CLU マシンシステムの開発†

久野 靖†† 佐藤直樹††* 鈴木友峰††**
中村秀男††* 二瓶勝敏††
明石 修†† 関 啓一††

高性能個人用計算機向け OS を，データ抽象機能を持つ言語 CLU を用いて開発した．本システムの基本設計は1985年秋に開始され，現在 NEC PC-98XA/XL 計算機上で中核部分（記憶域管理，プロセス管理，モジュール管理），ファイルシステム，CLU コンパイラ，ウィンドシステム，ネットワークモジュールおよびいくつかの応用プログラムが動作している．本システムは単一言語系の考え方を採用することにより，コンパクトで見通しのよいシステムにできた．また CLU 言語のデータ抽象機能は，モジュール間の独立性を高め，分かりやすく構造化されたシステムとする上で効果があった．

1. はじめに

最近の計算機ハードウェア技術の進歩により，高性能計算機を個人で占有使用することが可能になりつつある．そのような計算機環境においては OS の役割も大きく変化して行くものと考えられる．しかし，既存の OS はそのほとんどが中・低レベル言語により記述されたものであり，記述量の多さとあいまって気軽に手が入れられるものとは言いがたい．

そこで我々は，高性能個人用計算機（ワークステーション）向け OS を，データ抽象機能を持つ言語 CLU[3]）により記述・開発することを計画し，これを CLU マシンシステムと名付けた[1]．本システムの基本設計は1985年に開始され，現在 NEC PC-98XA/XL 計算機上で中核部分，ファイルシステム，CLU コンパイラ，ウィンドシステム，ネットワークモジュール，およびいくつかの応用プログラムが動作している．以下第2章で本システムの設計目標および特徴について述べ，続く第3，4，5章で本システムの中核である記憶域管理，モジュール管理，プロセス管理の各部について解説する．続く第6章ではファイルシステムをはじめとする主要なサブシステムについて簡潔に紹介し，第7章では本システムの大きさおよび開発環境について述べる．最後に第8章でまとめを行う．

† Development of the CLU Machine System by YASUSHI KUNO, NAOKI SATO, TOMOMI SUZUKI, HIDEO NAKAMURA, KATSUTOSHI NIHEI, OSAMU AKASHI and KEIICHI SEKI (Department of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科
* 現在 日本電気(株)
** 現在 (株)日立製作所

2. 設計目標と特徴

本システムの開発に当たって我々が目標としたことは，高級言語で記述された，コンパクトで自由に手の入れられるシステムとすることであった．また，使いやすいシステムであるためには複数のプロセスが協調して利用者環境を構成して行くことが必要であると考えたため，多数（1,000 個程度）のプロセスが生成でき，かつプロセス間で密接な連絡が可能であることも目標とした．

上述の目標に基づいて検討した結果，アセンブリ言語を含む複数の言語を機械語に翻訳して走行させる通常の OS では，その記述量が多く目標の達成は困難であると判断した．そこで，特定の言語を設定してその言語コードのみを効率よく走らせる，単一言語系の考え方を採用した．具体的には言語として CLU を採用したが，これは CLU のデータ抽象機能がシステムの構造化に有用であると判断したこと，および CLU のような高級言語による OS の記述性の評価を行う必要があると考えたことが理由である．上記を含め，本システムの特徴としては次の点が挙げられる．

- 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る．
- 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にできる．
- 文字型をすべて16ビット表現としたため，日本語と英字を区別なく扱うことができる．
- 記憶域をシステムが一括して管理する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 (Garbage Collection, GC) が通常のプロセスとして並行動作可能なため，CPU のあき時間を有効利用でき，またシステムの停止時間を短くできる．
- ダイナミックリンクによりコンパイル後リンクの段階を経ず直ちに実行が開始でき，またプロセス間でコードが共有できる．
- プロセスがオブジェクトを自由に共有できるため，プロセス間の密接なやり取りを可能にしている．

次章以降ではシステムの各要素について上記の特徴を含めて解説する．

3. CLU マシンの記憶域管理

CLU マシンの記憶域は「全プロセスに共通な一つの空間」であり，この中に CLU のオブジェクト群が配置される．**図 1** に示すように，生きている（ごみでない）オブジェクトはすべて，システムにただ一つ存在するルートオブジェクトからたどることができる．通常のデータのほかに，機械語コード，実行スタック，システムの各種テーブルなどもすべてオブジェクトとしてこの中に存在する．このようにすることですべての実体がプロセス間で共有可能となり，また記憶域を統一的に管理できる．以下本章ではオブジェクトの構造と記憶域管理について説明する．

