オペレーティング・システム　39－3
（１９８８．６．１７）

ＬＩＳＰ処理専用パーソナルメディア*ＳＬＩＭ*のオペレーティングシステムについて

白川　洋充　　谷口　健一

立命館大学　理工学部

本報告では、オペレーティングシステムの研究のツールとなるＬＩＳＰ処理専用のパーソナルメディア*ＳＬＩＭ*（Ｓｍａｌｌ　Ｌｉｓｐ　ｂａｓｅｄ　Ｍａｃｈｉｎｅ）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、開放的であり、オペレーティングシステム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という特徴を有する。　さらに、システムの柔軟性、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通信と同期に使用している。　また、構造的なシステムとするために、オペレーティングシステムを含めプログラミング言語、エディタのすべてのソフトウェアをモジュール化した。

OPERATING SYSTEM OF THE LISP BASED PERSONAL MEDIA *SLIM*

Hiromitsu SHIRAKAWA and Ken-ichi TANIGUCHI

Faculty of Science and Engineering, Ritsumeikan University

Kitaku, Kyoto, 603 Japan

SLIM (Small Lisp based Machine) is a single-language (LISP) personal media which is used as a tool for the study of operating system.

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 syncronization 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

パーソナルメディアは、A. Kay と A. Goldberg が、1976 年に発表した論文[1]に初めて登場した言葉である。パーソナルメディアとは知識を伝え、知識を蓄積でき、どのような人にも自由に扱えるものである。 A. Kay らがその当時提案したノートブックのような薄型のダイナブックは、実際に開発される技術的背景はなかった。現在の技術で薄型のパーソナルメディアが実現できるかどうかが、***ＳＬＩＭ***の開発の動機である。

近年のパーソナルコンピュータ、ワークステーションの普及は、このパーソナルメディアの理想に近づいたといえよう。 しかし、個人のコンピュータに対する知的あるいは技術的な要求を充分に満足させていない。 その理由は、コンピュータのハードウェア、ソフトウェアがオープンでないこと、特にオペレーティングシステムとプログラミング言語の間には厳然たる壁があり、そこにはシステムファンクションコールという隘路が存在するからである。 そのため、その壁を取り去る必要がでてくる。 そこで、本オペレーティングシステムは、ハードウェア、ソフトウェアともにユーザに解放的であるという基本方針のもとで開発された。

本報告では、***ＳＬＩＭ***のハードウェア構成について簡単に触れてから、そのオペレーティングシステムの詳細について述べる。

２．ハードウェア構成

ＳＬＩＭのハードウェアは、パーソナルメディアという言葉に示されるようにコンパクトに設計されている。

ＬＩＳＰの特殊性を考慮して、ＬＩＳＰを効率良く動作させるために、8 Mbytes の大容量メモリを搭載している。 基板の大きさはプラズマディスプレイコントローラの基板と同じ 200 × 300 mm の大きさである。***ＳＬＩＭ***本体とプラズマディスプレイとコントローラを合わせると 60 mm の厚さで実現できた。 ハードウェアのブロック図を図１に示す。 メインのＣＰＵは、モトローラ社の 32 bits ＣＰＵ MC68020 を用い、メモリはＤＲＡＭ 8 Mbytes、 ＥＰＲＯＭ 512 Kbytes を実装している。 入出力処理専用のサブＣＰＵは MC68000 を用いており、シリアルおよびパラレルのＩ／Ｏインタフェース、ディスプレイ表示回路、時計回路を持つ。シリアルＩ／Ｏは、 RS232C 規格でホストコンピュータとの通信に用いている。 パラレルＩ／Ｏは、汎用の 8 bits パラレルポートとして設計されているが現在はプリンタを接続している。

メインＣＰＵとサブＣＰＵとの通信は、双方に接続されたデュアルポートメモリを用いて行っている。 デュアルポートメモリは、 4 Kbytes の大きさでロケーションモニタ機能を有する。

