システムソフトウェアと
オペレーティング・システム 69－6
（１９９５．６．２２）

２次元アドレスとダイナミックリンクのための実行コンテキストの設計と言語Ｃ処理系の開発

並木美太郎[1], 中村浩之[2], 田中広幸[3],
森永智之[1], 加藤泰志[1], 早川栄一[1], 高橋延匡[1]

1. 東京農工大学　工学部　電子情報工学科　コンピュータサイエンスコース
2. (株)日立製作所　　3. (株)ＮＥＣ
E-mail: namiki@cc.tuat.ac.jp

マルチメディアデータの本質は、データの多義性・多様性にある。これら多義性・多様性を持つデータを処理する基盤として、筆者らは「電紙」と呼ぶデータモデルを提案し、OS/omicron第４版と呼ぶシステムに構築している。OS/omicron第４版の実行機構においては、「電紙」を２次元アドレスにより表現されたセグメントとして実現し、ダイナミックリンクによりデータや手続きの実行時結合を可能にした。２次元アドレスやダイナミックリンクをシステムのソフトウェアアーキテクチャとして採用した場合、システムを記述する言語の言語処理系は、アドレシング機構や名前の管理情報をOSのために生成することが必要不可欠となる。

本稿では、言語Ｃにおいて２次元アドレスとダイナミックリンクを実現するための実行コンテキストの設計と、インテル80X86用の言語処理系の実現・評価について述べる。本設計における実行コンテキストは、マルチタスクでのセグメント共有やスレッドライブラリ作成に有効となるよう、コンパイル単位ごとに複数のセグメントを持ち、リロケータブルかつリエンタラブルとなるようにした。また、インテル80X86のようにリンケージフォールトを持たないCPUでダイナミックリンクを実現する手法について論じる。

Design of OS's context and implementation of a language C compiler for 2D addressing and dynamic linking

Mitarou Namiki[1], Hiroyuki Nakamura[2], Hiroyuki Tanaka[3], Tomoyuki Morinaga[1],
Yasushi Katoh[1], Eiichi Hayakawa[1], Nobumasa Takahashi[1]

1. Tokyo University of Agriculture and Technology 2. Hitachi Ltd. 3. NEC Ltd.

The essence of multimedia is in the polysemy and diversity of data. We propose a data model called "DENSHI" (Virtual Paper) as a base for processing data that has these qualities of polysemy and diversity, and are designing a system called "OS/omicron V4". In the execution structure of OS/omicron V4, this DENSHI is implemented as a segment represented by two-dimensional addressing, and a dynamic link makes it possible to combine data and procedures at the time of execution. When using two-dimensional addressing or dynamic linking as the system's software architecture, it is essential that the language processor for programming the software system, for the OS, the addressing structures and management information for names.

In this paper we describe the design of a context for implementing two-dimensional addressing and dynamic linking in the C programming language, as well as giving details on the realization and evaluation of a language processing system for the Intel 80X86. The context for this design has been constructed, such that it is effective for segment sharing and creating thread libraries in multi-tasking, with multiple segments for each compiler and also so that it is relocatable as well as being reenterable. We also discuss a method of producing dynamic links in a CPU that does not have linkage-fault, such as the Intel 80X86.

1．はじめに

筆者らは、手書きやマルチメディアデータを処理するための基盤として、OS/omicron第4版というシステムを研究している。多義・多様な性質を持つ手書きやマルチメディアデータを管理するために、OS/omicron第4版では、実現機構として、2次元アドレス・ダイナミックリンクを採用している。特に、OS/omicron第4版では、言語処理系レベルではなく、OSレベルからこの機構を提供することで、実行効率や保護を損ねることなく、ダイナミックリンクを実現している。

このようなシステムを開発するためには、2次元アドレス・ダイナミックリンクに対応した言語処理系が必要不可欠となる。特に、ソフトウェアシステム全体のアーキテクチャを一貫させるためには、言語処理系によりシステム内で標準的な実行コンテキスト・実行環境を生成する必要がある。

かって、2次元アドレス・ダイナミックリンクのためのコードを生成する言語処理系として、MulticsにおけるEPL[1]、HITAC5020におけるPL/IW[2]が存在した。これらのいずれもPL/I言語である。OS/omicron第4版では、システム記述言語して言語Cを採用した。言語CはPL/Iと異なる変数スコープを持つため、新たに実行コンテキストと実行環境を考察した。さらに、現代のセグメンテーションを持つインテル80X86(386/486/Pentium)などのCPUアーキテクチャでは、リンケージフォールト割込みを持たないことから、ダイナミックリンク下におけるリンケージフォールト処理を何らかの方法で実現する必要がある。

