解　説

ソフトウェア使用における人間的要素

オペレーティング・システム†

森崎　正人††

1. はじめに

パーソナルコンピュータから超大型コンピュータまで，オペレーティング・システム (OS) のない計算機はいまや存在しなくなっている．さらに現在では，“シリコンOS”という名称まで現われ，オペレーティング・システムそのものがハードウェアの一部として扱われるようになってきた．その一方で利用者は，オペレーティング・システムを取り囲むように整備されているアプリケーションプログラム，ユーティリティ，ランタイムパッケージ等のおかげで，オペレーティング・システムそのものを直接に意識する必要がなくなっている．

最近のミニコンピュータ以上の計算機でオペレーティング・システムを直接意識する必要がある人は，パッケージ等を作成するシステムプログラマと，オペレーティング・システムを含めたシステムプログラムをそれぞれの計算機サイトごとに調整するシステムマネージャ，そして日常業務において計算機システム全体の運用を監視するオペレータぐらいのものである．オペレーティング・システムという巨大なソフトウェアの使いやすさあるいは使いにくさを細かいところまで意識せざるを得ない立場にあるのは，これらの人々である．

筆者は現在運悪くミニコンピュータ上の TSS システムにおいてこの三役を1人で請負っている．この経験を中心に，それぞれの立場からオペレーティング・システムのいかなる機能にかかわりを持つかについてふれ，私見ではあるが，オペレーティング・システムに期待する機能について述べる．解説の筋立てとしては，システムプログラマ，システムマネージャ，オペレータの目から見たオペレーティング・システムの使い心地についてふれ，最後に，オペレーティング・システムが持つべきバランス感覚ともいうべきものをさぐってみたい．

2. システムプログラマの目から

（1） 端末機種からの独立性

業務プログラムを作成するとき多くの工数を要するのがユーザ端末とのインタフェース部分である．対象業務がオンライン・アプリケーションであり，機種の異なる多くの端末を収容する必要があるときは各機種ごとに個別に端末インタフェースを作成することになる．

これらインタフェース・プログラムは，同質の構造あるいは仕様となっているが，細かい書式制御の部分（特に，ディスプレイ端末の場合）が異なる．書式制御コード・シーケンスの相違や，端末の印字速度，リフレッシュタイム等のタイミングの問題が関与する．さらに，既存のアプリケーションにおいて対象機種を広げる場合，新たな端末インタフェースの追加作成が必要となる．

一般に，オペレーティング・システム，アプリケーション・プログラムのライフタイムより端末機器のそれははるかに短い．システム・プログラマからは書式制御部分をアプリケーション・インディペンデントにオペレーティング・システムに追いやるのが一番である．しかし，オペレーティング・システムに移った機能やルーチン構成が従来と同一発想である限り，上で述べたライフタイムの問題を解決するどころか，より問題の根を深くする．結局は，次の新しいアプリケーションを考えるとき，オペレーティング・システムに新端末のサポートがないということが発生し，またアプリケーションサイドで新たにインタフェース・プログラムを作ることになる．

端末機能とプログラムを分離する方法としては

† Operating Systems by Masato MORISAKI (Yokosuka Electrical Communication Laboratory).

†† 日本電信電話公社横須賀電気通信研究所データ通信研究部データ通信研究室

UNIX オペレーティング・システムで行われているタームキャップ (termcap : terminal capability data base) のようなものが一つの解決策ではないかと考える[1]. これは端末機能に関する諸々の情報をオペレーティング・システム内にデータベースとして持つものである. プログラムはデータベースに記述されているメタシンボルを用いて書式制御を含む端末インタフェースを記述する. 端末ユーザの側では新しい端末の情報をこのデータベースに登録する. 一般に新機種の端末は書式制御コードシーケンスの相違はあるにしても機能的に拡張の一途をたどっているわけであるから, 少なくとも従来端末で扱えたアプリケーションは新機種でも使えることになる.

(2) 論理名とリソースアクセス

オペレーティング・システムはシステムのリソース管理のため, 計算機システムを構成する種々のデバイスを物理名 (physical name) (デバイス名やファイル名から構成される.) というシンボルを用いてシステム内でユニークに管理している. 一方プログラムは, 一般に, 仮想入出力チャネルを表わすものとして論理名 (logical name) を用いる. 論理名には, プログラムの外で次のルールにより物理名をアサインする. ディスク等ファイル構造を持つものはデバイス名とファイル名からなる物理名を, ファイル構造を持たないデバイスは直接デバイス名をアサインする. しかし, 直接物理名を扱う必要があるときのため, 最近のオペレーティング・システムはプログラムのうちからデバイス名, ファイル名を指定することにより, 論理名という仮想チャネルを用いないアクセス方法を許しているものが多い.

