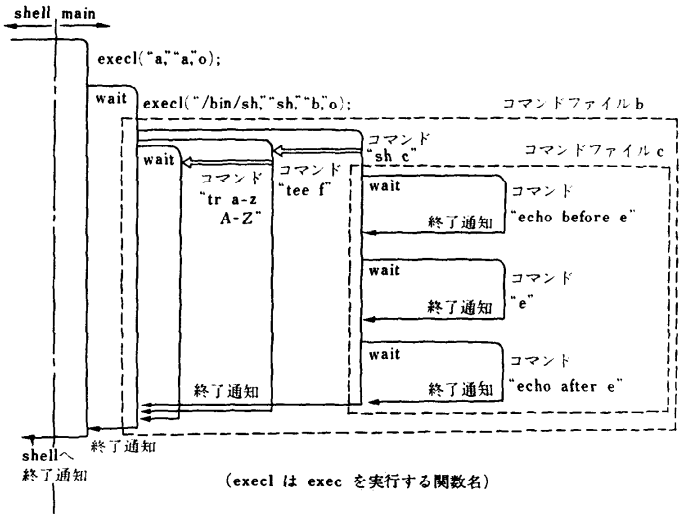


図-4 プロセスの関係とプログラム

る．そのため，pipe や signal などプロセスに結びついたものは，単に応用プログラムの中だけで使われるのではなく，むしろコマンドのレベルで頻繁に活用されていて，このことが UNIX におけるコマンドの柔軟性を高くしている．

プログラムの要素を構成するもう1つの材料に shell がある．shell は前に述べた通りコマンドの解釈実行用のユーティリティプログラムであるが，コマンド列をプログラミングの1つの要素として実行させるための道具としても重要なものである．**図-3** に shell を使ったコマンドファイルの実行と実行可能プログラムの混在したプログラムの例とその実行結果を，**図-4** にこのプログラムにおけるプロセスの関係を示す．この例では，main.c, a.c, e.c を翻訳した実行可能プログラムをそれぞれ，main, a, e とし，b と c はコマンドファイルとしている．main ではプロセスを生成して a を実行し，a ではプロセスを生成して shell によってコマンドファイル b を実行している．b では sh c (shell による c の実行) と tee と tr が pipe (1)でつながっていて，まず "echo before e" によって "before e" が pipe をつたって tee に行く．tee はパラメータで指定されたファイル f にそれを保存するとともに次の pipe へ再び "before e" を送り出す．これを受け取った tr は小文字を大文字に変えてターミナルに出力する．これが続けられて結果のようになり，ファイル f には小文字の文字列が保存される．

このようなコマンドと実行可能プログラムの混在によって，プログラムの生産効率は上昇し，部分的な変更や，それによる効率の向上のための作業も容易である．

4. 大型機におけるプログラミング環境

大型計算機の多くは数多くの作業を処理すべき環境に置かれていて，ハードウェアを直接操作するようなプログラムの開発は多大な困難をともなう．このような問題は，実計算機をかなり忠実にシミュレートして複数の仮想的な計算機の機能を提供するオペレーティングシステムの実現によって解決することができる．また，このようなシステムの持つ能力を十分に活用すると，これまでほぼ不可能であった応用を効率良く実現することができる．

4.1 VM 370[14]

IBM の VM 370 時分割システムは，同社のシステム 370 上で動くオペレーティングシステムであり，IBM 系列の大型機用オペレーティングシステムの中では会話型プログラミング環境を最も強力に支援するものの1つである．VM 370 の利用者は，システムが持つ全資源を自由に使用することができるので，専用に計算機を与えられた場合と同様な機能の提供を受けることになる．これら利用者に与えられる資源や計算機システムの機能は，VM 370 によって単一計算機システムの上でシミュレートされて提供される．シミュレートされることによって出現する計算機を仮想計算機 (Virtual Machine) とよぶ．