本報告では、2次元アドレス・ダイナミックリンクのオブジェクトコードを生成する言語C処理系の実行コンテキストと実行環境および処理系の実現と評価について述べる。

2．OS/omicron第4版と実行環境の設計目標

OS/omicron第4版は、手書きユーザインタフェースを用いたPDA構築を目的に設計されている。その設計思想は、次のとおりである[3,4]。

(1) 手書きやマルチメディアデータの多義性・多様性に対処するための枠組を提供する。「電紙」と呼ばれる資源モデルを採用し、手書きデータさらにはマルチメディアデータの持つ多様性・多義性をシステム内で統一的に管理する。
(2) 2次元アドレスを持つセグメントを資源管理の単位とする。特に、「電紙＝セグメント」とすることで資源を統一的に扱う。
(3) 名前とオブジェクトを柔軟に結合するためにダイナミックリンクを行う。多様性・多義性を持つデータを処理するためには、型とデータや手続きの対応を柔軟に関係付ける必要がある。そのため、静的なリンクより遅延評価のように実行時に結合関係を与えることのできるダイナミックリンク方式を採用した。
(4) ワンレベルストアとすることで、データ間の関係の表現を容易にする。
(5) マイクロカーネル構成によりシステムを構築することで、多様な資源管理を実現する。
(6) ハードウェアアーキテクチャとしてはセグメンテーションの機構を持つCPUを対象とする。具体的にはインテル80X86(80386/486,Pentium)、PA-RISCを想定している。

ソフトウェアシステムは、OSの規定する実行コンテキストで、できること・できないことが決まる。言語処理系は、そのOS上で実行されるプログラムのためのコンテキストを生成する。資源管理部を動的に拡張する要求に対しては、すべての層のコンテキストが同じでないと一貫性を確保できなくなる。特に、マルチメディアシステムの構築では、拡張性を確保するために資源管理においてユーザレベルの処理を行う必要が出てくる。このような局面で例えば、ユーザ層とOS層で異なる処理系・コンテキストとすると、ユーザレベルの処理を実現するのは困難となる。

前記特徴を有するシステムアーキテクチャを一貫して構築するために、2次元アドレス・ダイナミックリンクが可能な実行コンテキスト(言語処理系から見た場合は実行環境)を定めた。OS/omicron第4版の記述言語として、システムの早期稼動を考慮して、言語Cを採用した。本稿では、言語Cを対象とした実行コンテキストの設計について述べる。なお、現在、筆者らの研究グループのメンバがC++の実行コンテキストを設計している[5]。

3．OS/omicron第4版の実行環境の設計

3．1　設計方針

実行環境の設計において、次の設計方針を定めた。

(1) システム全体の標準的なコンテキストとする。すべてのプログラムは、この実行環境をともなって実行されることで、ソフトウェアアーキテクチャの一貫性を確保する
(2) 2次元アドレスによって番地付けされたセグメントの実行や参照を可能にする
(3) ダイナミックリンクを実現する
(4) マルチタスク下で手続きやデータの共有、さらにスレッドの作成を容易に行えるようにするた

め、リロケータブルかつリエントラントなコードとする

3．2　2次元アドレスとポインタ

OS/omicron第4版では、2次元アドレスを持つセグメントを資源管理の基本単位としている。本処理系では、2次元アドレスを次のように定義している。

2次元アドレス＝

＜セグメントid,セグメント内オフセット＞

セグメントid,セグメント内オフセットは、それぞれ32ビットの計64ビットで構成されている。セグメントidとオフセットは独立しており、オフセットはセグメントidへ桁上がりをしない。インテル8086のようなエイリアシングは起きない。セグメントの共有はアドレス変換機構によって実現される。

言語Cにおけるポインタは、すべてこの64ビットの2次元アドレスとなっている。ポインタを2次元アドレスにすることにより、言語Cによりシステム全体のメモリ資源を参照できる。例えば、

```
int i, *p, array[32] ;
p = &i ;     /* &iは64ビット2次元アドレス */
p = array ; /* array も2次元アドレス */
```

となる。ある構造体の各メンバや配列の要素は、同一セグメントに割り付けられ、オフセットはセグメント内オフセットとなる。先ほどのarray[32]の各要素は同一セグメント内に格納される。