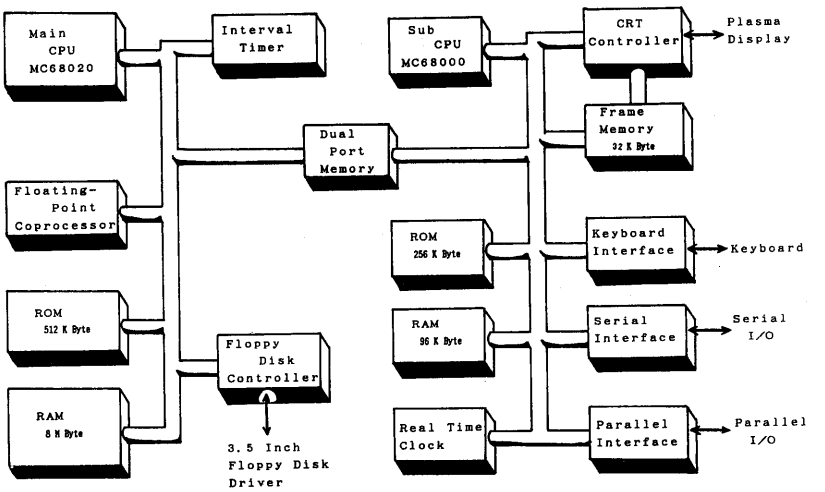

図１　ハードウェアの構成図

3． オペレーティングシステム

SLIMのオペレーティングシステムは、シングルユーザ、マルチプロセスのオペレーティングシステムである。 ソフトウェア構成図を図２に示す。

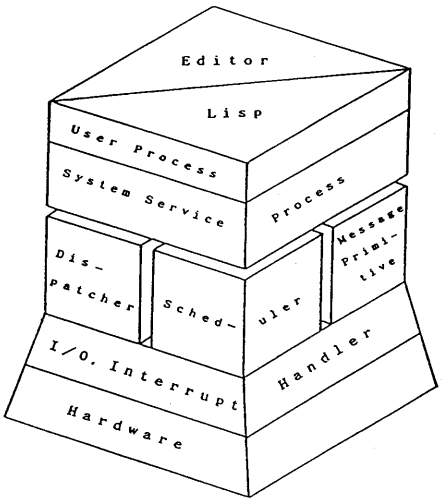

図１ ソフトウェア構成図

3．1 スケジューリング方式

SLIMのスケジューリングは、多重レディーキューを持ったラウンドロビン方式である。 レディーキューは優先順位ごとに存在し、プロセスは該当する優先順位のキューにキューイングされる。 また、レディーキュー間では、ある優先度のレディーキューはそれよりも優先度の低いレディーキューよりも相対的に高い優先順位を持つ。 すなわち、各レディーキューは、それよりも高い優先順位のレディーキューがすべて空の時しかスケジューリングされない。 また、実行中のプロセスより高い優先度のプロセスがレディーキューに入ってくると、実行中のプロセスは高い優先度のプロセスに実行権を横取りされる。 プロセスの優先順位はその生成の時に設定し、常に一定である。 システムのスループットを上げるために、割込み関係のプロセス、また、重要なハンドラプロセスに高い優先順位を割り当てている。 優先順位は、表１のように設定している。

3．2 プロセス間通信

SLIMでは、プロセス間の通信・同期はすべて、メッセージパッシング方式[2]を用いて行っている。 メッセージは可変長のデータを送ることができる構成とした。

priority	process
5	floppy device handler
4	process handler stream handler signal handler communication process
3	interrupt process
2	arbiter process
1	other handlers user process
0	idle process

表１ プロセスの優先度

メッセージプリミティブとしては、同期型の送信・受信、非同期型の送信、また、同期型の送信に対する応答のプリミティブを用意した。 また、メッセージの送り手・受け手の指定に関しては、どのプロセスからも送ることができ、送り手のみを指定する多対一の方式を採っている。