3.1 オブジェクトの構造

CLU マシンでは整数型・文字型等の直値データ (16 ビットで現され，直接レジスタおよび実行スタック上に置かれる）を除く全データはオブジェクトとして統一的に扱われる．**図 2** に示すように，オブジェクトのヘッダにはその大きさ，種類，状態などが格納されている．さらにオブジェクトが他のオブジェクトへのポインタを含む場合には，そのポインタの位置と個数の情報も格納されている．**図 3** に示すように，これらは記憶域管理レベルではすべて _obj 型の実体として扱われる．

システムの他のモジュールから見ると，これらのオブジェクトは_avec（ポインタベクタ），_wvec（直値ベクタ），_code (機械語コード)，および_stack（実行スタック）の 4 種類に分かれているが，記憶域管理はこれらの構造の違いについて関知しない．(ただし，スタックについてはハードウェアが戻り番地を操作するため，他のオブジェクトと完全に同じ構造にはできない．このため，記憶域管理の一部分では_stack 型を他の_obj 型と区別している箇所もある．)

さらに一般利用者が見る array, record, string などの組み込みデータ型は，上述の 4 種類のシステムオブジェクトに基づいて実現されている．このように同一実体を，システムの各レベルでそれぞれ異なる型として扱うことができるのは抽象データ機能の特徴であり，システムの構造化に有効である．

3.2 記憶域管理の構成

記憶域管理の仕事は，他のモジュールが使用するオ

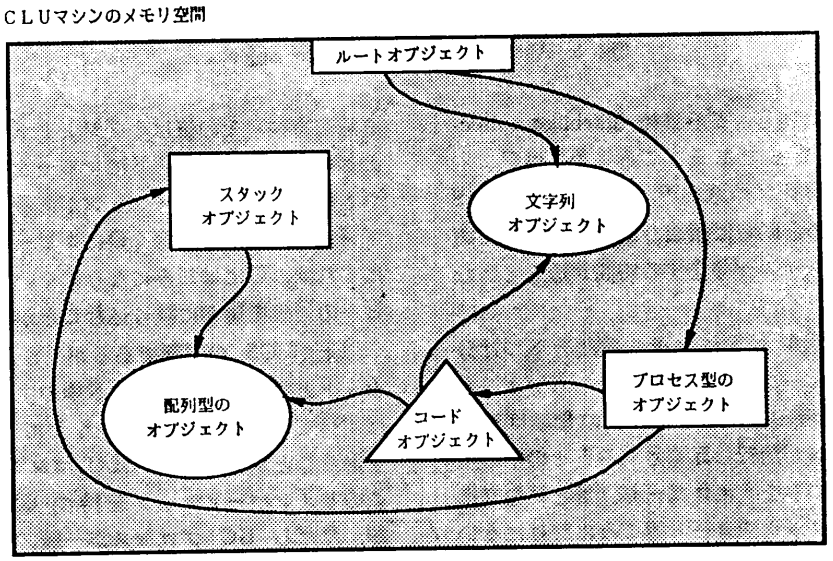

図 1 CLU マシンの記憶域モデル
Fig. 1 Storage model of the CLU machine.

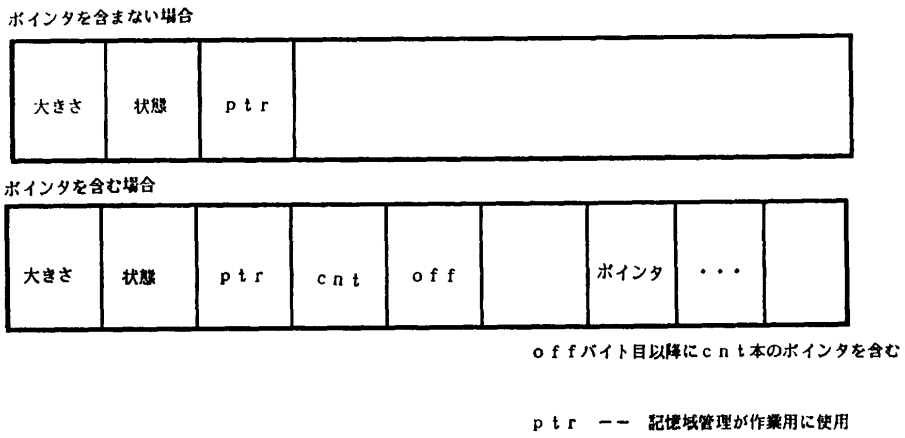

図 2 オブジェクトの構造
Fig. 2 Structure of the CLU machine objects.