最後に述べた機能が現われたため, 経験を積んだプログラマとそうでないプログラマで, 論理名を使う人, 使わない人というふうに大きな差が出ている. 経験の浅い人ほど単に手数がかかるということで直接に物理名を使いがちである. 論理名を使うということはプログラムを実際のリソース (ファイル等) から独立させるということで環境変化に強いプログラムとなることを知らないのである. こういう使い方において, オペレーティング・システム自身が変化に強くなる方法はなかろうか?

システムグローバルな (ファイルを含めた) リソースについては方法がある. システムグローバルな論理名さえもユーザには物理名と思わせてしまうのである. そしてシステムマネージャ側で環境変化に合せアサイン先を変えるようにするのである[2]. それではユーザが定義する論理名との関係をどうするか? 論理名の多重化を許すのである. 逆に, ユーザにはグローバルな論理名しかわからないのであるから, 物理名を知りたいときどうするか? オペレーティング・システムそのものが多重に定義されている論理名から現実の物理名へ翻訳する機能をユーザ (つまりアプリケーション) に簡易な形式で開放するのである[2]. 今日的オペレーティング・システムとしてほしい機能の1つである.

(3) JCL/コマンドとデフォルト

(2)で述べた論理名と物理名に関してデフォルトの問題がある. IBM 系のオペレーティング・システムではプログラム中の FCB (File Control Block) でファイル情報を定義する[3,4]. この情報は一般に新規ファイルを作成する場合のデフォルト値として用いられる.

一方, JCL のデータセット定義文やマウントコマンドではパラメータ値とデバイスのヘッダ情報の照合を行い, オペレーティング・システムのファイル管理で必要となる不足情報をデバイス上のヘッダから取ってくる. そしてこれら情報と FCB の情報をマージングする. このときの値の相異については,

デバイス>パラメータ>FCB

の順で優先度を決めていく. つまり, デフォルト値はデバイスのそれに従って定まり, プログラマは未確定なデフォルトを意識したプログラム作成を行う必要がある.

このメカニズムは複雑なものであるが, 使い方およびデバイスの多様性に対応がとれるということですぐれたものである. ところが最近のユーティリティは高級言語で書かれたものが多く, 言語処理系の入出力機能の貧弱さからオペレーティング・システムの機能を十分に生かしきれていないケースが多い.

3. システムマネージャの目から

(1) システム・ジェネレーション/システム・チューンアップ

システムマネージャにとってシステム・ジェネレーション, システム・チューンアップは一種の賭である. オペレーティング・システムの真のパワーを引き出せるか出せないかはこのときの作業にかかっている.

現在のオペレーティング・システムはほとんどパラ

メータ形式で各計算機サイトのシステム構成に合せたジェネレーション・パラメータ値を設定する．ミニコンピュータのオペレーティング・システムでも160個ぐらいのパラメータがある*．これらパラメータは

ワーキング・セット・サイズ
↔ スワッパ・マップ・サイズ
↔ システムページ・テーブルサイズ

というぐあいに相互作用を及ぼす．この相互作用をシステムマネージャが的確に把握するには職人芸的な一種の勘を必要とする．

良くできたオペレーティング・システムでは，パラメータのうちメジャな項目の値を変えるとシステムが自動的に他のマイナ項目を修正する機能をもったものもある[2]．これはこれで非常に便利な機能であるが，この種の自動化はオペレーティング・システムの作成者の意識が大きく関与しており，意外に各計算機サイトのシステムマネージャの意識とのずれが大きい場合が多い．たとえば，ファイルアクセスの高速化をねらう場合と，CPU 処理の高速化をねらう場合とでパラメータ設定の戦略は異なる．意識のずれが起こるのは各サイトの最大公約数が標準パラメータだからである．システムマネージャに役立つのはフルオートやセミオート化でなく，むしろマニュアル操作のアシスト機能である．どのマネージャも各計算機サイトの特性を一番よく把握しており，それに合せたシステム・ジェネレーション，システム・チューンアップを行いたいものである．