メッセージは、メッセージヘッダとメッセージノードから構成される。 メッセージヘッダは、メッセージに関する各種情報を格納する。 メッセージノードは、送るデータそのものを格納する。 4 bytes 単位のノードを必要な数だけリンクすることにより可変長のメッセージを扱うことができる。 メッセージノードは、送り手がシステムから確保し、受け手の責任でシステムに解放する。

プロセスがメッセージを受けるためのキューは、プロセスディスクリプタに存在する。 メッセージキューへのキューイングは、カーネルが行う。

以下に、それぞれのメッセージプリミティブの詳細について示す。

```
Send(pd, mhp)   /* 同期型送信 */
PD *pd;   /* プロセスディスクリプタへのポインタ */
MSGHEAD *mhp;   /* メッセージヘッダへのポインタ */
```

pd で示されるプロセスに mhp で示されるメッセージを送る。 Send() を実行したプロセスは、このプロセスに対する Reply() が行われるまで封鎖される。 Send() の返り値は Reply() によって決められる。 エラーが生じた時は－１が返される。

```
Notify(pd, mhp) /* 非同期型送信 */
PD *pd;   /* プロセスディスクリプタへのポインタ */
MSGHEAD *mhp;   /* メッセージヘッダへのポインタ */
```

pd で示されるプロセスに mhp で示されるメッセージを送る。 受信プロセスのフロー制御以外には封鎖されない。 返り値は送信成功の時１、エラーが生じた時は－１である。

```
PD *Recv(mhp)   /* 同期型受信 */
MSGHEAD *mhp;   /* メッセージヘッダへのポインタ */
```

このプロセスに送られてきたメッセージを mhp で示される受信プロセスのメッセージヘッダで受ける。 プロセスに到着しているメッセージがない場合、メッセージの到着までそのプロセスは封鎖される。 Recv() は Send()、Notify() を区別せずにメッセージを受信する。

それらの違いは返り値が Send()の場合には送信プロセスのpd、Notify() の場合には０であることである。

```
Reply(pd, val)  /* 応　答 */
PD *pd;   /* プロセスディスクリプタへのポインタ */
long val;       /* Send() への返り値 */
```

pd で示されるプロセスに応答を返し、Send() によって封鎖状態にあるプロセスを実行可能状態にする。 その際、 val を Send() の返り値とすることができる。返り値は成功すれば１、エラーの場合は０または－１である。

３．３　システムサービスプロセス

ＳＬＩＭにおけるシステムサービスプロセスとは、オペレーティングシステムの諸機能を機能毎にモジュール化してプロセスとして実現したものである。 システムサービスプロセスへの要求は、すべてメッセージパッシングによって行われる。 システムサービスプロセスへの要求の多くは、誤動作を防ぐ意味で直接メッセージを使うのではなくてライブラリとして提供している。

システムサービスプロセスの中でハンドラと呼んでいるものは、１つの計算機資源を管理するサーバプロセスとして機能する。 １つの計算機資源に対して１つの管理者となるプロセスを用意することにより、相互排除の必要性を最小に抑え、かつ、その資源を安全に運用することができる。 ハンドラは要求を先着順サービス（ＦＣＦＳ）方式でサービスする。 この方式はメッセージキューの管理以外にスケジューリングのための処理を行う必要がない。 このように、ある計算機資源に対する操作は、そのハンドラ以外は行わないので、メッセージキュー操作以外の相互排除は必要なく、システムのコンカレント性を高める。

現在の***ＳＬＩＭ***のシステムサービスプロセスをまとめると表２のようになる。

種類	処理内容
プロセスハンドラ	プロセスの生成、消滅、状態報告、検索などのプロセスディスクリプタの管理とメモリの割当、解放
ストリームハンドラ	ストリームの生成、削除、コピーなどストリームディスクリプタの管理
デバイスハンドラ	入出力の処理、制御など入出力デバイスまたはファイルを管理
インタラプトプロセス	入出力の割込みによるデータ転送とその通知の処理
アービタプロセス	入出力におけるインタラプトプロセスまたは入力割込みとデバイスハンドラとの調停
サブＣＰＵコミュニケーションプロセス	サブＣＰＵとの通信を必要とする要求の処理
シグナルハンドラ	割込み・例外に対する処理の設定とその発生時の処理

