オペレーティング・システム 33－6
（1986. 12. 13）

ＬＩＳＰ処理専用ワークステーションのオペレーティングシステムについて

白川洋充

立命館大学理工学部

本報告はオペレーティングシステム、３次元グラフィックス、グラフィカルＬＩＳＰ等の研究のツールとなる研究用ワークステーションのオペレーティングシステムについて述べるものである。　このオペレーティングシステムは、オペレーティングシステムが開放的であり、オペレーティングシステムとプログラミング言語との間に明確な境界を設けない特徴を有する。さらに、システムの融通性、ストリーム指向の入出力操作、メッセージ交換の機構をプロセス間の通信と同期に使用している。　また、オペレーティングシステムを含めプログラミング言語、エディタのすべてをＲＯＭ化するため、ソフトウェアのモジュール化を行った。

OPERATING SYSTEM OF THE DEDICATED LISP-BASED WORKSTATION

Hiromitsu SHIRAKAWA

Faculty of Science and Engineering, Ritsumeikan University
Kita-Ku, Kyoto, 603, Japan

Ryouanji is a single-language(LISP) workstation which is used as a tool for the studies of operating system, 3D computer graphics and Graphical LISP.

LISP and Editor are subset of ETALisp and EZ, respectively, both developed at Electrotechnical Laboratory.

The main features of the operating system are openness, flexibility, stream oriented input and output, message passing mechanism and modularization.

Message passing mechanism has been used as the basis for communication and synchronization between processes.

In this paper we describe the attributes of processes, message passing mechanism and stream oriented input and output.

1．はじめに

ＬＩＳＰ処理専用ワークステーション「龍安寺システム」は、オペレーティングシステム、３次元グラフィックス、グラフィカルＬＩＳＰ等の研究を行うためのツールとなるように開発されたものである。

研究用の本ワークステーションの設計思想は、(1) ハードウェアの融通性、(2) オペレーティングシステムの開放性、(3) ソフトウェアの観測可能性である。　ハードウェアの融通性としては、機能的な融通性と実現上の融通性を重視した。　ここで、機能的な融通性とはワークステーションの機能を変更、あるいは、付加できる能力を有することであり、実現上の融通性とは機能の変更をすることなく、性能の改善を図ることが可能なことを意味する。[1]　オペレーティングシステムの開放性とは、オペレーティングシステムとプログラミング言語やユーザのプログラムとの間に明確な境界を設けないことである。[2] この考え方は、ハードウェアとソフトウェアに深く関係する研究を行うために必須のものであるが、商用のワークステーションで期待することは殆ど不可能に近いといえる。　ソフトウェアの観測可能性とはプログラムの挙動を測定することによってシステム評価を可能にすることであり、例外処理やデバッグ機能をより高度化することによって、より良いプログラミング環境作りに役立たせることができる。

本報告の目的は、ワークステーションのソフトウェア、特にオペレーティングシステムの構成方法について詳細に述べることにする。

2．ソフトウェアシステムの概要

ＬＩＳＰ処理専用ワークステーションとは、ＬＩＳＰ　を可能な限り効率良く稼動させるとともに、ユーザに対しては、よりインタラクティブにＬＩＳＰベースのソフトウェア開発を行うことができるような環境を提供するものである。　ＬＩＳＰは近年、人工知能向きのプログラミング言語として脚光を浴びているが、２７年の歴史を持ち、種々の進化を遂げてきた。[3][4]　本システムに採用したＬＡＴＥＲＡＬ　というＬＩＳＰ　は、電子技術総合研究所で開発されたＥＴＡＬｉｓｐ[3]（Lisp ExTension for Automation）のサブセットである。

この　ＬＩＳＰ　はロボットの環境モデルを実現したり、ロボットの動作記述を容易にしたり、各種のリアルタイム処理を行う等の機能を有する。

ＥＴＡＬｉｓｐ　は、我々の研究用には向いているが、Ｃｏｍｍｏｎ　Ｌｉｓｐ　と根本的な部分において大きく異なっていた。　そのため、ＬＡＴＥＲＡＬ　では　Ｃｏｍｍｏｎ　Ｌｉｓｐ　の仕様に合わせるとともに種々の改良を行っている。ＬＡＴＥＲＡＬ　はテキストのみならず、ヒープエリア、セルエリア等が、他のプロセスと共有されることを可能にして、コンカレントＬＩＳＰ、コンカレントガーベジコレクション、コンカレントエディタを実現できるようになってる。　ＬＩＳＰ　のプログラム開発を支援するエディタ　ＥＺ[5]が存在する。　ＥＺ　は、ＥＴＡＬｉｓｐ　とともに開発されたもので、ＥＴＡＬｉｓｐ　とコンカレントに動作するとともに、プロセス間通信を用いてＬＩＳＰプログラムの編集を迅速に行うことができるようなものである。