仮想計算機の実現によって，各利用者は全く孤立した環境のもとでプログラム開発を行うことができる．また，必要に応じて明確にそれを表現することによって，仮想計算機間の通信や情報の共有の機能を利用することができる．

VM 370 の主な構成要素は，実計算機の全資源とすべての仮想計算機を管理するための制御プログラム CP (Control Program) と仮想計算機上で動くオペレーティングシステムである会話型モニタシステム CMS (Conversational Monitor System) である．

CP は，システム 360, 370, 43XX, 303X 等のプロ

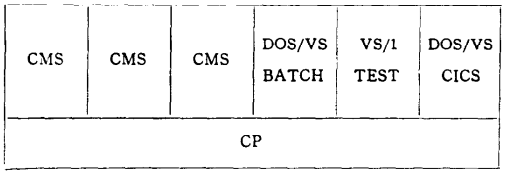


図-5 CP と CMS

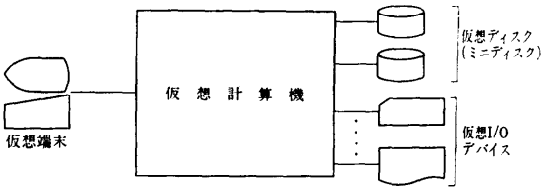


図-6 仮想計算機システム

セッサを仮想計算機の対象とし，それらの基本的なハードウェア機能をシミュレートする．

CMS は CP によって実現される仮想計算機の上で動くオペレーティングシステムの1つで，単一利用者，単一タスク用の会話型モニタシステムである．CMS は VM 370 の標準的なオペレーティングシステムで，ファイルシステム，コマンドインタフェース，言語プロセッサなどから成るプログラム開発環境を提供する．CP は CMS 以外に DOS[18]，OS[19] など，システム 360, 370 上の他のオペレーティングシステムを駆動することもできる[20]．

CP における仮想計算機の共存は，優先度と時分割を利用した多重プログラミングによって実現されている．図-5 に VM 370 における CP と CMS など仮想計算機上のオペレーティングシステムの関係を示す．

4.2 環　境

仮想計算機システムの最大の特徴は環境設定の柔軟さである．VM 370 では，各利用者に次のような環境が与えられる(図-6)．

- 仮想ハードウェア：1つの仮想計算機とそれに付随するコンソール，仮想カードリーダ，仮想プリンタ，仮想ディスク(ミニディスク)などの入出力装置．
- 会話型モニタシステム．

この環境はシステムに登録された利用者名に対応して与えられ，原則として他の仮想計算機から孤立しているので，利用者ごとに各自のパーソナルコンピュータの環境が与えられるといえよう．

端末から利用者が入力するコマンドは，CMS へのコマンド，CP と交信するためのコマンドの2つに分かれる．CP コマンドによって仮想機械を初期化 (logon) し，CMS の初期ロード (IPL) を行ってプログラム開発に必要な環境を初期化することができる．CP コマンドはこの他にも仮想計算機自体の設定や変更，デバッグなどに使われる．

CMS は仮想計算機上の他のオペレーティングシステムと異なり CP 専用のオペレーティングシステムである．CMS においては，他の IBM システム同様に言語プロセッサ，テキストエディタ，文書処理などを利用することができる．また，ファイルシステムは各仮想計算機に属するミニディスク上に独特の形式で存在する．ミニディスクは他の利用者と共有したり，容量，個数の変更が可能である．このファイルシステムとそれらを管理するユーティリティの充実が CMS の環境を使いやすくすることに大きく貢献している．

4.3 ファイルシステムとプログラムの入出力

CMS のファイルはミニディスクごとのディレクトリによって管理されている．ミニディスクは自由に仮想計算機に接続することができ，もしそれが他の利用者が所有するディスクであってもパスワード方式の保護手続きを踏むことによって利用可能である．接続されたディスクのディレクトリを仮想記憶上に常駐させることによってコマンドやプログラムからディスクの参照ができるようになる．

