解　説

# メイン・メモリの論理構造*

武　井　欣　二**

## 1. まえがき

計算機の構造は複雑化する一方であり，その主要な構成要素であるメモリの構造にも新しい技術，新しい概念がつぎつぎに採り入れられ，ひとつひとつを追いかけることが難しくなってきている．しかし，このような複雑化は無秩序に進んでいるのではなく，これらの構造をいくつかの階層に分解して整理してみれば，それぞれの階層内部での構造変化は意外に単純なものであることに気付くはずである．たとえば，計算機のハードウェアからアーキテクチュアという階層が分離されたことによって，プログラマーは計算機の構造に対して以前にくらべはるかによい見通しをうることになった．アーキテクチュアはハードウェアからその論理的な性質を抽象したものである．このような抽象化を段階的に積み重ねて構造的な見通しをよくすることは，ソフトウェアにおいてはプログラミング手法として利用され始めてきているところで，Dijkstra のストラクチュアド・プログラミングとして広く知られるようになっている．

以下ではこの考え方にならって，メイン・メモリの論理構造を整理して，いわゆるフォン・ノイマン・モデルの基本構造を明らかにし，メモリ・アドレスの抽象化として論理アドレスの概念を導入する．さらに，この抽象化の裏としてアドレス変換を説明し，これを使って仮想記憶の基本構造について述べたあと，重要性をましてきている記憶保護の基本概念を明らかにする．

## 2. 基本構造

### 2.1 物理構造と論理構造

ハードウェアとして実現されているそのままの姿でとらえられたメイン・メモリの構造をその物理構造というのに対して，プログラムの作成に関係するメイン・メモリの機能を抽出してプログラマーの扱い易い姿に形式化して再構成された構造をその論理構造という．もう少しまとめたい方をすれば，メイン・メモリの論理構造とは，メイン・メモリのもつ論理的機能とそれらの機能の間に成立する関係のことである．

ここで注意すべきことは，物理構造というのがひとつの実在として一意的にとらえられるのに対して，論理構造の方はプログラマーとしてどのような立場をとるかによって様々な形式化が可能であることである．例えば，アセンブラ言語でプログラムを組むひとと，高級言語しか利用しないひととでは，メイン・メモリに対するイメージが違うものと思われる．少なくとも次のような違いがあることは確実である．アセンブラ言語においてはメイン・メモリに対する相補的な概念としてレジスタが存在しているが，高級言語では捨象されてしまっている．

この場合のように，ふたつの構造があって，一方の構造からそのいくつかの特徴を捨象することによって他方の構造がえられるとき，あとの構造をまえの構造の上位構造ということにする．この意味で高級言語レベルの論理構造は，アセンブラ言語レベルの論理構造の上位構造になっている．上位構造に対して双対的に下位構造が定義できる．上位構造における出来事はすべて下位構造における出来事として対応づけることができることは明らかであるが，このような構造間の対応づけをトランスレーションという．高級言語でのメイン・メモリの論理構造をアセンブラ言語でのものにトランスレートすることはコンパイラによって行なわれている．物理構造はすべての論理構造に対して下位構造となっていることはいうまでもない．そうでなければプログラムは正しく実行されなくなってしまう．論理構造からの物理構造へのトランスレーションにみられる特徴のひとつは，このトランスレーションの一部がハードウェアによって行なわれることがあるという点である．この具体例の代表的なものとしては，バッファ・メモリや仮想記憶などがある．

* The Logical Structure of Main Memory Systems by Kinji TAKEI (Tokyo Shibaura Electric Co. Computer Division).
** 東京芝浦電気（株）

### 2.2 アドレス機構

メイン・メモリの論理構造として最も基本的なものは，フォン・ノイマン・モデル，すなわち一定の記憶要素が一次元的に配列されており，各要素には配列上の位置に対応した一連番号がふられているというものである．この番号はアドレスとよばれ，通常は0から始まる自然数が用いられる．記憶要素の大きさはビット，バイト，語などに規格化されているが，以下しばらくはこれらの違いを特に問わないことにする．CPUからあるアドレスが指定されると，それに対応する記憶要素に対して“読出し”あるいは“書込み”の動作が行なわれる．

プログラム上でのアドレス操作のために設けられている機構をアドレス機構というが，これには大きく分けて次のような2つのものがある．その第一は命令操作の対象をアドレスによって指定する機構で，命令語の番地部がこれにあたる．アドレスを1個だけ，2個だけ，3個だけ，さらには先頭アドレスと個数を与えるなどの方法によってある範囲を指定するものなどの形式がある．第二のものは，第一の機構によって指定されたアドレスに対してあるバリエーションを付けるための機構である．この機構は命令語の番地修飾部によって制御されるようになっているが，内容的にいってさらにふたつの形式が区別できる．そのひとつはレジスタ修飾という方式で，与えられたアドレスにあるレジスタの内容を加減算して修飾するものであり，アドレスの原点を局所的に平行移動することによってアドレス操作を単純化するものである．具体例としてはインデックス修飾あるいはベース・レジスタ修飾などが代表的であるが，さらに様々な変形が工夫されている．