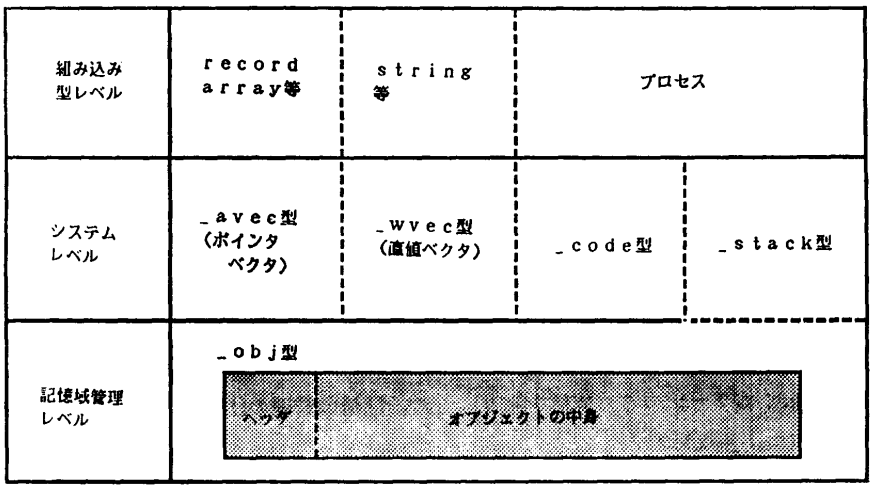

図 3 オブジェクトの階層構造
Fig. 3 Abstract hierarchy of the objects.

ブジェクト領域を管理し，ごみ集め (garbage collection, GC) を行うことである．ごみ集めにはジェネレーションスカベンジング (GS)[4] を用いている．GS の利点は，ごみ集めと同時に領域の詰合せが可能であること，およびシステムが停止する時間が短くて済むことである．

GS では図4に示すようにメモリ空間を3つに分割し，それぞれニュー (N)/フューチャー (F)/オールド (O) スペースと呼ぶ．オブジェクトの領域は通常Nに取られるが，Nが消費し尽くされるとその中の生きたオブジェクトのみをFにコピーし，NとFを交換する．Oには，スタックやコード等ごみになりにくいものを配置する．

また，各オブジェクトはジェネレーションと呼ばれる値を持つ．その値はオブジェクトが最初に作られたときに1とし，NからFにコピーされる度に1ずつ増やされる．ジェネレーションがある一定値を越えたオブジェクトはFではなくOにコピーされる．このようにすれば，Oにはごみになりにくいものが集まり，コピーの対象を適切に絞り込むことができる．

以上が GS の原理であるが，さらにリメンバードセットという表を用意し，N中の生きたオブジェクトはすべてここからたどれるようにすることで効率化を計ることができる[4]．CLU マシンの記憶域管理はこのリメンバードセットを持つ GS を採用しているが，ただし GC プロセスが一般プロセスと並行して実行される点が特徴である．次節ではこの点について述べる．

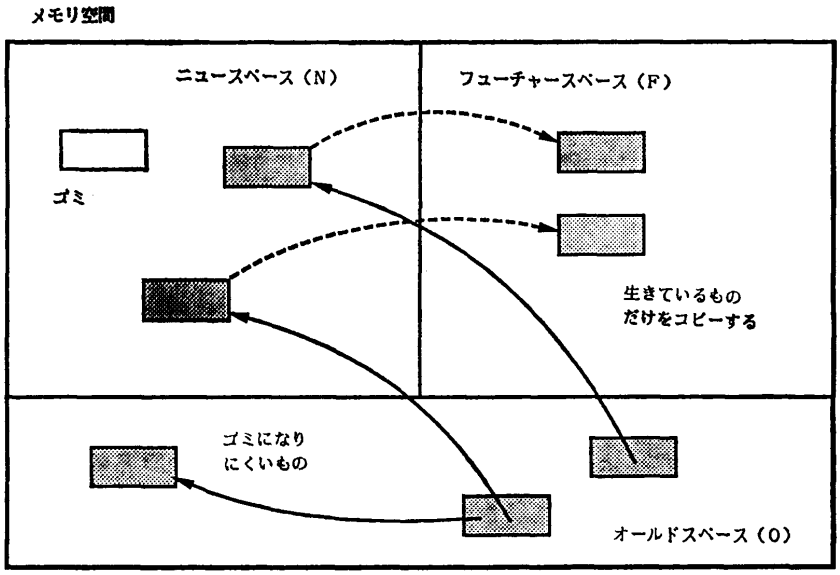

図 4 記憶域の構成と GS の原理
Fig. 4 Storage structure of the CLU machine.