3．オペレーティングシステムの構成

本ワークステーションに存在する唯一のプログラミング言語　ＬＩＳＰ　とエディタをコンカレントに動作させるとともに、マルチウインドウ機能、プロセス間通信等を実現するものがここで述べるオペレーティングシステムである。

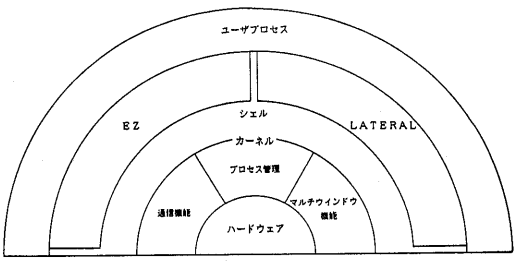

図１　ソフトウェアの構成

図１にソフトウェアシステムの構成を示す。

ワークステーション上のすべてのプログラムはメモリ常駐であり、オブジェクトがメモリ上に展開されている。　さらにオペレーティングシステムを含めプログラミング言語、エディタのすべてをＲＯＭ化することを考えているので、オペレーティングシステムの構成は次のような配慮を行った。

(1)　シングルユーザ
(2)　マルチタスク
(3)　ディスクレス
(4)　すべてのプロセスのテキスト部分の共有
(5)　モジュール化

したがって、通常のオペレーティングシステムに見られるようなメモリ管理とファイル管理は存在しない。　その代わり、ハードウェアの方で　ＬＩＳＰ、エディタ、オペレーティングシステムそれぞれのメモリプロテクションを実現している。　ファイル管理に関しては、ホストの　ＵＮＩＸ　マシンをファイルサーバとすることにより実現している。

３．１　プロセスの管理

プロセスを管理するために必要な情報は、プロセスディスクリプタと呼ばれるデータ構造体に記録されている。　プロセスディスクリプタは、プロセスを切り換える時、次にプロセスの実行を再開させるためのコンテクストと、そのプロセスに割り当てられているメモリ空間、プロセスがオープンしているストリームなど、プロセスの実行環境を記録する要素から成る。　図２にプロセスディスクリプタの構造を示す。

```
typedef struct prcdesc {
    long            reg[P_REGS];
    unsigned short  p_id;
    struct prcdesc  *qlink;
    unsigned char   p_pri,
                    p_stat;
    HEADER          free;
    SEGMENT         text,
                    ...
                    stack;
    PSTRM           strm[P_STRM];
    MSG_Q           m_queue;
                    ...
} PD;
```

図２　プロセスディスクリプタの構造

プロセスの状態には、実行状態、実行可能状態、封鎖状態、サスペンド状態がある。

実行可能状態のプロセスはレディーキューを用いて管理される。　レディーキューとはプロセスディスクリプタをリンクしたもので、データ構造は図３に示すように、最初の要素を指し示すポインタ変数 head　と、　最後の要素を指し示すポインタ変数 tail　を持つ。　プロセスディスクリプタ内には、次の要素を指し示すポインタ変数 qlink　がある。

```
typedef struct  {
            PD    *head,
                  *tail;
} QUEUE;
```

図３　レディーキューの構造

プロセス間の通信は、メッセージ交換方式で行う。[6][7]　メッセージ交換のプリミティブには同期型のものと、非同期型のものを用意する。　メッセージ用のキューは、各プロセスに１つ用意する。　また、メッセージのキュー操作はスケジューラが行う。

メッセージの送信は Send 、または Notify で行い、受信は Recv で行う。

メッセージはキューに無限に付けられるわけではなく、ある上限を越えればメッセージの送り手は封鎖されるというフロー制御を行っている。

メッセージはヘッダとノードから成る。　図４にメッセージの構造を示す。

```
typedef struct m_head  {
        struct m_head *next;
        PD            *pd;
        int           type,
                      mode;
        MSGNODE       *link;
} MSGHEAD;

typedef struct m_node  {
        MSG           msg;
        strcut m_node *link;
} MSGNODE;
```