表２　システムサービスプロセス

３．４　シグナル処理

端末インタラプト、アラーム処理、浮動小数点例外、また、メインＣＰＵ(MC68020) のバスエラーや不当命令などの例外を処理するためにシグナルの機能を実現した。 本オペレーティングシステムでは、シグナル発生時に行う処理をシグナル設定を行ったプロセスとは別のプロセスとして扱う。 その理由は、シグナルを設定したプロセスがシグナル処理から復帰する際シグナル発生時の状態に戻らなければならないこと、および、シグナル処理プロセスがプロセス間通信を容易にできるようにするためである。

シグナルの管理のために、シグナルハンドラプロセスを設けた。 シグナルハンドラは、シグナルの設定、シグナル発生要因の解析、シグナル処理プロセスのレディーキューへのキューイングを行う。

シグナル設定時に、シグナルハンドラはシグナルディスクリプタを確保する。 シグナルディスクリプタは、シグナルを設定したプロセスとシグナル処理プロセスとのそれぞれへのプロセスディスクリプタへのポインタ、シグナルステータス、シグナルの発生要因などからなるデータ構造である。 図３にシグナルディスクリプタを示す。 次に、シグナルハンドラは、シグナル処理プロセスのプロセスディスクリプタとシグナルディスクリプタを互いに関連付ける。

```
typedef struct sigdes {
  struct sigdes *next;    /* Link of next struct */
  unsigned char stat;     /* Signal status mode  */
  int           *sigcall, /* Signal caller pd    */
                *sigp;    /* Signal function pd  */
  struct sigdes *retsigd; /* Return signal
                                    function pd  */
  int           sig,      /* Cause of signal     */
               (*func)(); /* Signal function     */
} SIGD;
```

図３　シグナルディスクリプタ

シグナル設定プロセス、シグナル処理プロセスの関係を図４に示す。

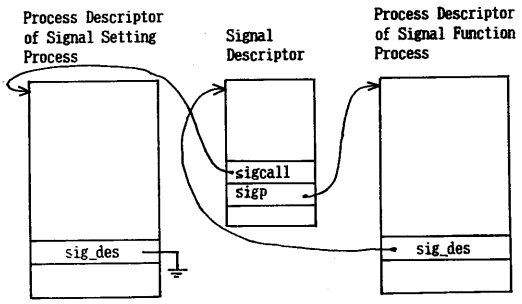

図４　シグナル設定プロセスと
シグナル処理プロセスの関係

シグナルが発生した時の処理は以下の通りである。

(1)　シグナルの発生要因を解析する。

(2)　その発生要因に対応するシグナル処理プロセスをレディーキューにキューイングする。

(3)　シグナル設定を行ったプロセスを封鎖状態にする。

シグナルプロセスが終了したときは、シグナル設定プロセスを実行可能状態に戻す。　ただし、バスエラーや不当命令などの例外の時は、復帰不可能とみなしてシグナル設定プロセスを終了させる。

３．５　入出力とファイル

SLIMのオペレーティングシステムの入出力処理の特徴としては、

(1)　OS6[3] で提案されたストリームの概念を採用していること

(2)　アービタプロセスを用いて、入出力要求を行うプロセスと割込みによって起動されるプロセスと同期を取っていること

(3)　入出力専用ＣＰＵを用いていること

が挙げられる。

また、ファイルの処理に関しては、

(1)　UNIX マシンをファイルサーバとしていること

(2)　外部ファイルとして MS-DOS フォーマットの3.5”フロッピディスクをサポートしていること

(3)　共有メモリを用いることにより、高速なファイル処理、および、エディターＬＩＳＰ間のテキストファイルの転送ができること

が特徴として挙げられる。

３．５．１　ストリーム

プロセスの入出力は、ストリームを用いて統一的に行う。　ストリームの特徴として、次のことが挙げられる。

(1)　デバイスの種類に依存せず統一的な入出力操作ができること

(2)　デバイスと同等にファイルが扱えること

(3)　プロセス間でのデバイスの共有ができること

ストリームは、プロセスに入出力のための統一的なインターフェースを提供し、異なる種類のデバイスに対して入出力のプロトコルの相違を吸収している。　したがって、プロセスはストリームを入出力の対象であるデバイスの特性を意識することなく入出力処理ができる。また、ファイルサーバ、共有メモリ、そして、外部ファイルシステムに存在するファイルも通常のデバイスと同等に扱うことができる。

