CLUマシンのユーザーインターフェース

東京工業大学理学部
佐藤直樹、久野　靖、鈴木友峰、中村秀男、二瓶勝敏、明石　修、関　啓一

０．はじめに

近年におけるハードウェア技術の進歩は著しく、このため数年前までは考えられなかったような高性能の計算機を個人が占有使用することが可能になってきている。このような計算機環境下ではＯＳの役割も従来と比べて大きく変化していくものと考えられる。そのような新しい環境のためのＯＳを研究していくためには、実際にＯＳに様々な新しい機能を組み込んでみたり、その内部構造を機能分割に合わせて大幅に変更してみることが必要である。

しかし、既存のＯＳはそのほとんどが中・低レベルの言語により記述されたものであり、その記述量の多さとあいまって気軽に手が入れられるようなものとは言いがたかった。そこでわれわれはデータ抽象機能を持つ言語CLU[Lisk81]を使用して新たに高性能個人用計算機（ワークステーション）用ＯＳを開発し、これを基にして上に述べたような新しい環境下のＯＳについて研究を行うことを計画した。

本システム（我々はこれをCLUマシンと呼んでいる）の基本設計は1985年秋に開始され、現在NEC PC-98XA/XL計算機上で中核部分（プロセス管理、記憶域管理、ダイナミックリンカ）、ファイルシステム、およびいくつかの応用プログラムが動作している[Kuno88]。

本システム開発の前半の段階ではシステムが立ち上がったあと、利用者からコマンド名を受け取り、そのコマンドをディスクからロードして実行させる、ごく普通のコマンドインタープリタが使用されていた。しかし、我々が目指す新しい環境下ではより進んだ、ビットマップ画面とマウスを活用した利用者界面（ユーザーインターフェース）が必要とされるのは明らかである。そこで本システムの中核部分が十分安定して動作するようになった段階でそのような新しい利用者界面の構想と検討を開始し、現在はその試作・改良と評価を行っている段階である。

本稿の主題はこの現在試作・改良を進めている利用者界面について報告することであるが、以下第１節ではまずこの利用者界面の設計の前提としてのCLUマシンシステムの構造および特徴について概説する。続く第２節ではビットマップディスプレイを活用した利用者界面のいくつかの先例について触れ、それらにおける問題点と改良の方向について検討を行う。続いて第３節において本システムの利用者界面の基本概念について説明し、第４節で本利用者界面の実現の概要について説明しする。最後に第５節でまとめをおこなう。

１．CLUマシンシステムの構造と特徴

CLUマシンシステムの基本的な特徴は、CLU言語のコードの実行のみを対象とした単一言語系の考え方を採用した点である。これによりコードの形式をシステムの都合に合わせて決めることができ、また比較的コンパクトで見通しの良いシステムとすることができた。以下、本節では本システム中核の主要部分である記憶域管理、コード管理、プロセス管理の各部について簡潔に説明し、システム全体の特徴をまとめる。本システムのより詳しい説明については[Kuno88]を参照されたい。

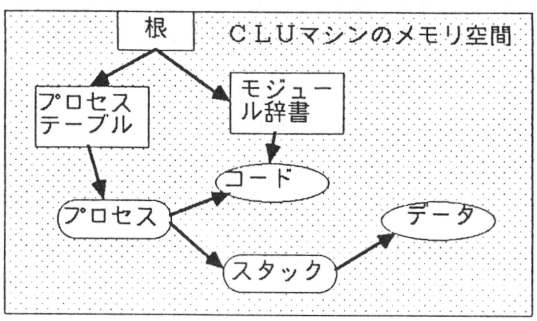

図１　ＣＬＵマシンのメモリモデル

1．1　記憶域管理

本システムでは整数、文字など16ビットの直値として扱われるごく一部の値を除くすべての値はヒープ上のオブジェクトとして記憶域管理部により統一的に管理される。システムのメモリ空間は論理的には一つの大きなヒープであり、コード、スタック、プロセス等のシステムオブジェクトも一般のデータオブジェクトと同様ここから割り当てる（図1）。

ヒープのごみ集めも記憶域管理部の仕事である。ごみ集めの方式としてはゼネレーションスカベンジングを採用している。実際のごみ集め作業は一般プロセスと並行して走るGCプロセスによって遂行される。これにより、GCによるシステム停止時間を短縮することができた。

記憶域管理部は割り込み管理を別にすればシステムの一番低レベルの部分であるが、GCを含めその大部分はCLU自身により記述されている。これはCLUの抽象化機能の強力さによるところが大であるが、システムの見通しを良くするうえで大きな効果があった。

