オペレーティング・システム 33－4
（1986. 12. 12）

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
（東京工業大学理学部）

強い型付けとデータ抽象機能を持つ言語ＣＬＵを使用して、高機能ワークステーション向けＯＳを開発中である。本システムは高級言語により記述された、コンパクトで軽いＯＳを目指している。その特徴としては

1. システムはＣＬＵ言語のデータオブジェクトの集まりであり、プロセス、スタック、コード等まですべて統一的に管理される。
2. ガベージコレクタを含む記憶域管理部までＣＬＵで記述され、またガベージコレクタは一つのプロセスとして一般のプロセスと並行して動作する。
3. ダイナミックリンクを採用することにより、モジュールのコードがプロセス間で共有でき、柔軟な管理が行える。
4. プロセスはメッセージチャネルを通して任意のデータオブジェクトをやり取りすることができ、プロセス間でオブジェクトを共有することも自由である。

などがある。本システムはVax/Unix上でクロス開発され、PC-98XA上で中核部分が稼働中である。現在は自立開発系を作成中であり、またファイルシステムが設計段階にあるが、今後はウィンドウシステム、図形エディタ等上位のモジュールを開発して行きたい。

Development of CLU Machine System

Yasushi KUNO, Naoki SATO, Tomomi SUZUKI, Hideo NAKAMURA, Katsutoshi NIHEI, Osamu AKASHI
(Tokyo Institute of Technology)

We are currently developing an operating system using CLU. Our objective was to create a compact and light-weighted operating system, totally described in a high-level language. Its distinguishing points are:

1. The system is a collection of CLU data objects, and the whole processes, stacks, codes, and other system resources are uniformly managed as data objects.
2. The memory manager, including the garbage collector (GC), is also described in CLU, and the GC process runs in parallel with other user processes.
3. Dynamic linking scheme is employed, so that all processes can share the same code object for each module, and flexible management of code objects become possible.
4. Processes can pass arbitrary objects to an another process via a mailbox object, and once passed, objects can be shared among those processes freely.

The system is cross-developed on a Vax-11/780 running Unix 4.3bsd, and run on a PC-98XA microcomputer system. Kernel modules are already working; the self development system and the file system modules are currently under development.

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はその殆どが中・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ような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[Lisk81]を用いて記述・開発することを計画した。本システムはＣＬＵマシンシステムと呼ばれているが、Lispマシン等とは異なり、純粋にソフトウェアだけから成る。

本システムの基本設計は1985年に開始され、現在PC-98XA上で中核部分が動作している。以下第１節で本システムの設計目標、特徴について述べ、２、３、４の各節でシステムの中核部分である記憶域管理、モジュール管理、プロセス管理の各部について説明する。続く第５節では、現在設計中のファイルシステムについて簡単に触れ、第６節では本システムの開発環境について述べる。最後に第７節で現状の説明とまとめを行う。

１．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最初に述べたように、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。

以上のような目標を達成する上で、任意の言語から機械語に翻訳したプログラムを実行させる通常のＯＳでは記述量が多くて大変であると考えたため、特定の言語を設定して、その言語の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することにした。具体的には言語としてＣＬＵを採用した訳だが、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考えたとともに、ＣＬＵのような高級言語でどのくらいうまくＯＳが記述できるか試してみたかった、という意味合いもあった。

次に、本システムの特徴としては次のような点を挙げることができる。

1. 単一言語系の考え方を採用した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た。
2. 日本語の使用を前提として、文字型データはすべて16ビットとした。これによりシステム内では英字と日本語を区別なく使用することができる。
3. ＣＬＵのデータ抽象機能を生かすことにより、システム全体を構造化し、理解しやすく拡張しやすいものとすると共に、移植性を高めることができた。
4. 記憶域管理／ごみ集めをシステムが一括して受け持つため、プログラマがこれらの事柄について気を使わずに済む。
5. ごみ集めも通常のプロセスとして動作するため、CPUのあき時間を有効に利用することができ、またシステムが停止する時間を短くできる。
6. ダイナミックリンクを採用することにより、プログラマはリンクの完了を待たずに実行に移れる。また、プロセス間で自然にモジュールのコードを共有できる。
7. プロセスがオブジェクトを自由に共有できることにより、プロセス間の密接なやりとりが可能になった。