ＵＮＩＸでも、ストリーム[4]という概念が導入されているが、ＵＮＩＸにおけるストリームは、ユーザプロセスとデバイスとの間の全二重結合のことで、いくつかの線形につながれたモジュールからなる。　このオペレーティングシステムのストリームは、内部にモジュールを持たず、処理はデバイス側のストリームの出口のプロセスのみが行うことが異なっている。　また、デバイスとファイルを統一的に扱う点でＵＮＩＸファイルシステムと本オペレーティングシステムのストリームは、似た概念となっている。

入出力操作は以下に示すように、open, close, read, write の命令を用いることにより行う。

open(pathname, o_flag, mode)

pathname : デバイスまたはファイルの指定

o_flag : 入出力属性

mode : パーミッション

返り値は、成功の場合はストリーム番号、
失敗の場合は－１である。

close(st)

st : ストリーム番号

返り値は、成功の場合は０、
失敗の場合は－１である。

write(st, buffer, nbyte)

st : ストリーム番号

buffer : ユーザ側の入出力バッファのポインタ

nbytes : 出力文字数

返り値は、成功の場合は実際に書き出した文字数、
失敗の場合は－１である。

read(st, buffer, nbyte)

st : ストリーム番号

buffer : ユーザ側の入出力バッファのポインタ

nbytes : 入力文字数

返り値は、成功の場合は実際に読み込んだ文字数、
失敗の場合は－１である。

ここで、ファイルをオープンするときに指定するパスネーム(pathname)は、通常のデバイスの場合とファイルを扱うデバイスの場合において、それぞれ表３.a,b のように指定する。

pathname	device name
d:tty0	console 0
d:tty1	console 1
d:tty2	console 2
d:aux	serial I/O
d:pia	parallel I/O

表 ３.a 通常のデバイスのパスネーム

pathname	device name
u:filename	UNIX file server
r:filename	shared memory
f:filename	floppy disk

表 ３.b ファイルを扱うデバイスのパスネーム

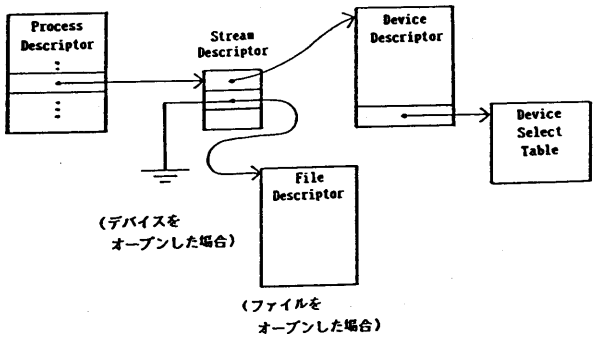

図５ 入出力に関係するディスクリプタ

入出力に関係するディスクリプタを図５に示す。 デバイスディスクリプタは各デバイスの情報を保存するデータ構造である。 ストリームディスクリプタはデバイスあるいはファイルとユーザプロセスを関連付けるためのデータ構造で、デバイスディスクリプタ、ファイルディスクリプタへのポインタ、入出力属性などなどの情報から構成される。 ファイルディスクリプタは、ファイルが共有メモリまたはファイルサーバ上のものであれば共有メモリ上での情報を、外部ファイルシステムであればそのファイルシステム上での情報を保持する。 図６にストリームディスクリプタの構造を示す。next、stat はシステム内での管理用に用いる。 o_flag はストリームの入出力モードを示す。 type はデバイスのタイプを示し、 fsel はファイルでオープンされたのか、デバイスでオープンされたのかを示す。