第二の機構に属するもうひとつの形式は，間接アドレス方式といわれ，第一の機構から与えられたアドレスに記憶されている内容として必要なアドレスをうるものである．この新しくえられたアドレスに対してさらに繰返して間接アドレス形式を適用することができるようになっているのが普通で，原理的には繰返しの深さに制限をおかない方が自然である．間接アドレス方式のねらいはプログラムの表面には仮の名目的なアドレスを出しておき，実質的なアドレスをすり変えるとか，その決定を適当な時点にまで引延ばすとか，命令語での番地部には盛込み切れない情報を追加できるようにするとかといったアドレス操作の融通性をうることにある．

以上ではフォン・ノイマン・モデルにおけるアドレス機構についてみてきたが，このような考え方に従わない方式も提案されていて，部分的に実用されるなど次第にその重要性が注目を集めてきている．これらは非フォン・ノイマン・モデルとよばれることもある．

このようなものとしてよく知られているのは，連想アドレッシングとスタックであるが，ここでは深入りしない．非フォン・ノイマン・モデルの考え方の基礎には，フォン・ノイマン・モデルのもつ制約性への批判があるが，これについても後に譲りたい．以下では特に断らない限りフォン・ノイマン・モデルの枠内で議論することにする．

### 2.3 アドレス写像

プログラムで用いられるアドレスすべての集合をアドレス空間という．ここで扱うアドレス空間はどれも0から$n$までの自然数からなるという構造をもっているものとする．メイン・メモリの物理構造に対しては一意的にひとつのアドレス空間が定められているが，これはその物理構造に対する論理構造のうちでも最も基本的なものである．これを基本アドレス空間といい，この空間に属するアドレスを実アドレスという．これに対して一般の論理構造に対するアドレス空間を論理アドレス空間といい，これに含まれるアドレスを論理アドレスという．プログラムは一定の論理構造を仮定して展開され，それを構成するデータや命令などの要素には論理アドレスが対応され，それらによって論理アドレス空間が構成されることになる．プログラムは実行されるまえに実際のメイン・メモリ，したがって基本アドレス空間内に配列されなければならない．すなわち，論理アドレス空間から基本アドレス空間への写像が必要になる．この場合プログラムが正しく実行されるためには，論理アドレス$i$番地にある要素は実アドレス空間でも$i$番地に入らなければならない．

マルチプログラミングにおいては，ひとつの基本アドレス空間に同時にふたつ以上の論理アドレス空間が写像されることになるので，上記の$i \to i$という条件を満足できなくなる．この撞着を解く鍵はCPUが用いるアドレスを実アドレスから論理アドレスにすりかえてしまうことである．そしてCPUとメイン・メモリの間に論理アドレスを実アドレスに変換する機構を別に用意し，CPUから$i$番地へのアクセスが要求されるときには，論理アドレス$i$に対してこの変換機構が与える実アドレス$f(i)$に対してアクセスを行なうようにする．

いまふたつのプログラム A, B に対してそれぞれ $f_A$, $f_B$ という変換を定め, $f_A(i) \neq f_B(j)$ (ただし, $i$, $j$ はそれぞれプログラム A, B の論理アドレス空間に属する任意のアドレスとする.) が満足できるようにできればうえに述べた撞着は解消する. ただこのような $f_A$, $f_B$ を与えるアドレス変換機構をどのようにして具体化するかという問題が残ることになる. プログラム A, B, C, …に対して, それぞれ $f_A$, $f_B$, $f_C$, …を構成することはこれらのプログラムに対してメモリ割付を行なうことである. マルチプログラミングにおけるメモリ割付ではひとつの条件として動的な割付が可能になることが必要である. したがって, $f$ には動的な可変性が要請されることになる. さらに $i \rightarrow f(i)$ の変換はメイン・メモリへのアクセスが行なわれるたびごとに必要となるものであるから, 当然できるだけ短い時間で完了できるものでなければならない. さらに実現性という点から変換機構に要するコストも大きな問題となる.

現在実現されているアドレス変換機構においては, いくつかの特異点を除いてはつぎのような性質をもったアドレス写像が用いられている.