２．ＣＬＵマシンの世界モデルと記憶域管理

ＣＬＵマシンの世界は、「全てのプロセスに共有される一つの大きな空間」から成り、この中にはＣＬＵのオブジェクトが浮かんでいる。また、生きているオブジェクトは、世界の根っこ（_world型の実体）からたどることができる（図１）。この様にすることにより、全ての実体をプロセス間で共有でき、また記憶域を統一的に管理することができる。本節ではこの記憶域管理部について説明する。

２．１　オブジェクトの構造

ＣＬＵマシンでは整数型や文字型など（これらは16

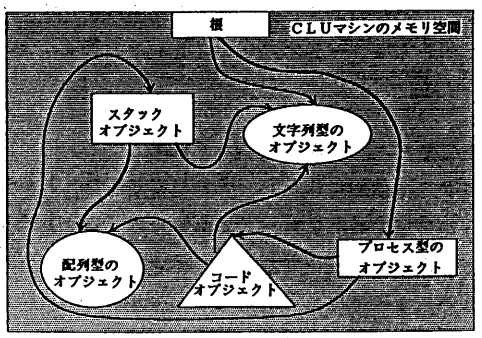

図１　ＣＬＵマシンのメモリモデル

ビットの直値として扱われ、直接レジスタ／スタック上に置かれる）を除くすべてのデータはオブジェクトとして統一的に扱われる。

すべてのオブジェクトは記憶域管理部によって割り当てられ、その先頭部分にはそのオブジェクトの管理情報が入っている（図２）。スタック（_stack）およびコード（_code）の２種類のオブジェクトについては、さらに特別な情報を持ち、これらの情報はプロセス／モジュール管理部によって操作される。

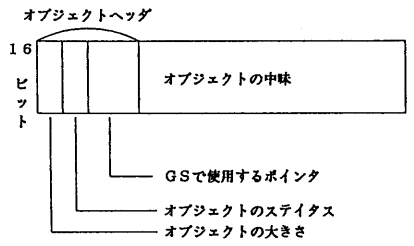

図２　オブジェクトの構造

これら以外のオブジェクト（プログラムが通常扱うデータオブジェクト）は、その中に他のオブジェクトへのポインタが格納されるベクタ型（_avec）と、直値が格納されるベクタ型（_wvec）の２種類をもとに作られている（配列やレコードは_avec、文字列は_wvecを内部表現に持つ）。

これにより、上位のモジュールでは様々な型を使用していても、記憶域管理部では上述の４種類のオブジェクトのみが存在すると考えてオブジェクトを管理すれば良く、その構造はかなりに簡単にできた。

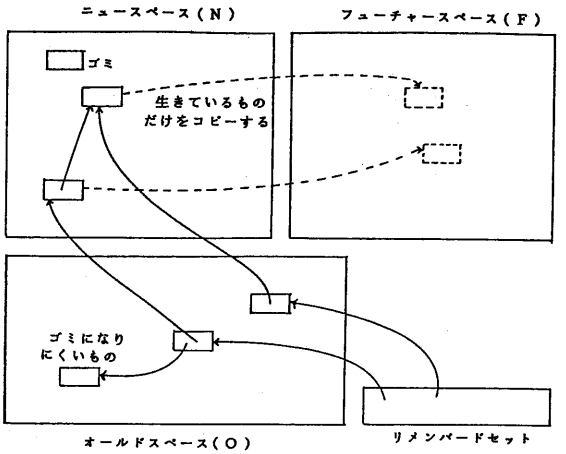

図３　メモリ空間の構成

２．２　ごみ集めの方式

使われなくなった記憶領域の再生（ごみ集め）をおこなうことも記憶域管理部の主要な仕事である。その手法としてはジェネレーションスカベンジング（GS）[Ung84]を用いているが、これはGSによればごみ集めと同時に領域の詰合せも可能であり、またシステムが止まってしまう時間も短くてすむことによる。

GSではメモリ空間を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（図３）。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のゴミ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時に１とし、ＮからＦにコピーされるたびに１づつふやされ、ある一定値を越えたもの（長生きしたもの）はＯにコピーされる。このようにすれば、Ｏにはゴミ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、さらに効率化のためリメンバードセットというテーブルを用意し、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しておく。

２．３　並列ジェネレーションスカベンジング

ＣＬＵマシンでは、ごみ集めも１つのプロセスであり、記憶域を消費するプロセス群と並行に走る。これにより、GCが他の処理を止めてしまうことが少なくなり、またCPUのあき時間を有効に使うことができる。その様にした場合に留意すべき点として次の２つがあげられる。