3.3 並列ジェネレーションスカベンジング

CLU マシンでは GC も一つのプロセスであり，記憶域を消費する他のプロセス群と並行に走る．これにより，GC が他のプロセスを止めてしまうことが少なくなる．このようにした場合に留意すべき点として，

(1) GC 中に他のプロセスが領域割り当てを要求したときの扱い

(2) GC プロセスと他のプロセスのオブジェクトへのアクセス競合，具体的には

(2-a) 他プロセスが GC により既にFにコピーされたオブジェクトのコピー前の実体をアクセスする場合

(2-b) GC がたどり終わったポインタを，他プロセスが別の値に書き換えてしまう場合

が挙げられる．

このうち(1)に関しては，GC 中は N は一杯であるが，Fには余裕があるので GC 中の割り当て要求に対してはFから領域を割り当てることで対処した．

(2-a)については，オブジェクトのヘッダにコピー済みのフラグとコピー先を格納し，他のプロセスはアクセス時にフラグを見てコピー済みならポインタをたどってコピー先の本物をアクセスするようにした．

(2-b)については，GC は N を複数回たどり，最後の1回だけ他のプロセスの実行を止めてたどることで対処した．ほとんどのオブジェクトは最後のフェーズ以前にコピーされるので，これによって最後に GC が排他的に実行する時間を少なくできる．

3.4 オールドスペースの GC

GS では O にごみになりにくいものが配置されるが，Oに割り当てられたオブジェクトも長い時間の間にはごみになって回収される必要が生じる．O領域はNに比べて大きく，オブジェクト数も多いため GC には基本的に長い時間を要し，その間のオブジェクト需要を満たす領域を別に用意することは難しい．また，コードやスタック等プロセス実行のために重要な資源が含まれているため，一般プロセスが走っている状態で GC を行うのは難しい．一方，その頻度は通常の GS に比べれば小さい．

これらの理由から，O 領域の GC は一般プロセスの実行を停止した状態でマークスイープ（ルートオブジェクトからすべての生きているオブジェクトをたどって印をつけ，続いてオブジェクト領域を先頭から順に調べ，印のついていないオブジェクトをごみとして回収する方式）により行っている．ただし，その間も割り込みハンドラ等は走行できる．

4. モジュール管理

CLU マシンでは全オブジェクトが単一空間中に共存するが，コードについてもこの例外ではなく，通常のオブジェクトと同じ空間に取られ GC の対象となる（このため全コードは再配置可能に作られている）．本章ではコードオブジェクトとその管理について説明する．

4.1 モジュール辞書とコード管理

プロセスは動的に作られ，それに応じてコードもモジュール単位で動的にロードされるため，コード間の参照もまた動的に解決される（ダイナミックリンク）．そのため，すべてのコードはモジュール辞書に登録され，モジュール管理部により統一的に管理されている．

コードオブジェクトは常にモジュール辞書から参照されているため，GC によって自動的に回収することはできず，別の方法で不要になったコードを発見する必要がある．このための一つの方法はリファレンスカウントを使用することであるが，CLU が proctype 型（手続きへのポインタ型）を持つこと，相互呼び出しによるループが生じることから手順が複雑になり，利点が少ないと判断した．

このため，CLU マシンでは全プロセスの全スタックをたどって実行中のコードに印をつける方式を採用した（このため戻り番地からコードの先頭が分かるようにポインタ形式を工夫している）．この方法は一見非効率的に思えるが，実際にはコードはO領域に取られるためコード GC は必ずO領域の GC に伴われて実行される（コード GC で不要とされたコードの領域を，引き続くO領域 GC により回収するため）．既に述べたようにO領域の GC は比較的長い時間を要するが，コード GC はそれに比べれば短い時間で完了するため，問題はない．