３．５．２ 入出力処理

メインＣＰＵ側の入力処理、出力処理、そして、入出力モードの設定について述べる。

```
typedef struct strmdesc
  struct strmdesc *next; /* link for management */
  int             o_flag;/* I/O mode            */
  unsigned char   stat,  /* status              */
                  type,  /* device type         */
                  fsel;  /* file select flag    */
  DD              *dev;  /* pointer to device
                                    descriptor */
  FD              *fd;   /* pointer to file
                                    descriptor */
  SD
```

図６　ストリームディスクリプタ

・入力処理

デバイスからの入力処理を行うプロセスの関係を図７に示す。　まず、インタラプトプロセスとアービタプロセスについて説明する。

インタラプトプロセスは、データの入力（割込み）によって起動されるインタラプトルーチンによって起動されるプロセスである。　デュアルポートメモリ上のバッファサイズには制限があり、大量のデータを保持することはできない。　そこで、デュアルポートメモリのバッファからメインＣＰＵ側のシステムバッファへ、データを速やかに転送するためにインタラプトプロセスを用意した。

アービタプロセスは、ユーザプロセスからの入力要求によって起動されるデバイスハンドラと、データ入力によって起動されるインタラプトプロセスとの調停を行う。デバイスハンドラがポーリングでデータの入力を待つと、この時間は実質的な無駄時間となりシステム全体のスループットを下げる。　また、データが入力要求より先に入力された場合もそのデータを保証したい。　そこで、入力要求とデータの入力の２つの事象を調停し、デバイスハンドラの入力待ち時間によるオーバヘッドを軽減するために、アービタプロセスを用意した。

次に、デバイスからの入力が先行した場合とプロセスからのデータが先行した場合とで、それぞれの場合についてのプロセスの動作の様子について説明する。

デバイスからの入力が先行した場合、インタラプトルーチンからのメッセージ（図７のB1）によりインタラプトプロセスが起動する。　インタラプトプロセスは、デュアルポートメモリからシステムバッファへ入力データを転送し、その後、アービタプロセスに対して入力データが存在することを通知する（図７のB2）。　そして、入力要求があり次第（図７のA1,A2）、アービタプロセスがデバイスハンドラを起動する（図７のC1）。　デバイスハンドラは、入力を行ったプロセスにデータを転送し、入力処理を完了させる（図７のC2）。

プロセスからの入力要求が先行した場合、まず、デバイスハンドラが起動さる（図７のA1）。　そして、デバイスハンドラは、アービタプロセスへ要求があったことを通知する（図７のA2）。　入力データが到着するまで、デバイスハンドラとインタラプトプロセスは封鎖される。入力データが来たら、アービタプロセスが起動され（図７のB1,B2）、アービタプロセスはデバイスハンドラを起動し（図７のC1）、　入力処理を完了させる（図７のC2）。

なお、スケジューリングに対するプロセスの優先度はインタラプトプロセスが最も高く、次に、アービタプロセス、そして、デバイスハンドラである。

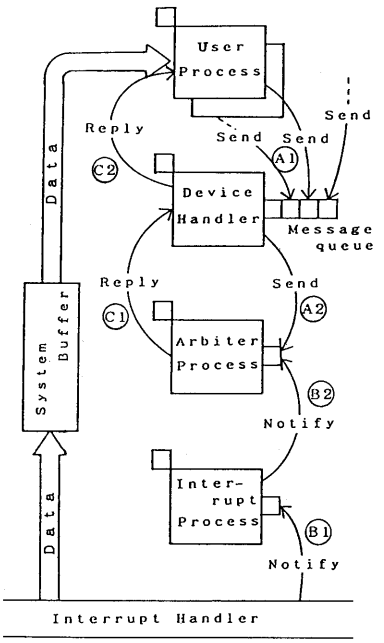

図７　入力処理に関係するプロセス

・　出力処理

出力処理に関しては、その処理の大部分をサブＣＰＵに任せて、メインＣＰＵ側はデバイスハンドラのみで行っている。　その理由は、サブＣＰＵがデュアルポートメモリからサブＣＰＵ側のバッファへ高速に転送できること、および、出力可能割込みを用いるとそのオーバヘッドが大きくなると判断したためである。