図４　メッセージの構造

図５にメッセージのヘッダとノードの様子を示し、プロセスにメッセージがキューイングされた状態を図６に示す。

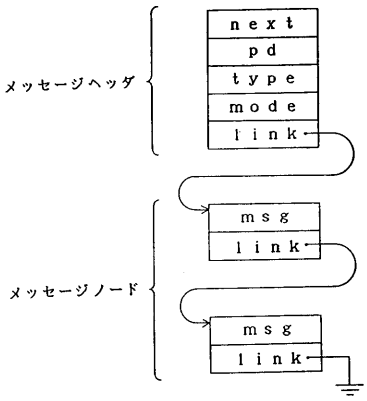

図５　メッセージ

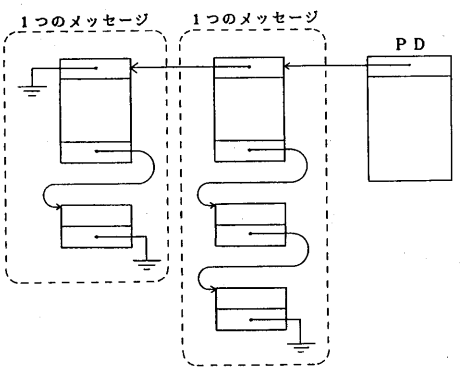

図６　メッセージとプロセスの関係

ここで、メッセージヘッダの next はメッセージをリンクするためのポインタである。　pd　は、メッセージを送ったプロセスを示す。

プロセス間の通信を同期型のメッセージ交換のプリミティブ Send で行う場合は、この pd で示されるプロセスへメッセージ受信の確認のため Reply を送る。　type　は送信されたメッセージの種類を示す。　mode　はメッセージ交換のプリミティブが同期型　Send　か非同期型 Notify で行われたかを示すものである。　link　は、メッセージノードへのポインタである。

次に、これらの関数について述べる。

Send(pd, msg)

メッセージ msg　を　pd で示されるプロセスへ送信する。　Send　には、間接的にプロセスと名前との関連づけを行う機能はない。　Send　は同期型のメッセージの送信であり、送り手は受け手からの Reply があるまで封鎖される。

pd　で示されるプロセスが Recv による封鎖状態であれば、Send　によって実行可能状態となる。

Notify(pd, msg)

Send　と同様に pd で示されるプロセスへメッセージを送信する。　しかし、Notify　は非同期型のメッセージ通信のプリミティブであり、送り手はメッセージ送信後も、封鎖されることなく次の処理へ移ることができる。　pd　で示されるプロセスが Recv　による封鎖状態であれば、Send　の場合と同様に Notify で実行可能状態となる。

Recv(my_msg)

自分に当てられたメッセージを受信する。
my_msg　で指定される場所にメッセージヘッダをコピーする。　メッセージのキューにメッセージが無ければ、メッセージが到着するまで封鎖される。
受信したメッセージが同期型メッセージ交換のプリミティブ Send で送られたものであれば、Reply を行う必要がある。

非同期型 Notify でメッセージが送られたのであれば Reply　を行う必要はない。

Reply(pd, val)

pd　で示されるプロセスに、メッセージ到着の確認を行う。　Reply によって、pd　で示されるプロセスは封鎖を解かれる。　val は pd で示されるプロセスへの返り値である。

プロセス間の名前付け機能を持った、高水準のメッセージ通信としてポートを用意する。　ポートはユーザプロセス間でメッセージの送信、受信をするランデブーポイントであり Send, Notify, Recv と同様の機能を実現する。

ポートのメッセージリンク等の情報はポートディスクリプタが保持する。　未使用のポートディスクリプタはフリーリストで管理する。　メッセージヘッダとメッセージノードは、Send, Recv　で用いるものと同一のものを用いる。

次に、これらの関数について述べる。

ptcreate(name, maxmsgs)

ユーザプロセス間で通信が必要な時にポートを生成する。 最大 maxmsgs 分だけのメッセージを保持でき、name でユニークに指定できるポートを生成する。

ptsend(name, msg, flag)

name で指定されるポートに msg を送信する。flagは、NoDELAY 等を指定する。

ptrecv(name, msg, flag)

name で指定されるポートからメッセージを受信する。

ポートはマルチライト、マルチリードでありメッセージの送り手、受け手、メッセージの種類に関係なく、メッセージの送受信を行うことができる。
ポートではメッセージを到着時間順に保持する。
時間的に先に ptsend で送られたメッセージから順に、受信要求 ptrecv を行ったプロセスに渡される。