(1) GC中に他のプロセスが領域割り当てを要求したときどう扱うか。

(2) GCプロセスと他プロセスのオブジェクトへの並行アクセスをどう調整するか。

このうち(1)に関してはGCの途中ではＮはいっぱいであるが、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から割り当てることで対処する。

(2)に関しては更に、(a)他プロセスが、GCによってすでに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しようとした場合、(b)GCがたどり終ったポインタを、他プロセスが別の値に変更してしまう場合、に分けることができる。

これに対処するためGCプロセスはオブジェクトをコピーする際、コピーされたことを示すフラグと、コピー先を指すポインタをオブジェクトヘッダ部分に格納する。(a)では他プロセスはアクセスしようとしているオブジェクトのヘッダを見ることによりコピーされていることを発見し、正しい（コピー先の）オブジェクトをアクセスする。(b)に対しては、GCがＮをたどることを複数回繰り返し、最後の１回のみプロセス切り替えを禁止して行うことで対処する。最後にはほとんどのオブジェクトは処理済みであるので、制御を独占する時間は短いと考えられる。

２．４　オールドスペースのＧＣ

Ｏにはゴミになりにくいものが配置されているので、GCの頻度は少なくてすむ。その方式は現在検討中であるが、[Yua85]に報告されているアルゴリズムの使用を考慮している。

３．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とられ記憶域管理の対象となる（このため全コードは再配置可能に作られている）。本節ではコードオブジェクトとその管理について説明する。

３．１　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よって統一的に管理されている（図４）。

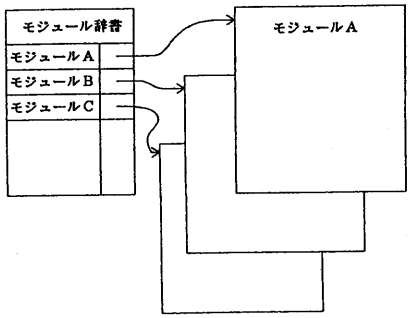

図４　モジュール辞書

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回収しなければならない。単純なモジュール間の呼び出しだけであれば、それに対応したリファレンスカウントを用いてコードが不要になったかどうか判定できる。しかしＣＬＵはproctype型（Ｃ言語での関数へのポインタに相当）を持つため、動的に変化する参照関係が存在する。そこで、全プロセスのスタックをたどって呼び出し中のコードをマーキングし、不要になったコードを解放する方法を採用した。

３．２　ダイナミックリンカ

先に述べたように、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。その際モジュールがモジュール辞書になければモジュールをロードして辞書に登録する。辞書にあればそのコードが使われるので、これにより全プロセスで自然にコードを共有することができる。

ダイナミックリンクの実現のため、モジュール間の

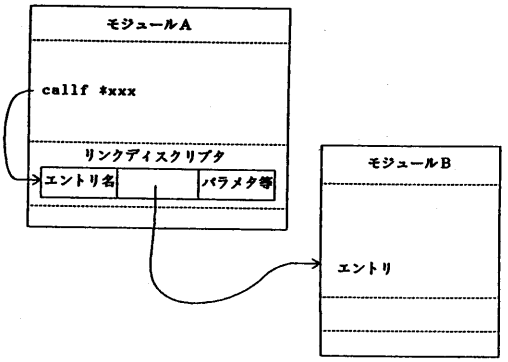

図５　ダイナミックリンカとリンクディスクリプタ

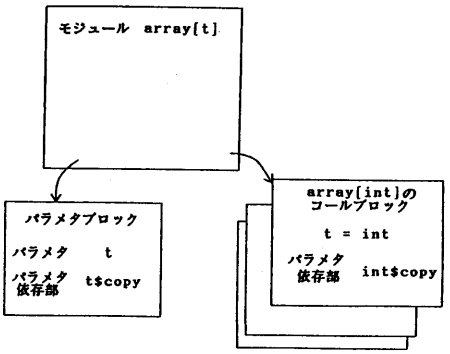

図６　パラメタライズドモジュール

呼び出しは必ずリンクディスクリプタを介して間接的に行うことにした。ディスクリプタはコードオブジェクトに付随していて、他モジュールへの参照ごとに作られる（図５）。

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て、呼び出しが起こるとリンカに制御が移る。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ディスクリプタのありかを割り出し、参照を解決して正しい呼び出し先へ分岐する。２回目からは呼び出し番地が既に正しい呼び出し先を指しているので、ほとんどオーバーヘッドなしで呼び出しが実行できる。