ポインタの加算は、セグメント内オフセットに作用する。例えば、“p+i”は、

＜pに格納されていたセグメントid,
pに格納されていたセグメント内オフセット＋
i*sizeof(*p)＞

となる。加算の定義からポインタの減算は、セグメント内オフセットの減算となり、異なるセグメント間のポインタの減算の結果は意味を持たない。

2次元アドレスから実アドレスへの変換およびセグメントの保護・共有は、各CPUの持っているアドレス変換機構によってなされる。インテル80X86では、セグメンテーション・ページングを用いて実アドレスへ変換される。また、80X86ではセグメントidは16ビットであるが、2次元アドレスとしては0をパディングし32ビットとしてある。言語Cでよく利用されるNULLは、<0,0> と解釈し、メモリ管理でこのセグメントを未定義とすることにより、間違った参照を検出できるようにしてある。

3．3　コンパイル単位、変数属性、セグメント、プロセッサコンテキスト

言語Cでは、ソースファイル一つをコンパイル単位としている。コンパイル単位中の各関数・変数は、次のようにセグメントに割り当てられる。

(1) 外部・内部関数

コンパイル単位内のすべての関数は、コンパイル単位に一つの手続きセグメント(PS:Procedure Segment)に格納される。

(2) 外部変数

変数一つごとに一つのセグメントを割り当てる。このそれぞれのセグメントを外部変数セグメントと呼ぶ(EDS:External Data Segment)。これとワンレベルストアにより、例えばファイルを外部変数と結合させ配列として操作することが可能となる。

(3) 内部静的変数

コンパイル単位内のすべての内部静的変数は、コンパイル単位に一つの内部静的データセグメント(IDS:Internal Data Segment)に割り付けられる。

(4) 動的変数

コンパイル単位内のすべての動的変数は、タスクに一つのスタックセグメント(SS:StackSegment)に割り付けられる。

なお、このようにセグメントを割り当てることにより、OS/omicron第4版の実行コンテキスト、特にプロセッサに関するコンテキストは次のようになる。タスクは、複数のコンパイル単位からなり、複数のセグメントを実行実体として持つ。

例として、インテル80X86でのレジスタを表1に示す。

表1．プロセッサコンテキスト

内　容	80X86でのレジスタ
プログラムカウンタ	%cs,%ip
PSW(Program Status Word)	%psw
計算のための汎用レジスタ	%eax,%ebx,%ecx, %edx,%esi,%edi
スタック・フレームポインタ	%esp,%ebp
各コンパイル単位におけるIDS	%es
〃　LTS(後述)	%fs
タスクのSS	%ss
一時的なセグメントレジスタ	%ds

特に、PS・IDS・LTSはコンパイル単位ごとに割り当てられる。EDSのセグメントidは、リンケージテーブル中に格納され、参照時に一時的なセグメントレジスタに代入される。各セグメントは、ロード時に主記憶が割り付けられ、セグメントidが確定する。本方式を採用することにより、リロケータビリティ・リエンタラビリティを実現し、タスクやコン

パイル単位ごとに異なるインスタンスを持つことができる設計とした。

3．4　ダイナミックリンクとリンケージテーブル

OS/omicron第4版では、ダイナミックリンクにより実行時に名前とオブジェクトの結合関係を解決することで、手書きやマルチメディアデータの多様性・多義性を解決する。ダイナミックリンクをシステムの標準の機能とするために、言語処理系ではリンケージテーブルを出力する。

3．3節での言語Cの変数属性に対応したセグメントに加え、一つのコンパイル単位は一つのリンケージテーブルセグメント(LTS:Linkage Table Segment)が生成される。OS/omicron第4版では、ダイナミックリンクを行うが、リンク処理において各タスクごとで異なる結合を行えるようにするため、各コンパイル単位ごとにLTSを設けた。リンクと同時にロードが行われるが、このとき、SSを除く、PS・EDS・IDS・LTSは、実行時に各コンパイル単位ごとにそれぞれセグメントが割り付けられ、セグメントidが確定する。

LTSには、外部関数・外部変数のようにアドレス解決が必要なオブジェクトに関する情報が格納される。LTSの一項目の内容は次のとおりである。

・名前(文字列)
・変数または関数の定義されている2次元アドレス
・関数の場合は、その関数の属するコンパイル単位のIDSのセグメントid
・　〃　〃
LTSのセグメントid

言語処理系が設定するのは、名前と、定義の場合における2次元アドレスのオフセット部だけである。すべてのセグメントidは、ダイナミックリンカとローダが決定し、この項目に格納する。