筆者が望むのは，1つのパラメータ値を変更したとき，変更によって影響をうけるパラメータについて推奨値，危険値を表示してくれるものである．危険値の表示がないため，効果もなくパラメータをあれやこれや小刻みにいじってしまい逆に悪い結果へ導くはめになる．

（2） ジョブ特性の把握

システムマネージャにとって管理している計算機システムのジョブ特性全体を把握する意義は，システムのクラッシュを未然に防ぐこととシステムの効率化を妨げるボトルネックの発見に役立つ．通常，システムマネージャはジョブ特性について次の2つの観点で把握する．

ⅰ）アベレージ特性——システムの効率的利用をはかるため．

ⅱ）トランジェント特性——安易に，システムの過負荷状態を作り出さないため．

* 世の中の大型機ではシステムパラメータが1000を超すものがある．

VAX/VMS Monitor Utility
PAGE MANAGEMENT STATISTICS
24-JAN-1983
11 : 23 : 08

	CUR	AVE	MIN	MAX
Page Fault Rate	3.66	3.66	3.66	3.66
Page Read Rate	0.66	0.66	0.66	0.66
Page Read I/O Rate	0.33	0.33	0.33	0.33
Page Write Rate	0.00	0.00	0.00	0.00
Page Write I/O Rate	0.00	0.00	0.00	0.00
Free List Fault Rate	2.00	2.00	2.00	2.00
Modified List Fault Rate	1.00	1.00	1.00	1.00
Demand Zero Fault Rate	0.33	0.33	0.33	0.33
Global Valid Fault Rate	0.00	0.00	0.00	0.00
Wrt In Progress Fault Rate	0.00	0.00	0.00	0.00
System Fault Rate	0.66	0.66	0.66	0.66
Free List Size	871.00	871.00	871.00	871.00
Modified List Size	35.00	35.00	35.00	35.00

図-1 オペレーティング・システムのモニタリング結果例[2]

システムマネージャにとって上記2つのうちⅰ）の情報を得るのは比較的やさしい．オペレーティング・システムに特別の細工を施さなくてもユーザごとのアカウント情報をもとに，各リソースの使用特性をグラフに描いてみればボトルネックとなる問題点のありかがわかるものである．

ⅱ）のトランジェントなものについても，最近のものはオペレーティング・システム内にモニタリング機能が付加されている．図-1に一例を示す[2]．しかし，この種のものはオペレーティング・システムの動きに合せて内部の動作を表示しているのみでオペレーティング・システムのチューンアップ等へは直接結びつかない．というより，結びつけるためにはシステムマネージャの経験が必要である．（1）で述べたように，システムマネージャがオペレーティング・システムに対して影響を及ぼせる唯一の道具がシステム・ジェネレーション・パラメータのみであることと，オペレーティング・システムの状態値をいくら把握してパラメータをいじっても今日的システムに共通する自動調整機能[2]のおかげで手をほどこす術が限られているのである．

では，システム・マネージャはどうするか？ この解答はしごく簡単であってモニタ情報をもとに各ユーザの頭を叩いてまわるのである．「計算機システムのモニタリング結果ではオペレーティング・システムのこの点に負荷が生じている．この負荷を生じさせたのはおまえらのプログラムが悪いのであって，なんとかしろ．」と……．

横道にそれたがこれはオペレーティング・システムのモニタリングが意味をなさないことをいっているのではない．モニタリング結果からシステム・ジェネレーション・パラメータ値へと結びつける方法論，およびこれを助けるオペレーティング・システムの機能を求めているのである．細かいところをみると，急激なワーキングセット増加を防ぐための自動ワーキング・セット調整機能*[2),5)]や，イベント処理でプライオリティを動かす機能はすでにそなわっている．しかし，大局的パラメータのベストな設定をアシストするものはないという点である．

（3） パブリックファイルのプロテクション

ファイルのプロテクションに関してはオペレーティング・システムによって管理哲学がまったく異なっている．パスワードで管理するものもあれば，UNIX のように 10 bits ほどのパーミッション[1)]で管理するものまでいろいろである．しかし実際にシステムを使ってみると分かるように，パブリックファイル（公用ファイル）に対してこれらの効果は小さい．