ファイルは，ソースファイル，コマンドファイル，実行可能ファイルなどいくつかのタイプを持っているが，**EXEC**[15] とよばれるコマンドファイルと，実行可能ファイルは，その環境だけで利用できるコマンドとして実行することができる．EXEC ファイルには，CMS や CP のコマンドの他にコントロールの流れを記述する制御文を含むことができる他，コマンド引数や変数も取り扱うことができる．また，CMS のコマンドはコンソールのスタックとよばれるバッファからシステムによって読み込まれて処理されるが，スタックに投入する入力行の列をあらかじめ EXEC ファイル内に記述することもできる．これによって，コンソールからの入力を要求するプログラムに対する入力行の列（たとえば，エディタのコマンド列など）をコンソールスタックへ投入してプログラムを駆動するといった手順を1つの EXEC ファイルで実現することができる．

プログラム内でのファイルの読み書きは，MVS[21] など IBM 系列のオペレーティングシステムと同様に，あらかじめ定義された論理名によって行われる．コンソールへの出力をコンソールスタックに積むこと

によって，複数のプログラムを関数的に結合した複合処理を実現することもできる．

4.4 多重プロセッシング

1つの仮想計算機は，CP によって管理されるタスクの1つであると考えることができる．したがって，多重プロセッシングを CMS の環境で実現するためには，複数の仮想計算機を利用して，それらの間の通信機能を活用することになる．

各仮想計算機は必ずしも端末からの logon コマンドによる初期化をする必要はなく，**autolog** コマンドによって他の仮想計算機から初期化することができる．これは，子タスクを生成する attach コール，あるいは UNIX における子プロセスを生成する fork コールに対応するものである(**図-7**)．

このようにして生成されたバックグラウンドの仮想計算機は，自動初期設定の機構によって目的のプログラムの実行を開始する．

この親仮想計算機と子仮想計算機とのデータの交信には3つの方法がある．1つはスプールファイルによる方法，他の2つは **VMCF** (Virtual Machine Communication Facility) と CTCA (channel-to-channel adaptor) と呼ばれる CP の機能を用いた方法である．1つめの方法は，送信側の仮想計算機の出力装置に対するデータを，スプールファイルの形で受信側仮想計算機の入力装置に転送する方法で，RJE やコンパイル等，バックグラウンド処理のサービスをする仮想計算機との情報転送に用いられる．このような仮想入出力装置の性格の変更は CP コマンドによって容易に実現できる．**図-8** に仮想計算機間の情報転送を行うために CP コマンドによって仮想カードパンチをログオン名 yamato の仮想計算機のカード読取り機へスプールファイルを介して結合させた例を示す．スプールファイルによる仮想計算機間の情報転送は，二次記憶上のファイルを経由して行われるので，論理的に明確で安全な方法ではあるが，オーバヘッドが大きい欠点がある．

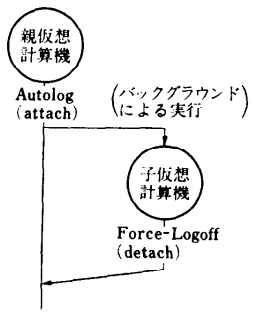


図-7 仮想計算機の生成

```
cp spool punch cont to yamato
punch file1 fortran
punch file2 fortran
punch file3 fortran
cp spool punch nocont
cp close punch
```

図-8 通信を行うコマンドの列

これに対し，VMCF は仮想計算機の特殊な命令によって仮想計算機間の情報転送を実現するもので，実際の転送は定められたプロトコルにしたがった仮想記憶間でのメッセージやデータの転送によって行われる．そのために，転送速度が速く，より柔軟な制御を記述することができる[16]．

また，CP コマンドによる仮想入出力装置の設定の機能を用いて，2つの仮想計算機間に，仮想チャネル-チャネルアダプタを設定し，それに対する入出力を行うことによって，仮想計算機間の情報の転送を実現するのが3つめの方法である．この方法は，CP コマンドにより転送先を容易に変更することができるという特徴がある．