$$f(i+1)=f(i)+1 \qquad (1)$$

論理アドレス空間のすべての $i$ に対して (1) 式が成立するならば, $f(i)=i+f(0)$ となり, $f(0)$ の値を入れるレジスタと加算回路を用意すれば, アドレス変換機構が完成する. いくつかの特異点がある場合には, それらを $i_1$, …, $i_k$, …, $i_p$ とすれば, $i_k \leq i < i_{k+1}$ なる $i$ に対しては, $f(i)=(i-i_k)+f(i_k)$ となるから $f(0)$, $f(i_k)$, $k=1$, …, $p$ の $p+1$ 項の要素から成るテーブルと加算回路によってアドレス変換機構が実現できる. 特異点をおくということは, これらの点を切れ目として, 論理アドレス空間をいくつかの部分に細分化することにあたる. この細分化をプログラマーに指定させるようにしたものと, システムで自動的に行なうようにしたものとがある. プログラマーに指定させる場合でも, $f(i_k)$ などのテーブルを管理するのはすべてシステムの仕事である. この仕事までプログラマーが行なうようにしてしまえば, それはアドレス機構に述べたベース・アドレス方式になる. アドレス変換機構は, 論理構造間のトランスレーションとして行なわれるもので, 上位構造となる側の論理構造からはみえないものであるのに対して, ベース・アドレス方式はひとつの論理構造内での機構として, プログラマーにみえているのもである.

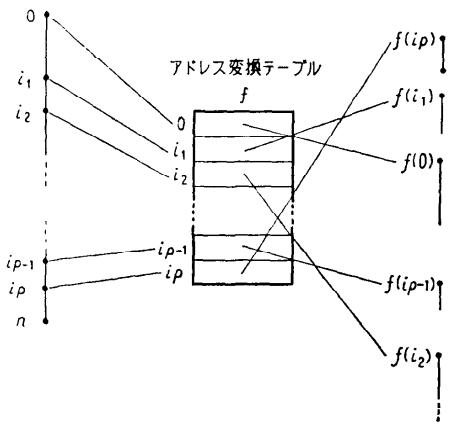



**図-1** ブロック化によるアドレス写像

## 3. 仮想記憶

### 3.1 補助メモリ

メイン・メモリはアドレス機構を通して CPU から直接アクセスされる記憶装置のことであることはいうまでもないが, これに対比されるものとして補助メモリあるいは外部メモリという概念がある. 補助メモリはメモリ階層においてメイン・メモリのすぐ下にランクされていて, プログラムやデータがメイン・メモリに入り切らない場合に直ちに必要とならない部分を入れておくのに用いられている記憶装置で, CPU からアクセスするためにはいったん入出力機構を経由しなければならないものである. 大きなプログラムを作成する場合にはそれをいくつかの部分に分割し, 適当なスケジュールを立ててメイン・メモリにおいておく部分の入換えを行なうようにしなければならない. すなわち, オーバレイの手続をとる必要がある. オーバレイを組込んだプログラムの作成はいろいろ面倒な工夫を要する上に, さらに大きな欠点としてオーバレイの機構がメイン・メモリの容量ならびに入出力機構に依存する要素が多いため, プログラムの互換性に強い制約をもたらすことになって望ましくない. この難点を解決するために考えられたのが"単一レベル・メモリ"という方式である. この方式はメイン・メモリと補助メモリというメモリ階層の2段階にまたがったメモリ構造の上位構造として, アドレス機構ではもとのメイン・メモリと同じであり, 容量では補助メモリと同じであるような論理構造を構成し, しかもプログラム処理速度などに関してはオーバレイに匹敵する性能のものを実現しようとするものである. この上位構造では

補助メモリの存在が消去され，メイン・メモリの容量が飛躍的に増加されることになる．したがってこの論理構造でプログラミングを行なう限り，記憶容量の制限は事実上無視してよいことになる．このように単一レベル・メモリ方式によって与えられる論理構造は“仮想記憶”とよばれている．

仮想記憶を実現するためには，この論理構造をその下位構造であるメイン・メモリと補助メモリとを組合せた構造にトランスレートする具体的な機構を用意する必要がある．仮想記憶は当然論理アドレス空間としての性質をもっているが，それの与える論理アドレスを特に“仮想アドレス”という．仮想アドレスはそれに対するアクセス要求が出たときにはメイン・メモリ上の実アドレスにトランスレートされなければならない．すなわち，仮想記憶のトランスレーションにはひとつのアドレス変換機構が具えられていなければならない．

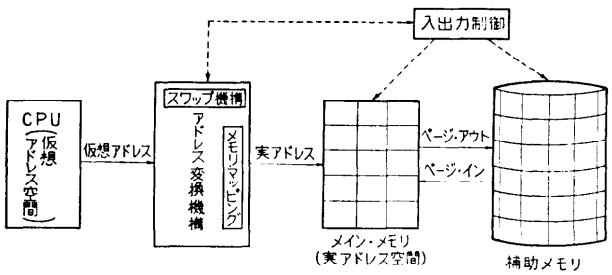
仮想記憶の定める論理アドレス空間の大きさ $N_V$ は，メイン・メモリの定める実アドレス空間の大きさ $N_R$ より大きいはずであるから，時間の違いを無視するとふたつ以上の仮想アドレスが同じ実アドレスに写像されるということが起きる．したがって，メイン・メモリの内容は必要に応じて補助メモリに移しかえ，また必要に応じて補助メモリからメイン・メモリへ戻すという機構がなければならない．この機構をメモリ・スワップ機構という．