ほとんどの場合，パブリックファイルのディレクトリは全ユーザに対して開放されており，個々のファイルについても実行可かリード可のパーミッションを与えることになってしまう．大型機でもロードモジュールファイル**は実行可のみでガードができるが，それが実行時に参照するシステムのデータファイルについてはスタティックにリード可のパーミッションを付ける必要が生じる．この結果，ユーザからは別の手段でこのファイルの内容を暴くことが可能となる***．

ガードの簡易なやり方として UNIX の Set-Uid ビット（Sビットと呼ばれることが多い．）の方法がある[1)]．ロードモジュールファイルにSビットを立てると，それがプロセスとして動くときオーナの Uid となるため参照ファイルに対しては他ユーザへの許可をなにも与える必要がない．

筆者の考えとしては，パブリックファイルについては，それがあるディレクトリも含め，中身を極力直接的に開放すべきでなく，関連するプログラムを通じて見せるべきと考える．

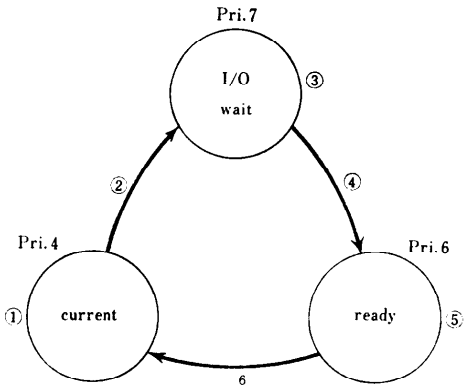
4. オペレータの目から

（1） ジョブの動的把握とスケジューリング

ジョブのスケジューリングの基礎となるのはプライオリティ操作である．最近のオペレーティング・システムのスケジューリング機能は非常に複雑になるとともに，ほとんど人間の介入を要しない．各ジョブのベースプライオリティを基に，ジョブのイベント状態に応じてプライオリティを変化させ，CPU の利用効率を最高にもって行く手法[2)]が多くとられている（図-2 参照）．このため，オペレータの仕事は計算機システムの効率的運用操作からシステムの異常監視へと大きく変わった．しかし過負荷状態における危険回避については立場が逆転する．

ほとんどのオペレーティング・システムでは過負荷対策がよく施されていて，過負荷がシステムダウンにつながることはまずない．しかし，過負荷状態を下げる操作は逆に上で述べた自動機能のおかげで非常に難しくなっている．

TSS 的使用状態での過負荷要因とその対策のうち，CPU バウンドの長時間ジョブが第1次要因で残留ジョブが積滞し，これに不安を感じたユーザがターンアラウンドを短かくするために一度に多量ジョブを投入



［説明］
① ユーザ・プロセスの実行中
（ユーザのベースプライオリティで動く）
② I/O イベントの発生
③ ユーザ・プロセスをウェイトさせる
（I/O 処理の完了通知を受けるため高プライオリティで待つ）
④ I/O イベントの完了
⑤ ユーザ・プロセスの実行待ち
（ディスパッチを速くするため，高プライオリティに設定される）
⑥ 早い契機でディスパッチされる

図-2 プロセスプライオリティの変化のモデル

* ページ・フォルトの割合に応じ，ワーキング・セットを調整する機能．急激な増加，減少はオペレーティング・システムの負荷となるため，CPU クォンタムを契機に少しずつ増したり，減したりする．
** エギュゼキュータブルファイルともいわれる．
*** 最近ではデータそのものの暗号化をオペレーティング・システムでサポートする研究が盛んである．

するために起る輻輳的過負荷の処置が一番いやらしい*．これをなくすには残留ジョブを早く処理してしまうことである．長時間にわたるシステムのリソースキープという，オペレーティング・システムからみて最も危険な状態（リソース・ハングアップを引きおこす．）を保留にしても，原因となるジョブを休止**させる操作である．ところが，オペレーティング・システムは長時間ジョブのリソース解放を促進させるため，CPU のあきをみつけてはＣＰＵクォンタム*** の間これを実行する．つまり，オペレータの意志とオペレーティング・システムの意志とが逆方向にむかう場合である．オペレーティング・システムのほとんどにおいてオペレータよりオペレーティング・システムの方が権力が強いため，オペレータの取り得るおおよその手段（たとえば，ロールアウト****，プライオリティの変更等）は効果が少なく，相変わらず過負荷が続くことになる．