インテル80X86における、外部変数の参照例を図1に示す。

なお、Multics、HITAC5020では、2次元アドレスを表すワード中にリンケージフォールトビットを設け、未解決のシンボルを検出できるようハードウェアアーキテクチャが設計されていた。現代のCPUでは、このようなアーキテクチャサポートはない。そこで、インテル80X86の実装では、次の方法でダイナミックリンキングを実現している。

(1) 言語処理系は、未解決のシンボルに対するリンケージテーブル中の2次元アドレスの値に、<不在セグメント例外となるid例えば7，オフセット0>を設定しておく
(2) アドレス変換側で常にこのセグメントidを不在セグメントとする
(3) (1)と(2)により、セグメントレジスタにこの値をロードしようとするとフォールト(不在セグメント例外)が発生する。図1の例では、下線部の命令で割込みが発生する(関数呼出しの場合はcall命令で発生する)
(4) フォールトを検出した後、ダイナミックリンカを実行し、割込み時のプログラムカウンタから命令の正当性を、<%ds,%eax>レジスタの値から未解決のリンケージテーブルを解析する
(5) ダイナミックリンカは、名前空間からオブジェクトを検索する
(6) 対応するオブジェクトが未ロードであれば、セグメントを割り付けてロードする
(7) リンケージテーブルに確定した2次元アドレスを書込み、命令を再実行する

3．5　関数の呼出し手順

前述のように、リロケータビリティ・リエンタラビリティを確保するために、コンパイル単位ごとにIDS・LTSを設定する。このため、コンパイル単位をまたぐ関数を呼び出す場合、セグメントidの値を退避し、呼び出される側のIDS・LTSを設定する必要がある。

```
言語C：                    80X86のコード：
extern int ei;             mov.d   %eax,LTS中のeiの項目のオフセット
ei = 1;                    lds     %eax,%fs:[%eax]
                                ↑eiの2次元アドレスを<DS,EAX>へ
                           mov.w %ds:[%eax],0x1
                                eiに対するLTSの内容
                           dd eiのセグメントid
                           dd eiのセグメント内オフセット
                           dstr "e1" ←識別子の文字列
```

図1．外部変数の参照例

本実行環境では、最適化のしやすさからcallee saveとし、IDS・LTSの待避と設定を呼び出された側で行うようにした。関数の呼出し手順を図2に示す。

3．6　初期化式中のポインタの扱い

言語Cには初期化式があり、その式中でアドレスを記述できる。静的変数のポインタ初期化、例えば、“int ei, *p = &ei;”では、何らかの方法により変数pへeiのアドレスを代入する必要がある。UNIXのような静的リンク、多重仮想空間のシステムでは、リンク時にアドレスが確定し、初期化データセクション中にそのアドレスを書いておけば、後はページングにより主記憶にデータをロードするだけですむ。

筆者らのシステムでは、アドレスは実行時まで確定しない。そのため、リンケージフォールト発生時、新たにコンパイル単位をインスタンシエートした時点で、ダイナミックリンカとローダが、すべての初期化式中のアドレスを決定し、データにパッチをあてる。

4．言語C処理系の開発

第3章で述べた実行環境を生成する言語Cコンパイラを、筆者らがM68000ファミリ用に開発した言語Cコンパイラ CAT(C compiler developed at Tokyo University of Agriculture & Technology)[6]を用いて開発した。CATは、マルチフェーズ分割により、複数のプログラミング言語・CPUアーキテクチャ向けの言語Cコンパイラの開発を効率よく行えることを目標に設計されている。また、CATも言語Cで記述されており、Earleyの方法によりセルフコンパイラを作成する。

```
caller
extern void g();        (1)必要なら返り値の領域をスタックに確保する
f() {                   (2)引数をスタックに積む
   g() ;          →     mov.d %eax,gのLTSのオフセット
}                       fcall %fs:[%eax]
                                ↑(3)呼出す関数のLTSのオフセットをレジスタに格納し、
                                call命令(セグメントをまたぐcall)を実行する。
                                間接呼出しとなっており、戻り番地として2次元
                                アドレスがスタックへ格納される。未解決時は、
                                この命令で割込みが発生する
                        (12)積んだ引数のバイト数、スタックを戻す
                                ……
                        gのリンケージテーブル
                        dd   gのセグメントid, gのセグメント内オフセット
                        dd   gの属するコンパイル単位のIDS, LTS のセグメントid

callee                  (4)関数gのエントリ。この時点で<%fs,%cax>によりcaller側の
                         gのリンケージテーブルを参照できる
g() {                   push.w %es
  …                     push.w %fs  ←(5)callerのIDS・LTSの待避
}                       move.w %es,%fs:[%eax+8]
                        move.w %fs,%fs:[%eax+12]
                                ↑(6)callerのリンケージテーブル中からcallee gの
                                IDS・LTSを設定する
                         (7)汎用レジスタの待避
                        enter 動的変数のバイト数,0
                        …(8)関数の処理。返り値は%eaxで返される
                        leave
                        (9)汎用レジスタの復帰
                        pop.w %fs
                        pop.w %es ←(10)callerのIDS・LTSの復帰
                        fret      (11)スタック上の<PS,retオフセット>へ戻る
```