以上に述べたことを要約すると，仮想記憶はメイン・メモリと補助メモリという2階層からなるメモリ構造に対して，適当なアドレス変換機構とメモリ・スワップ機構を導入し，メイン・メモリと同じアドレス機構をもち，補助メモリと同じ容量をもったメモリ構造として構成された論理構造であるということができる．

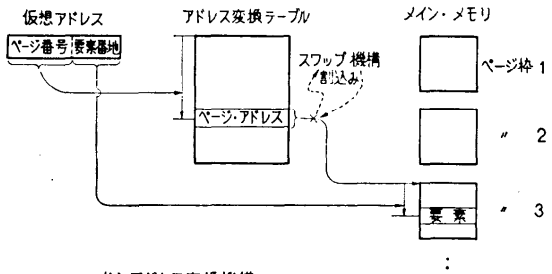
仮想記憶のアドレス変換機構においても**2.3**に述べたアドレス写像の考え方がそのまま適用される．したがって，写像における特異点の設定，すなわちプログラムの細分化をプログラマーに委せるかどうか，委せない場合にはシステムでどのような細分化を行なうかを決めなければならない．このような細分化によってえられるプログラムの各部分は，通常そのままメモリ・スワップ機構におけるメモリ入換えの単位となる．どの部分をどの部分と入換えるかに対しては選択の余地がありパフォーマンスの向上に活用できる．

### 3.2　ページング方式

仮想記憶の実現方式のうちでプログラムの細分化を一定のサイズで機械的に行なうものをページング方式という．この方式ではプログラムの細分化はプログラマーの目にまったくれ触ないレベルで行なわれているので，論理的には仮想記憶かどうかによってプログラムの作成が影響をうけることはない．勿論仮想記憶の場合には記憶容量の制限が事実上なくなったと考えてよいという違いがあることはいうまでもない．ページング方式の特色は記憶容量の点以外では論理構造に何ら変化がないことである．したがって，従来のシステムにページング仮想記憶を後から追加するようなことがあっても，プログラム互換性には心配がいらないということになる．




(a) ページング仮想記憶機構




(b) アドレス変換機構

**図-2**　ページング方式

ページング方式による仮想記憶機構の概要を**図-2**にもとづいて説明する．図-2(a)にみられるようにメイン・メモリも補助メモリもそれぞれ同じ大きさの単位ブロックすなわちページに細分される．ページの大きさは2のベキ乗となるように選ぶ．それを $2^a$ とすると，仮想アドレスのうち下位 $a$ ビットはページ内での番地を示めし，その上位にある残りの部分はページを指定する番号を表わすものとみることができる．図-2(b)の仮想アドレスにおける破線による区分はこれを示めしている．補助メモリ内

のページはスワップ機構のコントロールによって，必要に応じメイン・メモリ内のページ枠に移されたり(ページ・イン)，そこから再び補助メモリに戻されたり(ページ・アウト)する．

仮想アドレス空間内の各ページはメイン・メモリあるいは補助メモリのいずれかのページとして存在していなければならない．普通には仮想アドレス空間のページと補助メモリのページは固定的な一対一の対応づけが行われているがこれは必ずしも必要ではない．

つぎにアドレス変換機構について説明すると，CPUからのメモリ・アクセス要求として仮想アドレスが与えられると，そのページ番号によってアドレス変換テーブル内から対応するエントリーを読出す．エントリーの内容は，もし求めるページがメイン・メモリ中に入っていればそのアドレスが書き込まれており，もし入っていなければ，そのことをスワップ機構に知らせるための記号が書かれている．この記号によって割込みを発生してスワップ機構を起動させるのが通常のやり方となっている．アドレス変換テーブルによってページ・アドレスが与えられた場合には，これで仮想アドレス中のページ番号をすり換えて実アドレスが形成され，それを用いてメイン・メモリへのアクセスが行なわれることになる．

割込みによって，スワップ機構が起動された場合には，必要なページを入れるためにメイン・メモリ内で適当なページ枠を選定しなければならない．この選定は論理的にはどんなルールを用いてもよいが，システムの効率には重大な影響をもつ．仮想記憶の目標のひとつは，効率がオーバレイの場合をあまり下回らないことであった．したがって，効率のよいルールを見付けることがページング方式での重要な課題となっているが，ここではそれに深入りする余裕がない．とにかくこれによってページ枠が決定されると補助メモリに対して出力動作が開始され，そのページ枠を占有していたページをページ・アウトし，つづいて入力動作によって必要なページをページ・インする．

これと並行してアドレス変換テーブルがアップデートされ，ページ・アウトされたページに対応するエントリーには割込み発生用の記号が，ページ・インのエントリーにはそのメイン・メモリ中での番地がそれぞれ書き込まれることになって割込処理が完了し，アドレス変換過程のつぎの動作に進むことになる．ここでは簡単のために割込みが発生してからページ・アウトの動作を行なうように説明したが，これは必ずしもこの通りである必要はなく，先回り的にこの操作を行なっていくつかのページ枠をリザーブしておくことによって，割込処理での仕事を簡略化し，効率改善を計ることも可能である．