３．２ 入出力ストリームの操作

本オペレーティングシステムでは、入出力を行うデバイスに対する操作は、ユーザ側からはすべて統一的な手続き操作によって行うことを可能としている。 すなわち、異なるタイプのデバイス（例えば、マウス，ウインドウ等）に対して、ユーザは同じ関数を用いて入出力を行う。 これを実現するために、プロセスとデバイスとの間に「ストリーム」と呼ばれる入出力の通路を用意する。

ＵＮＩＸ では、入出力のデバイスは特殊ファイルとして取り扱うため、デバイスに対する入出力についてもファイルへの出力、ファイルからの入力という形で統一的に行われる。[8] 最新の ＵＮＩＸ ではストリームの概念が導入されている。[9] また、ＯＳ６ では入出力の情報の変換のための手段としてストリームを定義づけている。[10]

本システムのストリームは ＯＳ６ のそれに近いものであるが、プロセスとデバイスをつなぐ入出力の通路と考える点が違っている。 これにより入出力の操作においてプロセスはデバイスの種類を意識する必要はなく、ストリームだけを意識すればよい。 ストリーム操作はプロセスが入出力を行う唯一の方法である。

入出力操作では、プロセスが入出力の対象となるデバイスに対してストリームを open で開き、次に read または write で実際のデータの転送を行い、close でストリームを閉じて、入出力を終了する。

ストリームディスクリプタは、入出力に関する情報を保持する。 ストリームディスクリプタはフリーリストで管理し、open を行うたびにフリーリストから取り出して使用する。

デバイスディスクリプタはデバイス固有の情報を保持する。 このデバイスディスクリプタは、デバイスセレクトテーブルへのポインタを保持している。このテーブルは、デバイス固有の関数（そのデバイスの read , write 等）が登録されているテーブルである。

デバイスハンドラは、仮想デバイスを含むデバイス固有のものであり、デバイスへの要求の処理を行うプロセスで、デバイスディスクリプタごとに存在する。

open は入出力要求の対象となるデバイスのデバイスディスクリプタを直接指定して、プロセスとデバイスとの間にストリームを作る。 プロセスからストリームを open すると、ストリームディスクリプタをプロセスディスクリプタに登録するとともに、ストリームディスクリプタにはデバイスディスクリプタへのポインタ情報を保持させる。 すなわち、ストリームを経由して、プロセスディスクリプタからデバイスディスクリプタへのリンクを作る。

open はプロセスディスクリプタ内のストリームを登録した配列のインデックスであるプロセス固有のストリームディスクリプタ番号を返す。 read, write, close 等はすべて、open の返り値であるストリームディスクリプタ番号を用いてストリームを操作することによって行う。 １つのプロセスが同時に使用できるストリームの個数はプロセスディスクリプタ内に登録できる個数によって制限される。

また、１つのストリームから指し示されるデバイスは１つであるが、１つのデバイスを指し示しているストリームは複数個存在し得る。

ウインドウはそれぞれを仮想端末と考える。 出力はスクリーンに、入力はウインドウに対応した仮想的なキーボードから行われると考える。 ウインドウに対する open は既に存在するウインドウのみに行われる。

ウインドウは、複数のプロセスがストリームを通して、１つのウインドウディスクリプタを指すことによって共有される。 またウインドウに対する open を複数行うことによって、いくつかのウインドウを１つのプロセスが使用することも可能である。

read によってストリームからの入力を行う。 ストリームディスクリプタ番号を指定することによって、入力の要求を出すべきデバイスはユニークに定まる。 このストリームディスクリプタ番号は、open の返り値を指定する。

read は、ストリームで示されるデバイスハンドラに対して、入力要求のメッセージを送る。 デバイスハンドラは、それによって要求の処理を開始する。 デバイスハンドラは、そのメッセージによってデバイスセレクトテーブル内の入力ルーチンを呼び出し、指定されたアドレスに指定個数分だけのキャラクタを書き込む。

ウインドウからの入力は、カレントになっているウインドウからしか許可しない。 ウインドウごとにキーボードデバイスハンドラが存在するが、ある時点で動作しているのは、ただ１つのカレントのウインドウに対応するキーボードデバイスハンドラである。 その切り換えは、カレントのウインドウの変更時に行われる。 そのとき、処理中のキーボードデバイスハンドラは、そのまま封鎖され、新たにカレントになったウインドウのキーボードデバイスハンドラが封鎖を解かれて処理を開始する。 ウインドウからの入力要求のメッセージは、キーボードデバイスディスクリプタには必ず付けられるが、カレントでないウインドウに対して要求を出した場合は、その処理はウインドウがカレントになるまで待たされる。