1．2　モジュール管理

本システムではダイナミックリンカを採用することにより、コンパイル後にリンクプロセスを経ずに直接プログラムを実行開始できる。また、1つのモジュールのコードは1個だけをメモリにローディングし、全プロセスがこれを共有することにより、メモリ負荷を減少させている。これらの管理を請け負うのがモジュール管理部であり、ダイナミックリンカやローダもその一部として含まれている。

ダイナミックリンクはハードウェアのサポート無しで、純粋にソフトウェアにより実現している。具体的には全てのモジュール外への呼び出しはリンクディスクリプタ内の間接語を通じた呼び出しとしている。この語には最初はダイナミックリンカの入口点が入っているため、外部への呼び出しが最初に起こると制御はリンカに渡る。リンカは必要なら呼び出されるモジュールをロードした後、間接語を正しい入口に書き換えて戻る。以後はこのリンクディスクリプタを通じた呼び出しは直接目的のモジュールへ飛ぶので、2回目からはオーバーヘッドなしで呼び出しが行える（図2）。

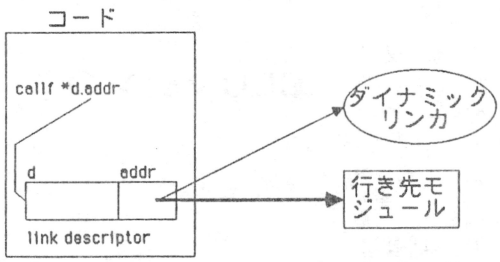

図2　ダイナミックリンキング

上記のような作業を行うため、モジュール管理部はモジュール辞書を用意してロードした全てのモジュールをここに登録する。これにより複数のモジュール／プロセスが共通のモジュールを参照した場合にも最初にロードした1個を全員で共有することができる。

システムが稼働している間、モジュール辞書には次々に新しいコードが登録されていくが、大部分のモジュールはしばらく経つともはや参照されず、不要になる。しかしこれらのコードはモジュール辞書や結合済みのリンクディスクリプタからはたどれるので通常のGCでは回収できない。このようなコードを回収するためにモジュール管理部はコードGCを行う。コードGCでは全プロセスの全スタックをスキャンし、現在呼び出されて走っているコードをマーキングしてそれ以外のコードをごみとして回収する。このとき、必要ならそのコードを指すリンクディスクリプタを初期値にリセットすることもおこなう（ダイナミックアンリンク）。これにより、本システムでは仮想記憶を採用していないにもかかわらずある程度大きなプログラムを走らせることが出来る。

1．3　プロセス管理

プロセス管理部の主要な作業はシステムの他の部分または利用者の要求に応じてプロセスの生成・制御を行うこと、およびタイマ割り込みなどを通じてスケジューリングを行いプロセス実行環境の下支えを行うことである。

本システムにおいては、各プロセスは一つのメモリ空間を共有している。このため、プロセス切り替えに際してメモリ空間を切り替える必要がなく、プロセス

切り替えを高速に行うことが出来る。また、コードは前節で述べた様にモジュール単位で全プロセスが共有するため、比較的多数のプロセスを自由に作り出すことが出来る。さらに、プロセス間でオブジェクトを共有したりプロセス間通信を通して大きなオブジェクトを渡すことも自由である。

各プロセスに固有のメモリ領域として最も大きいものはスタックであるが、スタックの大きさは予め予測することが出来ない。そこで、プロセスは生成時には最小限のスタックしか持たせず、必要になったときにその大きさを拡張する方式を採用した。すなわちCLUマシンでは手続き呼び出しごとにスタックのあふれ検査を行っているが、この時スタックが不足すると分かった場合には「スタック延ばしデーモン」と呼ばれるプロセスに依頼して自分のスタックを増やしてもらう。このようにすることでスタック領域の無駄を減らし、できるだけ多数のプロセスが走れるようにしている。

1．4　CLUマシンシステムの特徴

ここまでに述べてきた点も含めて、本システムの特徴としては次のようなものが挙げられる。

- 抽象データ型言語の採用により、抽象化を生かした構造化を行うことができた。
- 単一言語系とすることで、システムをコンパクトで見通しの良いものとすることが出来た。
- 文字型をすべて16ビット表現としたため、日本語を英字と区別なく扱うことができる。
- 記憶域管理をシステムが一括して扱うことでプログラマの負担を軽減し、メモリ資源を有効利用できる。
- ごみ集めが通常のプロセスと並行動作し、システムの停止時間が少ない。
- ダイナミックリンクの採用により、プロセス間で自然にコードが共有できる。
- ダイナミックアンリンクにより、不要なコードを自動的に追い出せる。
- プロセス切り替えが軽く、多数のプロセスを気軽に作って利用できる。
- プロセス間でオブジェクトが自由に共有でき、密度の高いプロセス間通信が可能である。