・　入出力のモード設定

入出力のモード設定は、サブＣＰＵ側で管理する。プロセスからの入出力モードの変更（または、現在の設定モードの情報の獲得）の要求は、コミュニケーションプロセスがデュアルポートメモリに書き込んでサブＣＰＵに要求を送る。　サブＣＰＵ側は、要求にしたがって自分の中で管理しているテーブルを変更し、以後、そのモードで処理を行う。

３．５．３　入出力専用プロセッサの処理

ＳＬＩＭではメインＣＰＵの負荷を軽減させるために入出力を専用に行うサブＣＰＵを用いている。　サブＣＰＵは、入力モードの設定に従ってその設定単位による割込み制御と実デバイスへの入出力を行う。　サブＣＰＵは、

・ディスプレイ
・キーボード
・シリアル回線（RS232C で UNIX マシンに接続）
・パラレル回線（プリンタに接続）
・リアルタイムクロック

のデバイスの処理を行っている。

ディスプレイに関する処理の中で画面の多重化について説明する。　***ＳＬＩＭ***では複数の画面を用意し、キーボードからの指示により瞬時に画面を切り替える構成とした。　画面の多重化は、グラフィックメモリを充分広く取ることにより実現した。　グラフィックメモリ中に１画面分の領域を複数確保した。　そして、それらの領域のうちどれを表示するかを表示回路に指示することにより、画面の切替えを行っている。　ハードウェアの構成上、７画面まで取ることができるが、現在は３画面構成として、それぞれ、エディタ、ＬＩＳＰ、ＵＮＩＸマシンの端末画面として用いている。　また、表示に対しては、ＶＴ１００のエミュレーションを行っている。

３．６　ユーザプロセス

ＳＬＩＭでは現在、ユーザプロセスとして、ＬＩＳＰとエディタを走らせている。　ＬＩＳＰは ETALisp[5]、エディタは Ez[6] を採用している。　ETALisp、Ez ともに電子技術総合研究所で開発されたものである。

ETALisp 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する機能を有する。

このエディタとＬＩＳＰは共有メモリを用いて高速にデータのやりとりができる。

４．あとがき

本報告では、ＬＩＳＰ処理専用パーソナルメディアのオペレーティングシステムシステムの開発と構成について述べた。　メッセージパッシング方式をベースとしたオペレーティングシステムであるので、マルチプロッセサシステムへの移植も容易であると考えている。

現在、このシステムは実際に稼働しており、オペレーティングシステムとＬＩＳＰプログラミング等の研究に用いている。

今後は、リアルタイム機能の実現、コンカレントＬＩＳＰに代表されるような高機能なＬＩＳＰのインプリメントとそのオペレーティングシステムによるサポートの実現等を考えている。

最後に、本研究のソフトウェアのインプリメントに直接協力して頂いた８７年度の白川研究室の大学院生・卒業研究生諸氏に感謝の意を表します。　なお、本研究の一部は大川情報通信基金研究助成によって行われたものです。

参考文献

(1) A. Kay and A. Goldberg, "Personal Dyanamic Media", Xerox PARC SSL-76-1, March, 1976

(2) W. M. Gentleman, "Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept", Software - Practice and Experience, Vol.11, 1981, pp.435-466

(3) J. Stoy and C. Strachey, "OS6 - An Operating System for a Small Computer", Oxford Univ. Computing Lab. Prog. Research Group Technical Monograph, PRG-8 and PRG-9, May, 1972

(4) D. M. Ritchie, "A Stream Input-Output System", AT&T Bell Laboratories Technical Journal, Vol.63, 1984, pp.1987-1910

(5) 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

(6) 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