以上に述べたような仮想計算機間の情報転送の機能を用いて，仮想計算機の機能を拡張することによって，次のような応用が可能である．

- 資源の共有に関する制御．
- 多重プロセッシング．
- 各種のテストや処理を監視するモニタプログラムの実現．
- ネットワーク機能．

5. おわりに

プログラムの効率の良い開発を支援するシステムの持つべき機能について，2つのオペレーティングシステムを例にとって述べたが，それぞれに幅広いプログラムの開発を可能とする環境設定能力を保有してい

る．プログラムの開発，実行に好適でかつ必要な環境を設定する能力と，その環境で自由な処理を可能にする機能がプログラム開発を支援するシステムには不可欠である．これらの機能は多少なりともこれまでのシステムにも含まれていたものであるが，この2つのシステムでは特に強調されているようである．そのために，時分割システムの最大の目的であるシステムの能力の自由自在な利用の達成に，さらに一歩近づいたといえよう．

紙数と視点に関する限定から，両システムのわずかな部分を述べたにとどまったが，数多くの文献が広範囲に紹介を試みているので，それらを参照願いたい．

参考文献

1) Haberman, A. N.: Introduction to Operating System Design, Science Research Associates (1976).
2) Madnic, S. E. and Donovan, J. J.: Operating Systems, McGraw Hill, Inc. (1974).
3) Corbató, F. J. and Vyssotsky, V. A.: Introduction and Overview of Multics System, AFIPS Conf. Proc. Vol. 27, pp. 185-195 (1965).
4) Strachey, C.: Time-Sharing in Large, Fast Computers, Proc. Int. Conf. on Info. Processing, UNESCO, pp. 336-341 (1959).
5) Corbató, F. J. et al.: An Experimental Time-Sharing System, SJCC, pp. 335-344 (1962).
6) Brooks, F. P.: The Mythical Man-Month, Addison-Wesley (1975).
7) Hall, D. E., Scherror, D. K. and Sventek, J. S.: A Virtual Operating System, Comm. ACM., Vol. 23, pp. 495-502 (1980).
8) Kernighan, B. W. and Plauger, P. J.: Software Tools in Pascal, Addison-Wesley (1981).
9) Parmelee, R. P. et al.: Virtual Storage and Virtual Machine Concepts, IBM System Technical Journal (1972).
10) Tompson, K.: UNIX Implementation, The Bell System Technical Journal Vol. 57, pp. 1931-1946 (1978).
11) Ritchie, D. M. and Tompson, K.: The UNIX Time-Sharing System, CACM, Vol. 17, pp. 365-375 (1974).
12) Kernighan, B. W. and Ritchie, D. M.: The C Programming Language, Prentice-Hall (1978).
13) Ritchie, D. M.: A Retrospective, The Bell System Technical Journal, Vol. 57, pp. 1947-1969 (1978).
14) Auslauder, M. A. et al.: The Origine of the VM/370 Time-Sharing System, IBM J. Res. Develop. Vol. 25, No. 5 (1981).
15) IBM VM 370: CMS User's Guide, IBM (1977).
16) Jensen, R. M.: A Formal Approach for Communication between Logically Isolated Virtual Machines, IBM System J. Vol. 18, No. 1 (1979).
17) IBM VM 370: System Programmer's Guide, IBM (1977).
18) Bender, G. et al.: Function and Design of DOS/360 and TOS/360, IBM System J. Vol. 6, pp. 2-21 (1967).
19) Mealy, G. H.: The Functional Structure of OS/360, Part I: Introductory Survey, IBM System J. Vol. 5, pp. 3-11 (1966).
20) IBM Virtual Machine Facility/370: Operating Systems in a Virtual Machine (1979).
21) Scherr, A. L.: IBM Virtual Storage Operating Systems, Part III: OS/VS 2-2 Concepts and Philosophies, IBM System J. Vol. 12, pp. 382-400 (1973).

(昭和57年1月18日受付)