今回の新しい利用者界面の設計に際しても、これらの特徴、特にプロセスを多数自由に使用できることを生かした設計を目標とした。

本システム開発のは最初にも述べたように近い将来における計算機環境に関して様々な実験をおこなうことであり、この立場から利用者界面の設計においてもまず既存の環境の特徴と問題点を考慮し、その上でこれらの問題を克服できる可能性を試みる、という方針を取った。次節では既存のビットマップ画面を使用した利用者界面について述べる。

2．ビットマップ画面による利用者界面

現在製造・市販されている高性能個人用計算機のほとんどにはビットマップ画面とマウス装置が装備されている。これらのハードウェアを活かした利用者界面は従来の文字端末を使用したものに比べてはるかに多量の情報を利用者とやりとりすることが可能であり、従ってより効率よく使いやすいものとなり得る可能性を持っている。

しかし、このような新しい利用者界面の開発は比較的新しい研究分野であり、さまざまな試みがなされている段階である。本節ではそのようなもののいくつかを取り上げ、その長所および問題点について論じる。

2．1　Ｓｔａｒ／ＪＳｔａｒ

Starはゼロックス社で開発された文書処理ワークステーションであり、またJstar[Kami86]はその日本語版である。これらは世界で初めてビットマップ画面とマウスを採用した同社のAltoワークステーションの流れを汲み、その改良版でもある。

これらのシステムの大きな特徴は、様々な文書やそれを集めて管理するフォルダ、プリンタなど利用者が扱う対象をアイコンとして画面上に表現したことである。これにより利用者は自分が操作する対象を画面上で直接指し示すことができる。また、文書をフォルダに入れたりプリントしたりするにはそのアイコンをマウスで指しそれを「摑んで」目的の場所にもって行けば良い。このような直接操作（direct manupilation）界面は特に初心者には分かりやすく、大きな安心感を与える効果もある。

ただし、これらのシステムではアイコンの移動だけ

で表せない操作については、基本的にコマンドキーによって指示する方式を採用している。このため、どの場面でどのアイコンに対してあるコマンドキーが使えるのかは覚えてしまうまで分からないという問題がある。さらに、新しいアプリケーションを作成した場合、それに必要な操作がコマンドキーになければそれは他の方法で指示しなければならず、統一性が損なわれるのも問題である。

２．２　Ｍａｃｉｎｔｏｓｈ

Macintosh[Appl85]（以下Mac）はApple社によって開発されたパーソナルコンピュータであるが、その利用者界面はビットマップ画面とマウスを活用した先進的なものである。Macでも様々な文書をアイコンで表し、それをマウスで指し示すところはStar等と同じであるが、それらに対する操作をコマンドキーではなくメニューで行なうところが異なっている。Macでは画面の一番上に現在利用可能なメニューの一覧が常に表示されており（メニューバー）、そこをマウスで指すとメニューが降りてきて（プルダウンメニュー）操作を選択することができる。

メニュー方式の利点はコマンドキーと異なり画面から目を離す必要がないこと、必要に応じて自由に操作を追加できること、および現在利用可能なもののみがメニューとして表示されるため利用者が迷いにくいことである。さらにMacの場合はメニュー名が常に見えること、複数のメニュー→その中の各項目という階層構造があること、複数のアプリケーションでメニューの構成の共通化が計られていることも使いやすさを増す要因となっている。

ただし、Macの使いやすさは文書数も比較的少なく、基本的に一時に一つのアプリケーションしか動かさないという枠組みとうまく折り合った結果であり、その方式をより高性能のワークステーションにそのまま適用できるものではないようにも考えられる。

２．３　Ｕｎｉｘワークステーション

近年、Unixオペレーティングシステムを搭載したワークステーションが各社から発売されているが、これらの上で稼働するウインドシステムの代表的なものとしてSunTools、X、GMWなどが挙げられる。これらではMacと異なり窓は文書が開かれたものではなく、窓を作り出すような実行中のプロセスに対応している（一つのプロセスが複数の窓を管理してもよい）。従って窓を閉じた状態としてのアイコンは存在するが、文書等が常にアイコンとしてみえているわけではなく、直接個々の文書をマウスで操作しているという感じではない。ただし、一部にはMacのようにすべてのファイルをアイコンとして表示するようなものもあるが、Unixのように多数のファイルを扱う場合には、むしろこれは必ずしも便利ではないように思われる。

これらのシステムでは操作の指示には主としてポップアップメニューが使用されている。ポップアップメニューの利点は現在マウスカーソルのある位置にメニューが表示されるため、選択のためにマウスを動かす距離が小さいことである（このことはMacに比べて大きな画面を持つシステムでは重要である）が、一方複数のメニューを使い分けるためにキーボードの特殊キーと複数のマウスボタンを組み合わせて使い分ける必要があり、初心者にはややなじみにくい。