ページ・インの場合についても仮想アドレス空間のページをグループ化しておき，グループ内のどれかのページにアクセスがあったときには，そのページ・インと同時にグループ全体をメイン・メモリに持込む方式も有力である．ただ，このグループ化はプログラムの論理的な構造にもとづいて行なう必要がある点で，ページング方式の特徴であるプログラマーからの透明性に若干の制限がつくことになることに注意しなければならない．

ページ・サイズについても効率に関連して多くの議論があるがここでは省略する．

### 3.3 セグメンテーション方式

プログラマーによる論理的な操作として，プログラムの細分化を行なうことを一般にセグメンテーションと呼んでいるが，これを利用した仮想記憶方式をセグメンテーション方式という．この方式では，仮想アドレス空間を多数の部分空間すなわちセグメントの集合とみる．各セグメントにはそれぞれ一意的な番号が与えられて区別が行なわれるようになっている．この番号をセグメント番号という．セグメントの部分空間としての大きさは論理的にはできるだけ大きくとれることが望ましいが，ハードウエア構成上から制限がでるばかりではなく，これがあまり大きいと仮想記憶としての効率に致命的な悪影響がでることになる．通常は普通のサブプログラムが入りうる程度にとっておき，使用上あまり大きなセグメントができないようにプログラマーが用心するというたてまえを仮定することによって設計上の妥協が計られている．このようにして定まるセグメント・サイズの上限はページの場合と同じように2のベキ乗が用いられる．いまそれを $2^s$ とする．

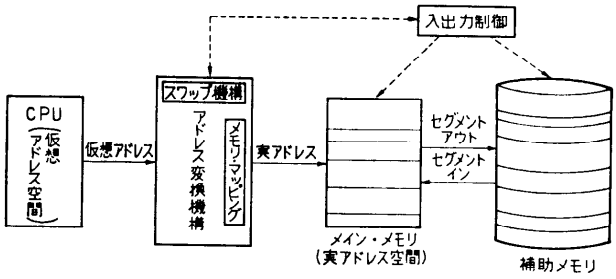
仮想アドレスはふたつの座標—セグメント番号とセグメント内での番地である要素番地—によって表わされることになる．この意味でセグメンテーション方式におけるアドレスのことを2次元アドレスと呼ぶことがある．ページング方式においてもページ番号と要素番地という二本立のアドレスが用いられていて一見2次元のようにみえるが，この二本立はあくまでもアドレス変換機構内部だけのものであってプログラマーの関知しないところであるのに対して，セグメンテーシ

ョン方式ではプログラマーがふたつの座標を指定するようになっているところに両者の本質的な相違点がある．

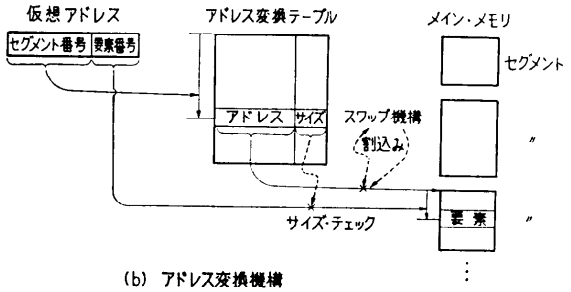
各セグメントに対するメモリ割付はセグメントの上限サイズをとったのでは無駄が大きいので，実際に用いられる範囲をセグメントごとにプログラマーに指定させ，その範囲についてだけメモリを割付けるようになっている．このセグメント・サイズはセグメントごとに設けられているセグメント・ディスクリプターに入れられている．このようにしてセグメンテーション方式におけるプログラム細分化によるブロックの大きさは可変長になるのである．

セグメンテーション方式による仮想記憶機構の概要を**図-3** に示す．この図と図-2 とを比較すれば明らかな通り，セグメンテーション方式とページング方式は機構的に共通するところが多い．両者の相違点を図に従って確認して行くと，まず仮想アドレスがふたつの部分に切られている点では同じであるが，その仕切りが一方が実線で他方が破線という違いがある．実線はこの仕切りがプログラマーによって意識されるべきものであることを示めし，破線はそうでないことを示めす．つぎにアドレス変換テーブルにおいてそのエントリーの構造に差異があり，セグメンテーション方式の場合にはアドレスの他にサイズが加わっている．これはページが固定長ブロックであるのに対して，セグメントが可変長ブロックであることによるものである．またページング方式の場合には与えられた要素番号は必ずページの中に存在するのに対して，セグメンテーション方式の場合にはその番号がセグメント・サイズを越えるものについてはメモリ割付がなされておらず，したがってアクセスは許されていない．