３．３　パラメタつきモジュールの実現

単純な呼び出しは以上のようにして実現できるが、さらにＣＬＵ言語ではモジュールがパラメタを持つことができる（AdaのGenericパラメタに相当）。たとえばarrayというモジュールにintという型を与えることでarray[int]（整数型の配列）を具体化することができる。この場合、arrayモジュール中にパラメタ型の操作呼び出しが書かれていれば、それはint型への呼び出しとして実行されなければならない。

パラメタの処理はまずパラメタのついたモジュールをロードしたときに可変部の情報をまとめてパラメタブロックという構造を作ることから始まる。実際にパラメタを伴う呼び出しが起こったときには、パラメタブロックにパラメタをあてはめてコールブロックという構造を作る。これは与えられたパラメタごとに作られる仮のエントリと、パラメタに依存した呼び出しに対応するリンクディスクリプタの集まりである。コールブロックが呼び出されると、仮のエントリでパラメタごとの前処理をしてから本体のエントリに分岐して処理をおこなう（図６）。

４．プロセスとプロセス間通信

ＣＬＵマシンではオブジェクト単位の並列実行ではなく、複数のプロセスが並行して走るプロセスモデルを採用している。これは、オブジェクト単位の並列実行では並列実行単位が小さすぎて効率よい実現が難しいと考えたからである。本節ではＣＬＵマシンのプロセスとプロセス間通信について述べる。

４．１　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ＣＬＵの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。

run：　プロセッサ上でプログラムを実行しているか、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っている状態

suspend：他のプロセスによって実行キューからはずされている状態

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

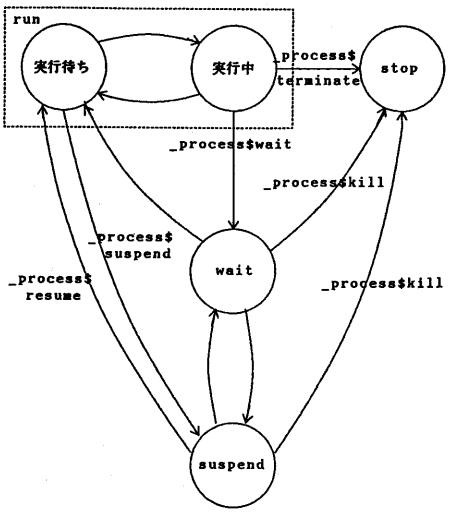

図７　プロセスの状態遷移

群が備えられている（図７）。

ＣＬＵマシンには、システム内のプロセスを管理するために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ランキューなどを管理している（図８）。

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って起こる。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、オーバーヘッドの少ないプロセス切り替えが可能となっている。

４．２　プロセス間通信

プロセス間通信を行うためのデータ型としてメイルボックス型が用意されている。メイルボックスはパラメタつきのモジュールであり、例えば _mailbox[char] は文字型，_mailbox[int] は整数型のメッセージを扱うことができる。

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状態になり、他のプロセスによってそのメイルボックスへメッセージがsendされることによりrun状態にもどされる。

通信はプロセス間で共有されたメイルボックスを介

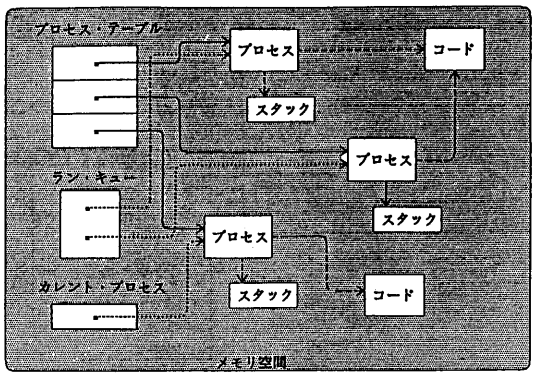

図８　メモリ空間中のプロセスとその管理

してメッセージを送受することにより行われる。メイルボックス型のオブジェクトはプロセスを生成する際に引き数として渡され、プロセス間で共有されることになる。

また、同期や相互排斥もメイルボックスを使うことにより実現することができる。

４．３　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、キーボードなどによるもの、および内部割り込みがある。割り込み処理によりユーザプロセスのスタックがあふれるのを防ぐため、割り込み処理は割り込み専用のスタックを使って行われる。割り込みは種類ごとに４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要求は待たされる。

また、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ぐことができる。

５．　ファイルシステム

ファイルシステムは現在設計中であるが、ＣＬＵのデータ抽象機能を生かした設計を目指している。基本設計概念としては、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（ポインタブロック）とデータを含むもの（データブロック）の２種に抽象化し、