また、Unixシステムとビットマップ画面という環境はまだ比較的新しい組合せであり、その有効な利用方法が確立しているとは言いがたい[Kuno87]。利用者はいくつかの端末窓を開き、そこで従来の画面端末用エディタやシェルの指令をキーボードから打ち込み、マウスに触るのは他の窓に行きたい時だけ、というのがこれらのシステムで多く見られる実態である。

２．４　Ｓｍａｌｌｔａｌｋ-８０とＳｙｍｂｏｌｉｃｓ Ｌｉｓｐ Ｍａｃｈｉｎｅ

Smalltalk-80[Gold84]システムはゼロックス社で開発されたオブジェクト指向言語Smalltalk-80のための環境である。また、Symbolic Lisp Machine[Symb86]はSymbolics社で開発された、Lisp言語のための環境である。これらはいずれも特定の言語専用のシステムとしてプログラム開発のための道具が統合された、高度なシステムであるが、操作指示方式の点でみればここまでに述べてきたものと同じ様な方式によっている。

例えばSmalltalk-80システムの場合は、基本的にマウスで窓や窓の中の対象（例えばブラウザならクラスやメソッド、テキスト）を選択し、その対象について

の操作はポップアップメニューにより指示する。

Symbolics Lisp Machineでは複数のシステムの切り替えにはコマンドキーを使用し、その中のオブジェクトの選択とポップアップメニューにマウスを使用するのが普通である。ただし、マウスの代わりにキーボード（またはその逆）を使用することができるが、これはブラインドタイプができてキーボードから手を放したくない利用者には有効であると考えられる。

２．５　従来の利用者界面の問題点

ここまでに述べた利用者界面は、（移動する／プリンタに打ち出すという比較的特別な場合を除いては）すべてマウスで操作したい対象を選択し、続いてそれに対する操作をコマンドキーやメニューによって指示する、という方式を取っている。この選択＋操作方式は最初のワークステーションであるAltoで採用され、後続のシステムが倣ったものと考えられるが、画面に見える対象とマウスという組合せの上では確かに自然なやり方である。

しかし、見直してみるとこのやり方は操作される対象が一つで、操作自体もその対象から決って来るような限られた範囲のものであるような場合にしか適用できない。例えば、ファイルの比較のように対象物が複数であったり、コンパイラのオプションのように付加的な情報が必要になったりする場合には、一旦操作が起動された後でシステムが利用者に必要な情報を補うように要求してくるのが普通である。

また、別の問題としてHelp機能の提供を考えてみる。利用者から見れば任意の場合について、「もしこれを選択してこのメニューを選んだら何が起きるか」をやってみる前に予め知りたいのが普通である。しかし既に選択もメニューも操作そのものを指定するのに使ってしまっているので、「これは本番ではなくHelpである」ということを示すのにモードを使ったり特別なキーを押しながら操作する等、別の手段を考えなければならない。

これらの問題はすべて選択＋操作型の界面が本来比較的限られた場合にしか適用できないという事に起因しているものと考えられる。

３．新しい利用者界面方式の構想

前節で述べたように、選択＋操作型の界面は必ずしも万能でないと考えたので、筆者らはCLUマシンシステムの利用者界面の設計に際してはこれとは別のやり方を試みることにした。本節ではこの新しい方式の基本的な考え方、具体的な動作および特徴について触れる。

３．１　基本概念

本方式を考案するきっかけとなったヒントは、Star等で採用されているもう一つの動作である、「アイコンを持って行って、重ねる」という操作である。これまでの例ではこの動作は対象の移動／複写／転送／印刷といったごく限られた操作の指定にのみ使用されていたが、本質的に「ＡをＢに重ねる」ということで複数の対象物を関連付ける事ができ、しかも重ねた状態でさらに別のものを重ねることができるので、柔軟な拡張が可能である。

例えば、あるファイルをコピーしたければそのファイルを表すアイコンを「複写」のアイコンに持って行って重ねると、複写されたファイルのアイコンが画面に現れる（図３上）。元のファイルのアイコンは最初にあった場所に戻る。Macなどではコピーはメニューやコマンドキーから「複写」を選ぶことにより指示していたが、それにくらべて本方式の方が直接的で視覚的にも分かりやすい。

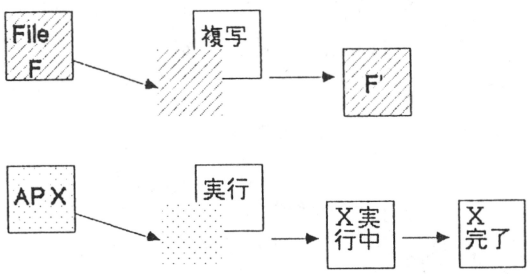

図３　重ねることによる操作指定

別の例として、あるアプリケーションを実行したければそのアプリケーションを表すアイコンを「実行開始」のアイコンの所に持って行って重ねると実行が始まる（図３下）。実行中のアプリケーションは元のアイコンとは別のものが作られ、アプリケーションが動