4.2 ダイナミックリンク/アンリンク

前述のように，CLU マシンではモジュールはダイナミックリンカにより必要になったときに動的にリンクされる．このためモジュール間の呼び出しは，図5に示すように必ずリンクディスクリプタを介した間接呼び出しとなっている．ディスクリプタは呼び出し側のコードに付随していて，他モジュールへの参照ごとに1個用意される．まだ参照が解決されていない状態ではディスクリプタの呼び出し番地はリンカを指しているので，最初の呼び出しが起こると制御はリンカに渡る．リンカは呼び出されるとスタックから戻り番地を通じて呼び出したコードとディスクリプタのありかを割り出し，参照を解決する（この時，行き先のコードをモジュール辞書から検索し，もし存在しなければローダを呼び出してディスクからコードをローディングする）．行き先が決まると，リンカはディスクリプタ内の呼び出し番地をその行き先に書き換えた後，直接行き先に分岐する．2回目以降の呼び出しではディスクリプタは正しい行き先を指しているので，オーバヘッドなしで呼び出しが実行できる．

このようにして次々と必要なコードが入って来ると，メモリが一杯になり，前項で説明したコード GC が起動されて実行中でないコードを捨てる．この時ディスクリプタ内の行き先が捨てたコードを指したままだと次に呼び出しが起こったときに困るので，そのようなディスクリプタは再びリンカを指すように書き換える．これをダイナミックアンリンクと呼んでいる．

以上の機能は，OS が自動的にオーバレイ処理を行っていることに相当する．このため CLU マシンでは仮想記憶を採用していないにも関わらず実記憶に入り

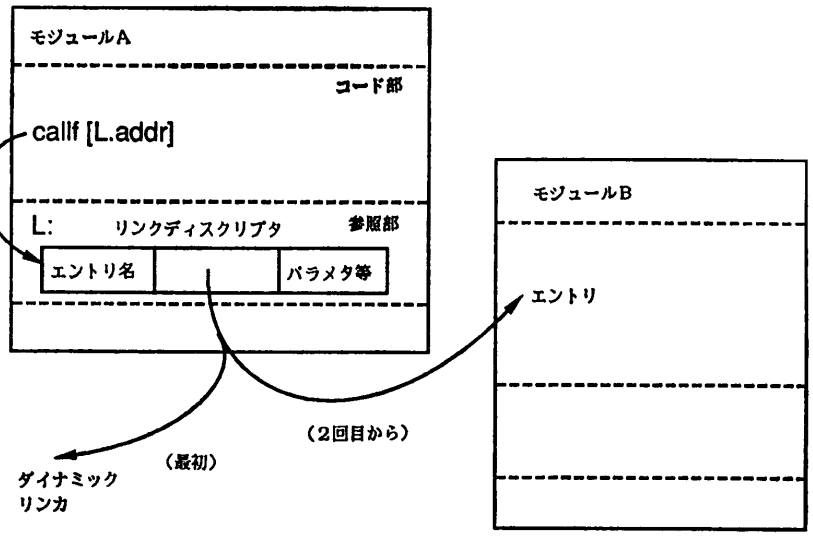

図5 リンクディスクリプタとダイナミックリンキング
Fig. 5 Link descriptors and dynamic linking.

きらない大きさのプログラムを走らせることができる．具体的には，一時に実行中の手続き群のコード量と恒久的（O 領域に配置される）データ量の合計が O 領域の大きさを越えないなら実行が可能である．

4.3 パラメタ付きモジュール

CLU 言語ではモジュールがパラメタを持つことを許している（Ada の generic パラメタに相当）．例えば array というモジュールは型パラメタを持ち，これに int, string などの型を与えることで array [int]（整数の配列），array [string]（文字列の配列）のような個別の型を具体化できる．この場合，配列全体をコピーする copy という操作はその要素をコピーする必要があるので，パラメタに応じて int$copy, string$copy 等異なる操作を呼び出す必要がある．

このため，パラメタ付きモジュールをロードする時には図 6 左に示すように，パラメタによって変化する必要がある部分の情報を集めてパラメタブロックという構造を作る．array の例ではパラメタブロックには「パラメタが 1 個あり，（仮にその名前を t とすると）このモジュールを具体化する時には t$copy が必要である」という情報が含まれる．

具体化の際には図 6 右に示すように，パラメタブロックをもとにコールブロックという構造を作る．これはパラメタに依存した呼び出しのためのリンクディスクリプタと，そのような呼び出しが必要な操作のための仮エントリを含む．例えば array のコールブロックは t=int である，という情報，int$copy を指すリンクディスクリプタ，array [int] $copy のための仮エントリを含む．他のモジュールから array [int] $copy を呼ぶ時はいったんこの仮エントリに分岐し，int$copy を呼ぶ準備をしてから本体のエントリに分岐するようになっている．