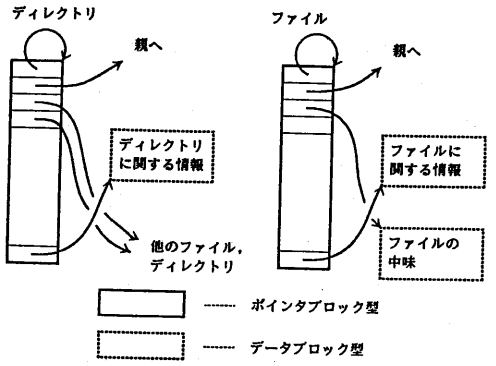

図９　ディレクトリとファイルの構造

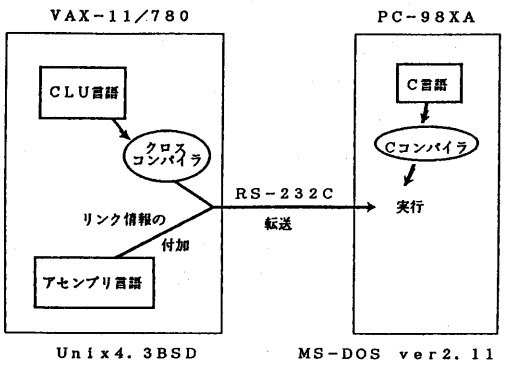

図１０　開発系

その上のデータ表現としてファイルシステムを考える。この様に抽象化することによって、ディレクトリ・ファイルの階層構造を素直に表現することができる（図９）。

ブロックの実体はディスク上、またはメモリ上のバッファにありすべてのブロックはルート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メモリに読みこむ。これにより上のレベルからはブロックがディスクにあるか否かを気にする必要が無くなる。

入出力用のバッファの数やその管理の方法については、現在検討中である。

６．ＣＬＵマシンのシステムの開発

本システムの開発は、ＣＬＵ言語、8086アセンブリ言語、およびＣ言語を使用して行っている。ＣＬＵ言語に関しては現在はVAX/Unix上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を、8086のアセンブリコードを出力するように書き変えたものである。また、ここではコード生成の他に、ダイナミックリンクのための情報の出力も行っている。クロスコンパイラによって出力されたコードは、VAXからPC-98XAへRS-232C回線を通して転送され、実行されている。またアセンブリ言語のプログラムは、ＣＬＵ言語と同様にVAX上で開発され、リンク情報を付加するプログラムを通し、転送し実行される。

Ｃ言語のコンパイルには、MS-DOS上のＣコンパイラを使用しており、開発もMS-DOS上で行っている。このＣ言語で書かれている部分は、主にシステムの立ち上げの部分であり、将来的には無くなる予定である。

現在のコードの記述量を、表１に示す。ＣＬＵ言語の性質上、ＯＳの中核部は、かなりコンパクトに記述できた。また、移植性を高めるため、現在のアセンブリ言語で記述された部分は、できるだけＣＬＵ言語で書くようにし、将来的には最小限（割り込みのハンドラや、主記憶の管理など）の記述量にしたい。

	ＣＬＵ言語	アセンブリ言語
記憶域管理	約　６００　行	約 １１００　行
モジュール管理	１４００	１００
プロセス管理	１１００	５００
ランタイムサポート	１０００	３００
クロスコンパイラ	１３０００	０
合計	１７１００	２０００

表１　コード量

７．現状とまとめ

現在、本システムはVAX-11/780上でクロス開発され、PC-98XA上で中核部分（プロセス管理、記憶域管理、モジュール管理）が動作中である。ＣＬＵを採用したことは、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。

また、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上のＣＬＵの処理系と同程度となっているが、全体としてはまだVAX-11/750にはおよばない部分が多い。しかし、まだコンパイラの出力するコードに対して十分な最適化を行っていないので、将来的にはより高速なシステムができるものと思われる。

今後は本システム上で自立開発が行えるようにファイルシステム、コンパイラ等を整備するとともに、ウィンドウシステム、文書エディタ、図形エディタ等の上位のユーティリティの開発を進めていく予定である。

参考文献

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

[Ung84] David Unger, Generation Scavenging: A Non-disruptive High Performance Strage Reclamation Algorithm, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, April 23-25 1984.

[Yua85]Taichi Yuasa, Realtime Garbage Collection on General-purpose Machines, 日本ソフトウェア科学会第２回大会論文集, pp.181-184, 1986.