いている間中そこに見える。最初に重ねたアイコンは元の場所に戻るので、同じアプリケーションをもう一つ並行して動かしたければこれをもう一度「実行開始」に重ねればよい。実行中のアプリケーションが終了すると（または死ぬと）、そのアイコンは無くなってしまうのではなく、「死骸」を表すアイコンに変化する。これは、終了したプロセスについてもそれに関する情報を得たい場合があり、そのためにはアイコンが残っている必要があるからである。

３．２　操作対象の視覚化

前節の例からもわかるように、本方式では利用者が意識する対象をすべて視覚化してアイコンで表すことを原則とする。そのような対象としては現在の所ファイル、文字列、アプリケーション、ツール、プロセスなどがあるが、必要なら利用者が新しいカテゴリーを追加して自由に拡張することができる。

これは、従来の視覚型インターフェースが比較的小数種類の対象のみをアイコンとして扱っていたのと大きく異なる。例えばStar／JstarやMacではアイコンは基本的にファイル／ディレクトリに対応していた（ただし、例外としてプリンタやディスクなどのデバイスを含むが、これらは自由には拡張できない）し、XやSunToolsなどではアイコンはウィンドの閉じたもので、ほぼプロセスに対応している。従って、これらのシステムでは視覚化されていない対象については利用者がその存在を「暗記」して「めくら」で操作しなければならない（この観点から見ると、従来のTSSは「全ての対象を視覚化せず、めくらで操作するようなシステム」であると考えることができる）。

３．３　アイコンの管理

前節で述べたような方式を採用した場合、アイコンをどの様に管理するかが問題となる。「重ねる」ためには必要な道具やファイルのアイコンを出しておく必要があるが、一方で画面に置いておけるアイコンの数には限りがある。そこで、本方式では「アイコンは常に存在するのではなく、必要になったときに作り、不要になったら捨てる」という方針を採用した。各アイコンはさまざまなツール群から作り出される。

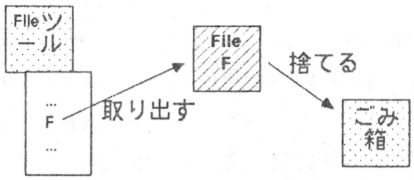

図４　アイコンの生成と消滅

例えばファイル／ディレクトリに対応するアイコンはファイルシステムを覗いたり探索したりするツールによって作り出される（図４）。一旦出来てしまえばそのアイコンはファイルのさまざまな操作をするのに使用できる。また、同様に自分の使用する様々なコマンドやアプリケーションを管理するためには「道具箱」に相当するツールを開いて、そこから必要なものを取り出す。一旦取り出されてアイコンになったものは前節で説明したようなやり方で使用することが出来る。

画面がアイコンで込み合ってきた時には必要に応じてこれらのアイコンをごみ箱に捨てて画面を整理する。ただし、「捨てる」のはあくまでもアイコンであり元のアプリケーションやファイルが無くなってしまうわけではないので、また必要になればこれらをツールを用いてアイコンとして再び取り出して来ればよい（削除のためには、別の「削除」という道具を使用する）。

MacやStar/Jstarではとりあえず見えなくてもよいファイルのアイコンはフォルダに入れてしまっておくことができるが、現在開いているフォルダのファイルはたとえ作業と関係なくても全てアイコンとして見えてしまうので、特にファイル数が多くなった場合に煩わしく感じることが多い。これに対して本方式では利用者が意識して操作しているような対象だけを常にアイコンとして画面に出しておくという方針により、上記の問題点を解消している。

３．４　複数対象の指定と部分結合

選択＋操作型の界面で問題であった、複数対象の指定を必要とするような操作は、本方式によれば各対象を順次重ねることで自然に表現できる。例えば、ファイルの比較の場合は「比較する」アイコンの上に一つ目のファイルＡを重ねただけではまだ実行が開始でき

ないので、代わりに「ファイルAと比較する」という中間状態のアイコンになる。これに二つ目のファイルを重ねたとき初めて実行が開始される。同様に、コンパイラのオプションもまず「コンパイル」の上に必要なオプション（実際にはオプションを指示する文字列）を重ねて「オプション・・・でコンパイルする」を作り、これにファイルを重ねるとコンパイルが開始される（図5）。