固定長ブロックと可変長ブロックの違いはメイン・メモリにおけるブロックの取扱いにおいて決定的な相違を生む．ページング方式ではメイン・メモリは固定的なページ枠によって細分化され，補助メモリ上のページはページ枠のどれにでも入ることができた．これに対してセグメンテーション方式ではメイン・メモリにおいてあるセグメントの跡に別のセグメントが入りうるという保証はなく，場合によってはひとつのセグメントを入れるために，複数個のセグメントをスワップ・アウトしなければならないこともある．また入りえた場合にも余りが出ることもあるなどメモリ割付の




(a) セグメンテーション仮想記憶機構




(b) アドレス変換機構

**図-3**　セグメンテーション方式

ために複雑なコントロールを必要とし，スワップ機構の負担が大きくそこだけをみる場合には大変効率の悪いものにみえる．それにもかかわらずセグメンテーション方式が用いられる理由は，ページング方式の終りに述べたページのグループ化による効率向上がセグメンテーション方式では自動的に実現されていること，すなわちこのようなページ・グループがひとつのセグメントに対応するとみられることがひとつ，さらにセグメントという単位が後に説明する記憶保護機能での保護の単位としての存在意義をもつこと，セグメンテーションの機構がプログラム構造に動的な可変性を与えるために用いうることなどが大きな理由としてあげられる．

このようにセグメンテーション方式は非常に可能性に豊んだものであるが，これを十分に使いこなすためには高度なオペレーティング・システム的な裏付けが不可欠であり，単に仮想記憶としてページング方式のような即効性を望むのは必ずしも適切な見方ということはできない点に注意しなければならない．セグメンテーション方式の特徴のうち非仮想記憶方式的なもののみを活用し，仮想記憶方式としては別にページング方式を活用して両者の長所だけを利用することも可能であり，それを実現したのが MULTICS システムである．このシステムでは，セグメントをさらに細分化

するものとしてページング方式を用いるようになっている．すなわちセグメンテーション方式でのアドレス変換機構からの出力である実アドレスを，ページング方式の仮想アドレスとしてそのアドレス変換機構に入力して，そこからえられる出力を最終的な実アドレスとするわけである．

### 3.4 多重仮想アドレス空間

ページング方式での仮想アドレス空間においては，実アドレス空間と同様に $i$ 番地(ただし $i$ は非負整数)はただひとつしか存在しない．したがってマルチプログラミングを行なう場合には，何らかのプログラム・リロケーション機能を用意しなければならない．これはセグメンテーション方式による仮想記憶の場合についても事情はまったく同様であるが，以下では話を簡単にするために専らページング方式の場合についてのみ考えて行くことにする．

マルチプログラミングのためのアドレス写像の満たすべき条件については，すでに **2.3** アドレス写像において検討した．それを仮想アドレス空間に適用すれば，各プログラム $X$ に対して写像 $f_X$ を構成し，$X \neq Y$ ならば $f_X(i) \neq f_Y(j)$ となるようにとらなければならない．いまページング方式でのアドレス変換テーブルによる仮想アドレスから実アドレスへの写像を $g$ とすれば，$g * f_X$ という合成写像がえられ，仮想アドレスから実アドレスへの写像すなわちひとつのアドレス変換テーブルを構成することになる．したがってマルチプログラミングを行なう場合には最初から $g * f_X$ をひとつのアドレス変換テーブルとして直接構成することができる．これは別のいい方をすれば，プログラム $X$ ごとに固有の仮想アドレス空間を実現したことになる．このようにして構成された同一システムにおいて共存する複数個の仮想アドレス空間のことを多重仮想アドレス空間という．

これは論理的な意味からいえば一般の論理アドレス空間を多重化する場合と何ら異なったものではない．ただ違う点はいまみた通り仮想アドレス空間の場合にはその実現が極めて自然に殆んど新しい機構の追加を必要とせずに可能となることである．追加すべき機構としては，プログラムごとのアドレス変換テーブルの位置を示めすためのレジスタと，プログラムが変わるごとにその内容を更新するための機能とである．

### 3.5 ファイル・ダイレクト・アドレッシング

プログラム・オーバレイにおいては，プログラムの大部分をいくつかのファイルとして補助メモリ上に記憶しておき，実行の必要が起きたところで I/O 命令を出してメイン・メモリに取込んでくるものであった．ところが仮想記憶という考え方によって補助メモリはメイン・メモリの中に一元化されてしまい，補助メモリ上のプログラムに対しても I/O 命令を媒介とすることなく，CPU から論理的な意味でいって直接にアクセスすることができることになったのはこれまでみてきた通りである．

この仮想記憶の考え方をさらに推し進めれば，一般的な外部記憶装置もインライン化してメイン・メモリの中に一元化しうることになり，普通のファイルに対しても CPU から直接アクセスすることが可能になることは容易にみてとれるところである．こうなればプログラムからは I/O 命令が消え去ることになり，そのメリットは非常に大きい．これはいわばメイン・メモリの容量的な制限を完全になくしてしまうことになる．このような方式をファイル・ダイレクト・アドレッシングという．