図2．関数の呼出し手順

本研究では、このCATのパーザを再利用(27K行)し、新たにコード生成(27K行)・アセンブラ(10K行)を開発した。gccのコード生成を書換える方法も考えうるが、筆者らのシステムはフル2バイトコードなので、独自に開発した方が有利と判断した。

セルフコンパイラの開発にあたって、OSは開発中であり、実際に2次元アドレス・ダイナミックリンクを用いたプログラムの実行はできない。そこで、UNIXやMS-DOS上に2次元アドレス環境をエミュレートするイクステンダを作成し、セルフコンパイラおよびそれが出力するオブジェクトコードを実行することでOSの開発を進めている。

インクステンダ上で実行した本処理系(CAT X86)の生成したオブジェクトコードとgccのコードの性能比較を表2に示す。計算機は、DELL Optiplex 4100(80486 DX4)、OSはLinuxを用いた。

表2．CAT X86とgccのコード実行性能の比較

	CAT X86	gcc
`auto int i;  i = 1;`	20ns	20ns
`static int si; si = 1;`	20ns	20ns
`extern int ei; ei = 1;`	150ns	20ns
`auto int *pi;*pi = 1;`	140ns	30ns
関数呼出し `f() ;`	1010ns	160ns
`auto int *p1,*p2;` `p1 == p2;`	190ns	90ns

表2を見ると、CAT X86はポインタ演算や関数呼出しにおいて、gccの2倍から7倍程度の実行時間を要している。gccのポインタは4バイト、CAT X86のポインタは8バイトと、サイズが異なるので、完全に比較することはできないが、ポインタ関係については、セグメントレジスタに対する操作が80X86では遅いことがオーバヘッドの要因となっている。関数呼出しについてはセグメントの待避・設定、セグメント外call命令などが実行速度の低下を引き起している。

実行速度と柔軟性はトレードオフであるが、高速性が要求されるところは静的リンクを用いること、最適化によりセグメントレジスタの操作を減らしたりセグメント内関数呼出しの命令を生成する必要があろう。

いずれにせよ、言語Cの持つ変数属性・名前のスコープ規則はダイナミックリンクに不向きな点がある。また、インテル80X86は2次元アドレスを実現するのに不利なアーキテクチャとなっている。前者の問題に対しては、他の言語、例えばオブジェクト指向言語で対処して行きたい。後者については、ハードウェアアーキテクチャの改善が望まれる。

5．おわりに

本稿では、言語Cにおいて2次元アドレス・ダイナミックリンクを可能にするための、実行コンテキスト・実行環境の設計と言語処理系の開発について述べた。本研究により、OS/omicron第4版開発の基盤が整い、また、μカーネルの設計と実現が完了している。今後の課題は、これら環境上にファイルシステムやウィンドウシステムなどの資源管理部をOS/omicron第3版より移行し、OS/omicron第4版の実現を急ぎたい。

参考文献

[1] Organick, E.I.: The Multics System, MIT Press, 1972.

[2] PL/IW言語の仕様, 日立製作所 中央研究所, 1969.

[3] 早川他, 手書きインタフェースを支援するOS OS/omicron V4, 情報処理学会コンピュータシステムシンポジウム論文集, Vol.92 No.7, 1992.

[4] E.Hayakawa et al, Basic Design of SHOSHI Operating System That Supports HAndwriting Interfaces, 情報処理学会論文誌 Vol.35 No.12, 1994.

[5] 加藤他, : オブジェクト指向によるOS/omicron第4版の構築法, 情報処理学会コンピュータシステムシンポジウム論文集, 1994.

[6] 並木他, :OS/omicron用システム記述言語C処理系Catのソフトウェア工学的見地からの方式設計, 電子情報通信学会論文誌 D Vol.J71-D No.4 , 1988.