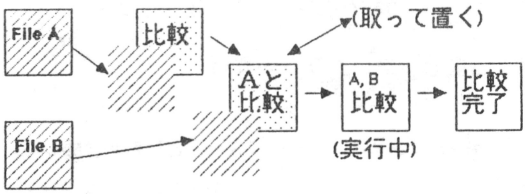

図5　複数対象の指定

複数のファイルをまとめてコンパイルする場合には、最初のファイルを指定した時点でコンパイルが開始されると困る。このような場合には、「コンパイル」にいったん「停止」を重ねて実行を止めた状態で必要なものを順次重ね、最後に「実行」を重ねればよい。最初に出てきた、特にパラメタがなくていきなり「実行」により起動するアプリケーションもこの「停止」状態にあるものと考えればよい。

この方法は従来の「起動するとオプションを聞いてくる」方式に比べて、すべての操作が「重ねる」という動作で統一されている利点がある。さらに、中間的な状態がすべてアイコンとして残っているので、「さっきと同じ事をしたい」場合や「途中までは同じだけれど少しだけ違う事をしたい」場合にもそのためのアイコンが画面上にあることになる。これらのアイコンのうち繰り返し使うようなものは、道具箱に入れることにより、後日そこから取り出して利用できる。そうしなかったものはシステムを止めるまでの間だけ画面上にある。全く不要なものは画面上でじゃまになるのですぐにごみ箱に捨ててしまえばよい。

例えば特定のファイルを複数のファイルと繰り返し比較する、繰り返し同じオプションでコンパイルする、といった場合にこの中間状態を保存・再利用する機能は有用である。もっと極端な例としては、ファイルの中身を表示する道具といつも見ておきたいメモを記したファイルを結合して「いつものメモを表示する」アプリケーションを作ることもできる。

3．5　アプリケーション同志の結合

前節ではアプリケーションに対象を結合して新しいアプリケーションを作る場合について述べたが、アプリケーション同志を結合することも当然考えられる。最も単純なのは「順次実行」のアイコンにアプリケーションを複数重ねることでこれらを順番に実行するアプリケーションを作ることである（図6）。

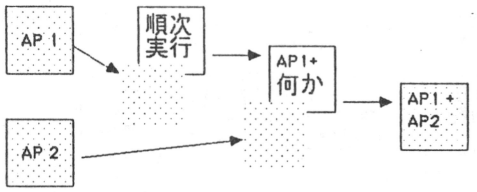

図6　アプリケーションの結合

これのごく自然な発展はUnixのパイプである。CLUマシンシステムもUnixと同様に標準入力／出力の概念を持つ。特に指定しない場合にはプロセスの標準入力／出力は一つの文字入力／表示用ウインドになるが、これをパイプラインの形に結合することもできる。ただし、新しい環境下では文字（ないしバイト列）をやりとりするようなアプリケーションの地位は相対的に低下するものと予想される。

このほかに「反復実行」「条件分岐」なども考えられるが、あまり複雑になってくるとこのように視覚的に組み合わせるやり方が良いかどうかは疑問である。むしろ、PADやHCPなどの図形表現の中にアイコンをはめ込んで組み合わせるような「視覚的アプリケーション結合ツール」を用意することが正解であると考えられる。

このようにして作ったアプリケーションはそのままでは一時的であるが、これを道具箱に入れておけば自分専用のアプリケーションとして保存し、繰り返し使うことができる。このように自分固有の指令を増強して行けることはUnixなどでは当たり前であったが、ビットマップ画面とマウスに基づく界面では多くの場合

新たにプログラムを作る必要があり、弱点であったといえる。

３．６　文字列の扱い

先に説明した例の中に、コンパイラにオプションを重ねる、というものがあったが、この「オプション」は実際には（Unix風に言えば-c、-Oなどの）文字列である。そのほかにも「探したいパターン」、「接続したいホスト名」など、文字列が必要とされる場面は多く存在する。しかし、ビットマップ画面とマウスに基づく界面では通常手がマウスにあるので、文字列をキーボードから打ち込むのは煩わしく時間のかかる操作となる。

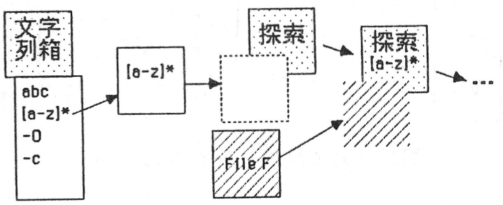

図７　文字列と文字列箱

このため、本方式では一旦打ち込んだ文字列はできるだけ保存して再利用することでキー入力を減らす方針を取る。具体的には、文字列は「文字列箱」と呼ばれるツールを用いて、最初はキー入力により作られる。一旦作られた文字列は他の対象と同様にアイコンを持ち、これを通じて操作できる（図７）。繰り返して使うような文字列については、アプリケーションの場合と同様、文字列箱に入れておけば消えることなく保存できる。使いたい文字列が文字列箱に入っていれば、そこからアイコンにして取り出し、使用することができる。