5. プロセスとプロセス間通信

CLU マシンではプロセスが並列実行の単位となっている．他のシステムではオブジェクトが並列実行の単位となっているものも見かけるが，CLU マシンの場合にはそのような方式では並列実行単位が小さすぎて効率のよい実現が難しいと考えたため，より「普通の」アプローチを取ったものである．本章では CLU マシンのプロセスとプロセス間通信について述べる．

5.1 プロセスとプロセス型

プロセスはプロセス型のオブジェクトを作ることにより生成され，その操作を通して自由に操ることができる．プロセスもまた CLU マシンの単一空間中に存在しており，プロセス間でオブジェクトを自由に共有することが可能である．各プロセスは常に次の 4 状態のいずれかにある．

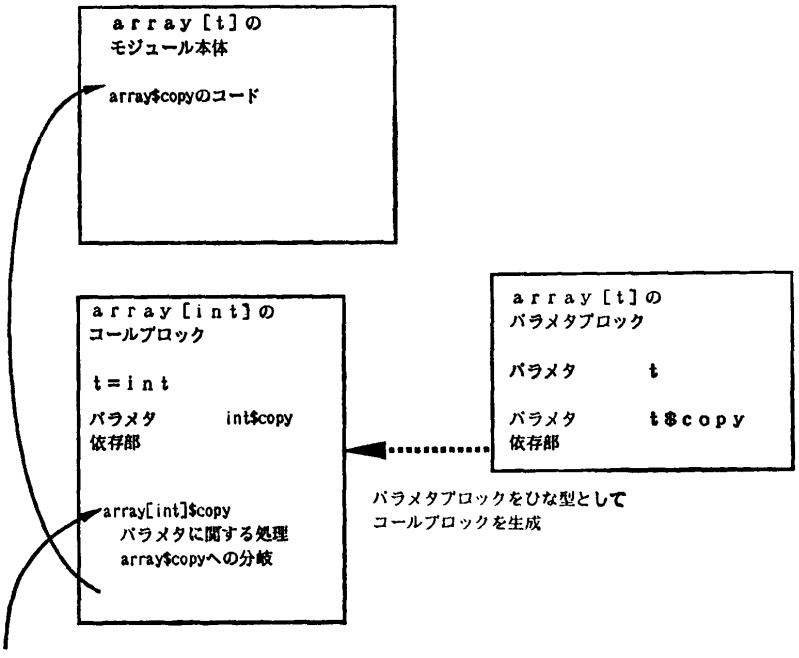

図 6 パラメタ付きモジュールの扱い
Fig. 6 Handling of the parametrized modules.

- run—実行中であるか，または実行キュー中にあって実行されるのを待っている状態
- wait—事象が起こるのを待ち合わせている状態
- suspend—他のプロセスによって実行キューから外されている状態
- stop—実行を終了した状態

プロセス型にはこれらの状態を遷移させるための操作群が備えられている．

CLU マシンには，システム内のプロセスを管理するためプロセスマネージャ型という型が用意されている．この型のオブジェクトはシステム内に一つだけ存在し，システム内のプロセスを登録してあるプロセステーブル，実行待のプロセスのためのキューなどを管理している．プロセスの切り替えは，タイマ割り込みまたは実行中のプロセスの実行権放棄により起こる．CLU マシンでは全プロセスが同一メモリ空間上に存在しているため，オーバヘッドの少ないプロセス切り替えが可能となっている．

5.2 プロセス間通信

プロセス間通信および同期処理を行うための機能としてメイルボックスが用意されている．メイルボックスはパラメタ付きモジュールであり，例えば_mailbox [char] は文字型，_mailbox [int] は整数型のデータをメッセージとして受け渡す．

通信はプロセス間で共有されたメイルボックスを介してメッセージをやり取りすることにより行われる．メイルボックスはプロセスを生成するとき，その引数として渡すことができるようになっている．

空のメイルボックスに対して receive 操作を実行するとそのプロセスは wait 状態になる．逆にメイルボックスに対して send 操作を行うと，メッセージの到着を待っていたプロセスのうち一つが run 状態になり，メッセージを受け取る．

5.3 割り込み

CLU マシンの中で発生する割り込みには，タイマ，ディスク，マウス，キーボード，シリアルポートなどのデバイスによるもの，および内部割り込みがある．割り込み処理は割り込み専用のスタックを用いて行われる．割り込みは四つの優先度に分けられており，高い優先度の割り込み処理中には低い優先度の割り込みは待たされる．