ファイル・ダイレクト・アドレッシングを現実のものにするためには，従来の計算機アーキテクチュアに対していくつかの変更を行なう必要がある．その第一はファイルを持ち込む結果，アドレス空間が極めて大きくなる必要があり，そのためのアドレス機構をどのようにするかという問題である．命令語のアドレス部を大きくすることは無駄が多く限度がある．この解決案としては相対アドレスあるいはインプリシットなアドレス手法を採用することであろう．第二の変更点としては CPU のデータ構造を高度化することである．現在のファイルはそれ自身ひとつの論理的単位としての意味をもっているものであり，さらにその内部にも論理構造をもっている．これらの構造をメイン・メモリでのアドレッシングにはめ込むための方式として，仮想記憶に用いられているセグメンテーション方式が適していることは特に説明を要しないところである．

## 4. 記憶保護機能

### 4.1 記憶保護の基本概念

記憶保護機能はメイン・メモリの内容がプログラムによって違法に破壊されたり，アクセスされたりすることを防止するための機構である．メイン・メモリへのアクセス動作には内容を読出す“fetch”と内容を書き換える“write”という2つの基本形式があるが，このうち“fetch”の方は読出された内容が命令語として CPU で実行されるだけで，内容そのものがアク

セスしたプログラムに渡ることがない場合を区別するために，この場合のアクセス形式を“execute”，それ以外の fetch を“read”とよんでいる．メイン・メモリに直接アクセスするハードウェアとしては，CPUと入出力チャンネルとがあるが，入出力チャンネルからのアクセスには“execute”の形式がないことはいうまでもない．

記憶保護機能の中心はアクセスの適法／違法をどのように設定するかというところにある．これはアクセス動作の対象と主体とを定立し，その間にひとつの関係を定義することになる．アクセス動作の対象はメイン・メモリ，仮想記憶の場合を考慮して厳密にいえば論理アドレス空間上の記憶内容であるが，ある主体に対して全内容が一様に適法あるいは違法であるというのではもともと記憶保護機能が不要になってしまうわけであるから，何らかの形で全内容から特定の部分を区別することができなければならない．すなわち論理アドレス空間の部分空間を指定する手段が必要である．これには普通論理アドレスの大小関係にもとづいて，ある範囲を指定することによって行なわれるが，セグメンテーション方式による論理アドレス空間の場合には，セグメントをこのための部分空間として用いることができる．これはセグメンテーション方式の重要な効用のひとつである．またページング方式でのように固定的な細分化によるブロックが存在する場合にもそれを単位として部分空間を設定してもよい．

つぎにアクセス動作の主体をどのように設定するかが問題になる．メイン・メモリへのアクセスはプログラム実行によって発生するものであるから，その動作の直接的な主体はプログラムということになる．しかし，マルチプログラミングにおける共用プログラム(shared program)を考えてみれば明らかなように，同じプログラムが異なった目的で実行されることがあり，この場合には当然記憶保護の内容が変わらなければならない．したがってアクセスの主体はプログラムを超えて，それを使用しているある意志体に求めなければならない．計算機システムにおけるこのような意志体としては，MULTICS システムにおける“プロセス”，OS/360 における“タスク”という概念がある．

マルチプログラミング・システムにおいては必ずこのような概念が設定されていなければならないので，アクセス動作の主体としてこれらを用いることは一応合理的であるということがいえる．タスクあるいはプロセスはシステム内では番号によって同定されるようになっているが，使用しうる番号の種類は比較的少なく，同じ番号が時間をずらして異なる実体を表現するのに用いられる．したがって記憶保護は一時的なアクセス対象とアクセス主体との関係だけをチェックできるに過ぎなくなる．しかしファイル・ダイレクト・アドレッシングが実現される場合を考えると，保護されるべきアクセス対象は半永久的に論理アドレス空間内にとどまることになるので，それとプロセスあるいはタスク番号の組合せについてアクセス制御を規定したのでは不十分となることは明らかである．この場合にはプロセスあるいはタスク番号を超えてさらにその背後にある意志体を追求しなければならない．この条件に合うような，いいかえれば半永久的な同定に耐える意志体としては，“ユーザ”という概念が用いられる．これは現在においても，いわゆるファイル保護の水準ではすでに行なわれているところである．

いま，定義されたアクセス動作の対象と主体を用いてこれらの間の関係をつぎのように定めることによってアクセスの適法／違法を判定する．すなわち，ある対象 0 に対して主体 $S$ からアクセス形式 $M$ でのアクセスが許される場合に（これと双対的に許されない場合の方を用いてもよい），つぎのような 3 連子を構成する．

$(0, S, M)$

このようにしてえられるすべての 3 連子の集合をアクセス制御リストとよぶ．いま実際にアクセス要求が出されると，その実行に先立ってそのアクセスにおける対象，主体，形式を調らべ，それらに一致する 3 連子がアクセス制御リスト中に存在すれば，そのアクセスは適法であり，そうでなければ違法とする．