３．７　プログラムの実行過程

本節までにプログラム、コード、プロセス、アプリケーション、ツールなどの用語を使用したが、これらはすべて互いに関連している。以下本節ではプログラムの実行過程についてまとめ、これらの関連を整理して示す。まず、プログラムとは計算機に手順を指示するもので、ソースプログラムとそれを翻訳したコードファイルがある（CLUマシンではダイナミックリンクを採用したため、再配置形式のコードファイルがそのまま実行形式ファイルとなる）。

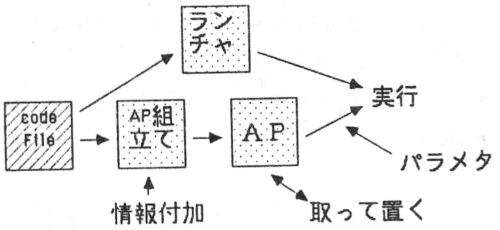

図８　プログラムの実行

コードファイルその物にはほとんど情報が付随していない。これにこのコードは最低限どのようなパラメタを必要とし、それに加えてどのようなパラメタが指定できるか、どのようなアイコンを表示するか、何を重ねたとき何が起こるべきか、などの情報を付加したものがアプリケーションである。コードファイルからアプリケーションを組み立てるのには専用のツールを使用する（図８）。ただし、すべてのコードファイルを一旦アプリケーションにしなければ実行できないのでは煩わしいので、コードファイルを直接実行させるためのツール（ランチャ）も別に用意されている。ツールとは、アプリケーションのうち本方式の界面を構成する中核部分に相当するものを特にこう呼んで区別するものである。

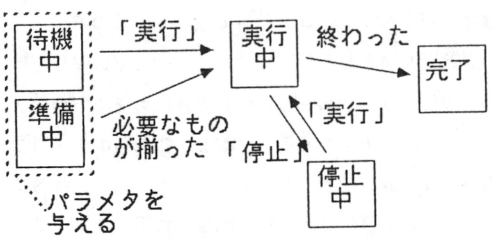

図９　アプリケーションの実行過程

アプリケーションは準備、実行中、完了の３つの状態を持つ（図９）。準備状態はパラメタが揃うのを待っている状態であり、必要なものが揃うと実行中となり、実行が終わると完了状態になる。前２者については、それぞれ待機／停止状態に置くことができる。と

くに待機状態の場合は多数のパラメタを与えるのに有効なことは前に述べたとおりである。完了状態ではその結果を見たり保存でき、また「実行」を重ねることで再実行させることもできる。

３．８　環境制御型のアイコン

ここまでに述べてきたやり方では、すべての動作はアイコン同志を重ねた際に起き、それによって新しいアイコンが生まれたり重ねられたアイコンの状態が変化する。動作が起きた後は重ねた方のアイコンは元の場所に戻り、次の動作に備える。

しかし、もう一つの可能性として「重ねたまま置いてある間だけ・・・する」という方式を導入することも考えられる。このようにすると、あるアイコンを重ねた上でさらに別のアイコンを重ねることができるので、これを利用して直交性のある界面を実現することができる。例えば先に挙げたHelpを例に取ると、一般に任意のアイコンＡをアイコンＢに重ねたときに何が起きるかの情報を得たければ、まずＢを「Help」に重ね、その状態でさらにＡを重ねればよい。さらに情報を見た結果、本当にその動作を実行したければ「Help」を取り除くことで行える（図１０）。

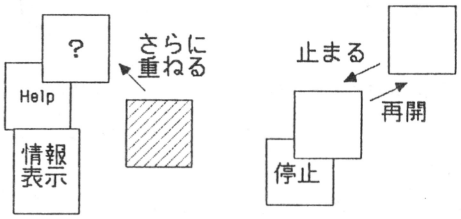

図１０　環境型のアイコン

上の例は見方を変えれば「実行する代わりに情報を提供する」という環境で動作を実行していると考えることもできる。この立場に立つと、そのほかにも様々な有用な環境を考えることができる。例えば「停止」や「nice」というアイコンの上にプロセスを重ねればそのプロセスは一時的に停止状態や低優先順位状態になるが、もとの所に戻せば元の状態に戻る。

従来はこのような状態制御は「停止させる」「再開させる」といった指令を実行することで行っていたが、それに比べて本方式のやり方は状態が視覚的に見え、より直接的であるという利点がある（本方式でも「停止」や「実行」の方のアイコンをプロセスに重ねることで指令型の操作とすることもできる）。