３．３ ウインドウの管理

ウインドウマネージャは、仮想端末である各ウインドウへの出力を統合するプロセスである。 各プロセスがウインドウに出力する時は、すべてウインドウマネージャに対して処理を要求する。 処理要求は、キューによって管理され、逐次処理される。

ウインドウマネージャはウインドウごとにオフスクリーンにビットマップを持っている。 実画面、すなわち、フレームバッファは、これらのビットマップの写像である。 文字出力の要求があれば、端末のエミュレートを行い、結果をビットマップ上に反映させる。 また、グラフィック出力の要求があれば、必要な計算の後、ビットマップ上に反映させる。 そして、実画面の更新要求により、フレームバッファに写像する。

マウスカーソル及びカレントウインドウのカーソルの表示も、マウスデバイスハンドラ及びキーボードデバイスハンドラからの要求により、ウインドウマネージャが処理を行う。

シェルは、ユーザとカーネルのインターフェースとなるものである。 それぞれのウインドウには、シェルが存在する。

ウインドウ生成関数によって新しいウインドウが作られると、新たに生成されたウインドウを標準ウインドウとするシェルが起動される。

シェル上から起動されたプロセスはその環境を引き継ぐ。 起動されるプロセスのためにプロセスディスクリプタとシェルのストリームディスクリプタと同じ内容のストリームディスクリプタを新たに作る。

シェルは、新たにプロセスが起動されると、起動されたプロセスが終了するまで封鎖される。

シェル上で起動されたプロセスが終了すれば、そのプロセスのストリームディスクリプタ等はフリーリストに返され、封鎖されていたシェルが再び実行される。

シェルの終了、もしくはウインドウ削除により、ウインドウは実画面上から消去される。

４． おわりに

研究用のワークステーションのオペレーティングシステムの構成法について述べた。

ワークステーションの唯一のプログラミング言語である ＬＩＳＰ とエディタは、研究の増殖的な発展を支援するよう独自の進化をしている。 また、開放的なオペレーティングシステムは、従来のシステムコールだけを使ったものに較べ、よりコンカレ

ントで、より適応性を有するソフトウェア開発を可能にしている。

最後に、本システムの各パートはすべて完成し評価の段階であるが、システムとしての統合はまだなされていない。 システム全体の評価は次の機会に発表する予定である。

謝辞

本システムの開発は、６ヶ月におよぶオペレーティングシステム研究グループのミーティングの中から生まれた。 システムのインプリメントは同グループの上坂 靖氏、谷口健一氏、笹川元彦氏、田中正樹氏、丹波 覚氏、中嶌浩之氏によってなされた。ＬＩＳＰ は澳 直氏、中野圭祐氏によってインプリメントされた。 ハードウェア開発とその他種々の支援は白川研究室の４回生の諸君に負うところが多い。

ここに謝意を表す次第である。

参考文献

(1) M. Sloman, J. Magee, J. Kramer, "Building flexible distributed computing systems in CONIC", in Distributed Computing Systems Programme, D. A. Duce ed., Peter Peregrinus Ltd., (1984)

(2) B. W. Lampson, R. F. Sproull, "An open operating system for a single-user machine", Proc. of the Seventh Symposium on Operating System Principles, (1979)

(3) 小笠原，松井：「ロボットプログラミング機能を持つＬｉｓｐ処理系の開発」昭59，日本ロボット学会学術講演会

(4) 白川，熊谷，古川：「グラフィカルＬＩＳＰの開発について」，立命館大学理工学研究所紀要，第43号，pp.235-243，(1984)

(5) 松井：「環境を保持するディスプレイエディタ：ＥＺ」昭59，情報処理学会前期全国大会

(6) W. M. Gentleman, "Message passing between sequential processes: the reply primitive and the administrator concept", Software-Practice and Experience, Vol.11, pp.435-466,(1981)

(7) W. Zwaenepoel, "Message passing on a local network", Stanford University, STAN-CS-85-1083 (1985)

(8) K. Thompson, "UNIX time-sharing system: UNIX implementation", B.S.T.J., Vol.57, No.6, pp.1931-1946, (1978)

(9) D. M. Ritchie, "A stream input-output system", AT&T Bell Laboratories Technical Journal, Vol.63, No.8, pp.1897-1910, (1984)

(10) J. E. Stoy, C. Strachey, "OS6 - an experimental operating system for a small computer", Computer Journal, Vol.15, No.2 and No.3, (1972)