割り込み処理中には，プロセス切り替えが起こる可能性のある操作（領域割り当てなど）は禁止されている．さらに，データ操作時に排他制御が必要な場合には，プロセスの優先度を一時的に高くすることによって他のプロセスや割り込み処理に切り替わることを防いでいる．

6. その他のサブシステム

本章では前章までで説明した中核部分上で動作するサブシステム中，主要なものについて簡潔に説明する．

6.1 ファイルシステム

ファイルシステムの機能は Unix の場合と同様にファイルとディレクトリから成る階層構造とその中での恒久的データ記憶を提供することであるが，実現に際しては CLU のデータ抽象機能を生かした設計を採用している．基本概念としてディスクブロックを他のブロックへのポインタを含むものの2種に抽象化し，その上のデータ表現としてファイルシステムを実現している．このようにすることで，ディレクトリとファイルの階層構造を素直に表現できる．

フロックの実体はディスク上，またはメモリ内のバッファ上にあり，すべてのブロックは根ブロックからたどることができる．ポインタをたどることは「ポインタを含むブロック型」の fetch 操作を呼ぶことに相当し，必要ならこの操作がブロックをディスクから主記憶に読み出す．これにより上のレベルからはブロックが主記憶にあるか否かを気にする必要がなくなる．

6.2 ネットワーク機能

CLU マシンのようなシステムでは他のマシンとデータを交換する機能が不可欠であるが，現在使用中のハードウェアではシリアルポートのみが利用可能であったため，これに基づく通信機能を実装している．その最下レベルにはシリアルポートドライバが存在し，9600 bps 双方向の通信をサポートしている．

その上位レベルとしては端末エミュレータと UDP/IP（ARPA ネットワークの Internet Protocol およびその上位レイヤとして稼働する User Datagram Protocol）の2種類を実現している．端末エミュレータは Vax などの計算機に無手順端末として接続するのに使用され，そのうえで独自のプロトコルによるファイル転送をサポートしている．UDP/IP モジュールは，Vax および Sun の Unix に備わっているシリアルライン IP と相互接続可能である．

6.3 ウィンドシステム

本システム開発の一つの目的はビットマップ／マウスを活用したユーザインタフェースについて様々な試

みを行うことにある．その最初の段階として，本システム上で稼働するウィンドシステムを開発した．構造としては X-Window などと同様の画面サーバ方式を採用し，アプリケーションはメイルボックスを通じてサーバに画面操作を依頼し，またマウスやキーボードのイベントを受け取るようになっている．CLU マシンではプロセスの切り替えが軽く，プロセス間で構造を持つ大きなデータを渡すときもコピーの必要が無いため，他の Unix マシン上のウィンドシステムと比較してサーバ方式に対する適合性は高いといえる．

開発の初期の段階ではプログラムの起動には通常のコマンドインタプリタを使用していたが，このウィンドシステムが完成した段階でマウスとビットマップを活用した「アイコンを重ねる」方式による新しいユーザインタフェースを設計し実現している[2]．

7. システムの大きさと開発環境

本システムの中核部分の大きさを表 1 に示す．本システムは基本的に CLU 言語により記述されているが，入出力，ランタイムルーチンなどの低レベル部分については 8086 アセブリ言語を使用している．ただしアセンブリ言語では CLU で記述できないデータアクセス等のみを行い，論理の制御は CLU レベルで記述するようにした．上記のほかにコンパイラ (8,450 行)，ウィンドシステム (3,850 行)，その他のユティリティも開発したが，これらはほとんど CLU で記述されている．

CLU 言語については現在本システムの上でセルフコンパイラが稼働しているが，開発用には主記憶容量やファイル容量の関係から Vax または Sun 上の Unix で動作するクロスコンパイラを使用している．これは MIT で開発された CLU コンパイラのコード生成部分を書き換えたものである．アセンブラもやはり Unix 上で動作するが，これは新たに開発した．これらが出力するコードにはダイナミックリンクのための情報やオブジェクトヘッダの情報も含まれている．これらのコードファイルはシリアル回線経由で本システムに転送される．

表 1 中核部分の大きさ
Table 1 Sizes of the kernel modules.