４．実現

本節では前節で解説した新しい利用者界面方式の実現について、その概要を述べる。CLUマシンではＸと同様、画面の管理は基本的に画面サーバが行なう。これは、同期を必要とするタスクはそれぞれサーバに割り振って担当させるという、CLUマシン全体の方針に沿ったものである。また、CLUマシンではUnixなどと比べるとプロセスの切り替えが軽く、プロセス間で大きなデータを受け渡すのが容易であるので、サーバ方式はその特徴を活かしているといえる。

画面上に表示されるアイコンの管理、「重ねる」ことによる動作の起動等の作業は代理人（Agent）と呼ばれる特別なプロセスによって行なわれる。代理人は各ツールによってアイコンが作られるときに同時に作られ、そのアイコンに関するすべての動作を受け持ち、利用者の操作をそのアイコンに対応している対象物に及ぼす仲介役を勤める（図１１）。

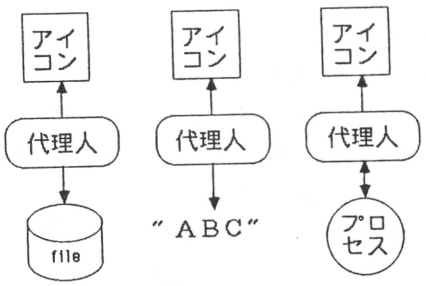

図１１　代理人と実体

代理人の種類としてはこれまでに説明してきたアイコンの種類に応じて

・ファイル
・文字列
・準備／待機中のプロセス

・実行中のプロセス
・実行完了したプロセス

を用意しているが、この他にも必要に応じてさまざまな種類の代理人を利用者が用意することができる。

このため、代理人の動作はプログラムとして組み込むのではなく、代理人ファイルと呼ばれるファイルにどのような重なりが起きたときにどのような動作を行うかを記述することによって指定する。ファイル、文字列などでは一つずつ動作が違うというようなことは通常ないので、標準の代理人ファイルを用意しておけばよい（利用者が「標準的でない」ファイルや文字列を作ることも自由である）。一方、アプリケーションプログラムのような場合には一つずつが異なったパラメタを必要とする。このために、アプリケーション組み立てツールがそれぞれのアプリケーションに適合した代理人ファイルの生成をおこなう。

代理人ファイルに記述できるのは、何らかの重なりが生じた／なくなった事に対応して動作を起こす（実際には下請けのコードを起動する）事のみであるが、ツールによってはアイコンに対する動作をより有機的に制御したい場合もある。このため、代理人は実行プロセスを生成する場合には通信用の郵便箱をプロセスに渡す。画面を意識して自らウインドを操作するようなプログラムはこの郵便箱を用いて代理人と通信し、アイコンに起こる事象を直接受け取ったりすることができる。

逆に、このような制御について関心のないプログラムの場合には特に何もしなければこれらの事象については知らないで済む。例えば原始的な端末入出力を想定したプログラムの場合にはその入出力のための窓が自動的に提供されるが、これはライブラリと代理人によって管理され、プログラマは特にそのことを考慮する必要はない。

この実現方式を既存のビットマップ画面とマウスを使用するシステムと比較すると、既存のシステムのほとんどではアイコンの管理を直接ウインドマネージャないしそれに相当するモジュールが行なう点が大きく異なる。このため、アイコンに対して行なえる操作はあらかじめ決められていて、自由に拡張するのは困難である。これに対し本システムの方式では画面サーバと代理人を分割することにより、システムの見通しを良くし、また様々な種類の対象物を扱うように拡張することも容易である。

5．まとめ

本稿で述べた利用者界面については、まだ現在試作・改良の途上にある。このため、その評価について述べられる段階ではないが、少なくとも既存の利用者界面の問題点の一部を克服する試みとしては有効であると考えている。最初にも述べたように、CLUマシンシステム全体が新しい計算機環境のための実験素材としての意味合いを強く持つものであり、この上で今回説明した利用者界面だけにとどまらず、様々なものを試して行きたいと考えている。

謝辞

東京工業大学理学部情報科学科木村研究室および米澤研究室の方々との討論は本稿をまとめる上で大変有益であったので、ここに感謝します。

参考文献

[Lisk81] Liskov et al., CLU Reference Manual, Springer, 1981.

[Kuno88] 久野ほか、CLUマシンシステムの開発、投稿中.

[Kuno87] 久野ほか、流れて行かないUnix環境をめざして、夏のシンポジウム「究極のプログラミング環境」講演論文集、1987.

[kami86] 上谷編、ＪＳｔａｒワークステーション、丸善、1986.

[Appl85] Apple Computer, Inc., Inside Macintosh Vol. I～IV, Addison-Wesley, 1985.

[Gold84] Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, 1984.

[Symb86] Symbolics Inc., Symbolics Lisp Machine Manual, 1986.

本研究の一部はNTT(株)および日本電気(株)の研究補助金によっている。