違法なアクセス要求は実行されず，ここで CPU 命令の実行あるいはチャンネル動作を打切って CPU 割込みを発生する．適法なものについては当然アクセスが実行されることになる．

### 4.2 保護方式の諸相

以下では既存の保護方式のうちで代表的なものを採り上げ，前節に述べた基本概念がどのようにして具体的に展開されているかをみて行くことにする．

（1） バウンダリー・プロテクション

実行中のプログラムからアクセスしてよい論理アドレスの上限と下限のアドレスを CPU および入出力チャンネルに付属したレジスタに設定し，この範囲内のアドレスには任意の形式でのアクセスを許し，範囲外のものにはいっさいのアクセスを認めないようにした

ものである．これではアクセス対象はバウンダリーの設定を変えるごとに必要なものひとつが同定される．この設定はモニターによって行なわれ，アクセス主体はモニターからCPUあるいはチャンネルのコントロールを渡されたプログラムということになる．

この保護方式は非常にシンプルで手っとり早いものであるが，プログラムの共用を行なうためにはバウンダリーの設定を複数化する必要があること，バウンダリー内部においても，部分的にアクセス形式に制限を設けたいことが多いことなどの理由から最近の新しい計算機では次第に採用されなくなりつつある．

（2） キイー・プロテクション

論理アドレス空間を一定サイズでブロック化し，各ブロックごとにキィーと称するコードを付す．コードの種類はシステム全体で16種程度である．したがって相異なるブロックに同じコードが付されることになる．一方，各ユーザ・プログラムはその実行開始に先立って上記コードのひとつが与えられ，そのプログラムからのアクセス要求はすべてこのコードを一緒につけて出すようになっている．あるブロックに対してアクセス要求が出されると，そのブロックのキィーとアクセス要求のキィーとが比較され，それらが一致している場合に限ってアクセスが許される．ただし，各ブロックにはコードの他にアクセス形式が指定されていて，その形式でのアクセスのみが許される．

この方式ではコードの種類だけのアクセスの対象と主体とが区別されることになる．したがって，共用プログラムも可能となり，部分的にアクセス形式を制限することができる．しかし，固定サイズによるブロック化とコードの種類が比較的少ないということから，記憶保護の精度に強い限界をおくことになる．

（3） リング・プロテクション

セグメンテーション方式を利用した記憶保護方式である．したがって，アクセス対象はセグメントを単位とすることになり，プログラマーによって自由に対象を構成できることが大きな特徴となっている．アクセス形式もセグメントごとに指定できる．一方アクセス主体に関してはプロセス番号そのものを用いることはアクセス制御リストの構成が複雑になり，アクセスのたびごとにその内容をテーブル・ルックアップしてチェックを行なうことは時間的にいって不可能である．このためプロセスとセグメントをいくつかの階層に分類し，高い階層のプロセスが低い階層に属するセグメントにアクセスすることは認めるが，この逆は許さないことにしてアクセス制御を行なおうとするものである．この階層が環状構造とみることが都合がよいことからリング・プロテクションの名がつけられている．この方式はMULTICSシステムにおいて実現されているが，なおいくつかの弱点が指摘されている．

## 5. おわりに

メイン・メモリの論理構造としては，うえに述べたもの以外にも，特殊なアドレス機能をもった構造として，スタック構造やリスト構造などについても当然触れなければならず，またこのほかの非フォン・ノイマン構造についても，他の項目に説明があるとはいえ，論理構造としての視点からの整理が必要であったかも知れない．しかし，スタック構造およびリスト構造はプログラミングにおけるデータ構造として広く利用されているものであって，その基本的な性格についてはよく知られているのでここではあえて採り上げなかった．非フォン・ノイマン構造をおとしたのは専ら紙数からの理由によるものである．

また，記述したもののなかでも記憶保護については明らかに説明不十分であり，特にバークレイ・コンピュータ会社のLampsonによって導入されたドメイン保護方式はリング保護方式をその縮退形として含むような一般性の高いものであり，将来性も大きいと考えられるところから，ここに入れるべきであったが，概念的な準備に紙数を要するところから見送らざるをえなかった．これにつのては参考文献の6）を参照されたい．

## 参考文献

1) 元岡達編「計算機システム技術」オーム社 1973年.
2) Iliffe, J. K.: Basic Machine Plinciples. American Elsevier, New York, 1968.
3) von Neumann, J., et al.: Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946.
4) Needham, R. M.: Protection System and Protection Implementations. Proc. AFIPS 1972 FJCC.
5) Saltzer, J. H.: Protection and the Control of Information Sharing in Multics. Com. ACM Vol. 17, No. 7 July 1974.
6) Lampson, B. W.: Dynamic Protection Structures. Proc. AFIPS 1969 FJCC.

（昭和50年1月8日受付）
（昭和50年2月6日再受付）