モジュール	ソース行数	バイト数
ランタイムルーチン	4,781 (2,972)	71,554
入出力ドライバ	4,562 (3,691)	40,136
記憶域管理	1,408 (105)	39,124
プロセス管理	1,895 (614)	30,604
ダイナミックリンカ	1,596 (167)	43,330
ファイルシステム	2,784 (115)	87,312
その他	676 (161)	17,168
総計	17,702 (7,825)	329,228

かっこ内はアセンブラ行数

システムを構成する各モジュールのうち，システム立ち上げ時から存在していなければならないものの集まりを初期リンクセットと呼ぶ．記憶域管理，プロセス管理，ダイナミックリンカの各モジュールなどがこれに含まれる．これらを組み込んで初期メモリイメージを組み立て，boot ディスクを作成するのは現在の所 MS-DOS 上で行っている．

boot ディスクからシステムが立ち上がった後は，各モジュールのコードファイルはダイナミックリンカにより自動的に取り込まれて実行される．ダイナミックリンカは呼び出そうとしたモジュールと同一名を持つファイルをサーチパスから探してきてそこからコードをロードし，参照を解決する．したがって，初期リンクセット外のモジュールの実行は端末エミュレータによりコードを転送した後，それを名前で呼び出すことで手軽に行える．

現在，本システムは主記憶 640 KB の PC-98XA および主記憶 768KB の PC-98XL 上で稼働している．初期リンクセットおよびシステムテーブルの大きさは合計約 300 KB であり，残りの部分が利用者のコードおよびデータを格納するのに利用できる．利用者プログラムの大きさについては，個別のモジュール，配列，レコードなどをそれぞれオブジェクトとして管理するため，一つのモジュールや配列の大きさは最大 64 KB であるが，全体としての大きさは記憶領域が不足しない限り特に制約はない．

8. まとめ

本システムではコンパイラ，エディタ，端末エミュレータ，ウィンドシステム，およびシステムやファイルの状態表示／操作のためのツールが稼働しており，プログラムの自立開発が行える最低限の環境が整った段階である．

CLU 言語を採用したことは，抽象データ機能を活用してシステムのモジュラリティを向上させ，開発を行う上で大きなメリットとなった．また，システム自身の記述量もコンパクトにできた．

現在のシステム全体の性能は，簡単なテストプログラムを走らせた場合で Vax-11/750 の CLU 処理系

の半分程度となっており，満足できる状態ではない．今後コンパイラのコードの見直し，頻繁に呼び出されるコードのインライン展開，中核部分のオーバヘッドの削減等のチューニングを進めて行く予定である．また，これらと並行してファイルシステム，ネットワーク，ウィンドシステム／ユーザインタフェースの各機能を増強し，本システムの特徴を生かした図形エディタ，文書作成系などの上位アプリケーションの開発も進めて行きたいと考えている．

謝辞 東京工業大学の木村 泉教授にはシステムの構成や開発方法などに関して有益な示唆をいただいた．また，荒井俊史君（現日立製作所）の記憶域管理方式についてのアイデア，歴本純一君（現日本電気）とのシステム設計全般に渡る討論は貴重であった．よって，これらの方々に感謝します．

参 考 文 献

1) 久野 ほか：CLU マシンシステムの開発，情報処理学会オペレーティングシステム研究会資料，33-4 (1986)．
2) 佐藤，久野ほか：CLU マシンのユーザーインターフェース，第29回プログラミングシンポジウム報告集，pp. 13-22 (1988)．
3) Liskov, B. et al.: *CLU Reference Manual*, Springer, Berlin (1981)．
4) Unger, D.: Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *ACM Symposium on Practical Software Development Environments*, pp. 157-167 (1984)．

(昭和63年2月8日受付)
(昭和63年9月5日採録)

佐藤 直樹（正会員）
昭和37年生．昭和61年東京工業大学理学部情報科学科卒業．昭和63年同大学院修士課程情報科学専攻修了．同年日本電気(株)に入社．現在OS の開発に従事．

鈴木 友峰（正会員）
昭和38年生．昭和61年東京工業大学理学部情報科学科卒業．昭和63年同大学大学院修士課程修了．同年(株)日立製作所入社．コンパイラ，マンマシンインタフェースに興味を持つ．

中村 秀男（正会員）
昭和38年生．昭和61年東京工業大学理学部情報科学科卒業．昭和63年同大学大学院情報科学専攻修士課程修了．同年日本電気(株)に入社．CLU マシンの開発，コンパイラ生成系の研究などをへて，入社後言語処理系の開発に従事．オペレーティング・システム，コンパイラ，高級言語マシンなどに興味を持つ．ACM 会員．

二瓶 勝敏（正会員）
1964年生．1987年東京工業大学理学部情報科学科卒業．現在同大学院に在学中．オペレーティングシステム，分散環境に興味を持つ．

明石 修（正会員）
1964年生．1987年東京工業大学理学部情報科学科卒業．現在 同大