（2） デバイスのマウント・ディスマウント

一世代前は（1）と同様にオペレータの主要な作業項目であった．人間が介在するということでオペレーティング・システムにおいてもメッセージの出力およびその応答についていろいろと工夫されている．全体的にみれば高密度ディスクの急速な進歩によりほとんど計算機システムの立上げ・立下げ時しかこれら操作の必要性はなくなりつつある．

デバイス操作に関しては本質的手順はあまり変わっていない．しかし，TSS の発展ということからセンタオペレータを介したデバイスマウント操作に新たな工夫が多くみられる．たとえば **図-3** はセンタオペレータ・アシストを含むマウント要求の一例である[2]．ユーザからはマウント要求と一緒にオペレータへの指示メッセージが送れる．マルチボリュームの場合など，オペレータはデバイスのマウント完了通知と同時にデバイスのイニシャライズが可能となっている．

* ジョブの入力制限を行っても，制限が解除されたときユーザは前より不安を感じて多量投入する．

** ジョブをキャンセルしてはいけない．キャンセルされたユーザは再度手を変え同一ジョブを投入するのがつねである．問題を後に回すだけである．

*** 意外と忘れられているのが CPU クォンタムである．ジョブがいくら低プライオリティでも CPU を握ったらこの時間はシステムのぬしである．システムジェネレーションとも関連するが，メーカの標準値はオペレーティング・システムのオーバヘッドを小さくするため，長目に設定されていることが多い．筆者の考えではクォンタムを短かくして，イベント完了待ちのジョブを早く解消する方がジョブの捌けということで安定性がいいと思っている．

**** プロセスの空きができると自動ロールインされるのが多い．

```
$
$MOUNT/ASSIST/COMMENT="New Tape."-
$_MSA0: MYTAPE
%MOUNT-I-OPRQST, Please mount volume MYTAPE in
device _MSA0:
New Tape.
```

（a） ユーザからのデバイスマウント要求

```
$
%OPCOM, 24-JAN-1983 15:32:12.56, request 14, from user
MORISAKI
Please mount volume MYTAPE in device _MSA0:
New Tape.
$
```

（b） オペレータコンソールのマルチボリューム要求

```
$
%OPCOM, 24-JAN-1983 15:41:10.23, request 15, from user
MORISAKI
MOUNT new relative volume 2 ( ) on MSA0:
$
$REPLAY/INITIALIZE_TAPE=15 "MYTAPE"
$
```

（c） マルチボリュームの応答

図-3 センタオペレータを含むデバイス・マウント手順[2]

一方，グローバルな目で見るとどうか．センタオペレータに複数ユーザからの要求が輻輳したときどうするか？ 一応の手段はあって要求をペンディングすることが可能となっている[2]．しかしオペレータがペンディングした要求を忘れてしまったらどうするか？ 意外に，ペンディング要求の一覧をディスプレイするという機能を，コマンドレベルで持っていないオペレーティング・システムがある．オペレータのコンソールは各種情報が次から次へと出力され，応答を要するメッセージはこのなかに埋れやすくなっている．人間的な観点からみると，コンソール出力のディスプレイ，ヘルプ機能に工夫の余地が残っているのではなかろうか？

（3） 障害時のオペレーション

オペレーティング・システムの IPL (initial program load) は現在ほとんど自動化されている．つまり電源の投入からパッケージのインストール，起動まで完全自動が可能である．それだけに障害時等でミニマム構成の IPL を行うことは難しくなっている．おおよそのオペレーティング・システムも IPL の後半で計算機サイト固有のカスタムコンフィギュレーション部分を含んでいるが，この部分をマニュアル操作で修正していく機能を持っていないものがある．周辺機器が不調で一部のインストールを取りやめる場合などシステム全体を強引に立ち上げた後，この部分を修正して改めて IPL をやりなおす操作が必要である．

障害等で縮退 IPL をやるというのは非常にまれな

ケースであることと障害という事態が競合しているため，オペレータ自身は精神的に動揺している状態が多い．こういう場合は IPL 手順を1つ1つ確認しながら自動 IPL を動かすというふうにはいかないのだろうか？ 1つ1つの手順（コマンド）に確認指定の修飾詞 (Qualifiers) を付けると日常時においても必要となってしまう．これをブート初期時にオプション指定を加えると可能となるようにするとよい．現実，ハードウェアモニタではブート操作時にオプション・パラメータの設定が可能なように設計されている[2]．つまり特定のレジスタに値を設定することが可能なのであるが，これはフィールドエンジニアやブートデバイスを変更するときに使用することになっている．このメカニズムを拡張して，通常のオペレーティング・システムの IPL 時にも内部に保有し，さらに立上げ後これの参照機能がそなわっていると，その用途は結構多い．システムマネージャが，この値を参照し値に応じてパッケージ等の切り替えを自動化する手続きを作成するのはたいしたことでない．従来はパッケージ等の切り替えを行うためカスタム部分でオペレータ入力を必要としていたものを初期ブートコマンドに移し，オペレータの操作方法を簡略化できるメリットがある．オペレータ自身は1オペレーションですむことと，次のコマンド投入待ちがなくなり別の作業に移りやすい．

IPL は障害時等を考えれば多くの手段がある方がよい．一方で人的作業による操作手順というものを長々と行うのは間違いの原因となる．

5. おわりに
—オペレーティング・システムのバランス感覚—

世の中のオペレーティング・システムのユーザインタフェースとして表に現われない機能・特質について述べ，最後の結びとしたい．

（1） システムガード

MULTICS 以来オペレーティング・システムのガードは多く研究されており，今日のものはこれら研究成果を多く取り入れている．これらシステムガードはユーザの悪意からシステムを守ると同時に人間のミスオペレーションからの防御も兼ねている．しかし，人の悪意は際限がなく，これを防ぐことに専念したオペレーティング・システムは使い心地という点で固苦しいものとなっていることが多い．

筆者の考えでは，オペレーティング・システムのガードとして人間の習慣的ミスオペレーションから守ることを中心にすえ，一方でミスオペレーションを少なくする方策を考え出す時期にきているのではないかと思う．空気のような抵抗もなく目にも見えないオペレーティング・システムは作り得ないだろうか．

（2） コンフィギュレーションの動的変更

システムチューンアップの期間は意外と長時間を要する．最適パラメータを見つけるため，少し時間をおいてはパラメータをいじるという作業を繰り返す．このとき多くのオペレーティング・システムは運用を中断する必要がある．システムジェネレーションをやりなおす時間と IPL の再実行を必要とするためである．ユーザから見るとたえず計算機が止められ「いったい，いつからまともに使えるのか．」と意味もわからず不満をシステムマネージャ，オペレータにぶつけてくる．

スタティックに変更するシステム・ジェネレーション・パラメータはオペレーティング・システム動作中に変更してもたいした影響を与えないと思われる．システムの動作中にカレント状態も含めパラメータを動的修正するユーティリティ[2]は，システムマネージャに福音をもたらす．システムマネージャが一番時間を取るのはユーザの訳のわからない苦情，不満を聞いているときである．

（3） ログ情報の重要性

システムの異常はハードウェア，ソフトウェアを含め重大な障害をもたらす前に必ず兆候が表われる．これはごくさ細なことで見落しやすい．しかし予知ということでは重要である．ではどうやってみつけるか？これは不断からログ情報をきちんと取っておくことである．

オペレーティング・システムは動いているハードシステムをモニタリングしつづける唯一のプログラムである．人が来て診断プログラムを短時間かけるより，はるかに多くの時間をかけた情報を取れる立場にある．ログ情報（特にハードログ情報）はシステムが占めるディスク容量を増化させるしオペレーティング・システムの内部を複雑にするため手を抜いたものが多いが，それはユーザが使い方を知らないため不用なものと判断し，メーカへのインパクトとならないことに起因する．人の健康と同じで予防処置と早期発見は計算機システムを長もちさせる秘訣である．障害による損失とディスクのちょっとした無駄とどちらが得であろうか．ユーザはオペレーティングシステムに対し見

せかけのファッション機能よりもっとこういう機能を求めるべきではなかろうか？ 流行はすたれるものである．

参 考 文 献

1) UCB : UNIX Programmer's Manual. 7th ed. Vertual VAX-11 Version, University of California Berkeley (1981), など．
2) DEC : VAX Software Handbook (1982) および VAX/VMS Software Document Set (1982)．
3) IBM : VSE/Advanced Functions System Management Guide (1982), など．
4) 益井清紀 : DIPS-1 DIPS-103-10 のジョブ制御について，情報処理，Vol. 15, No. 10 (1974)．
5) Denning, P. J. : The Working Set Model for Program Behavior, ACM Symposium on Operating System Principles, Gotlinberg, Tenn., Vol. 11, No. 5, pp. 323-333 (1967)．

(昭和58年2月1日受付)