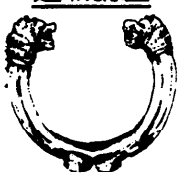
Vol. 33 No. 11　　情　報　処　理　　Nov. 1992

連載講座



計算機の記憶システム—II

# キャッシュと仮想記憶の技術†

朴　泰祐††

## 1. はじめに

キャッシュと仮想記憶はどちらも現在の計算機にとって必要不可欠な技術となっている．前者はますます高速化する CPU の性能を最大限に活かすために必要であり，後者は大規模なプログラムを限られたメモリ上で実行したり，複数のプログラムが同一のメモリ空間上で共存することを許すために必要である．本稿ではこれらの技術について解説する．

## 2. 記憶装置の容量とアクセス速度

計算機における記憶装置は，その容量が大きく，そのアクセス速度が速いことが理想である．しかし，現実には容量の大きい装置はアクセス速度が遅く，逆にアクセス速度の速い装置は容量が小さいという**記憶容量とアクセス速度のトレードオフ**が存在する．計算機システム中の記憶装置を見ると，最も高速で小容量のレジスタから低速で大容量のテープ装置まで，いくつかの階層が構成されている．これを**記憶装置の階層** (Memory Hierarchy) と呼ぶ．

最近のデバイス技術の発達により，メモリデバイスは大容量・高速度の方向で急速に発展しつつあるが，それでも十分ではない．なぜならば，それを利用する CPU そのものの速度も同時に飛躍的に向上してきているからである．特に，近年マイクロプロセッサの主流となりつつある RISC (Reduced Instruction Set Computer) アーキテクチャをもつ CPU では，そのクロックレートは数十から百 MHz 程度まで引き上げられている．これは，一つの命令の実行が数十ナノ秒ですんでしまうことを意味し，メモリはこの速度に追随する必要がある．さらに，CPU の性能が向上してきたということは，単位時間当りの処理量の増加を意味し，結果としてより大量のデータの処理が可能となる．このため，プログラムやデータの大きさもどんどん大きくなっており，結果としてさらに大容量のメモリが必要となる．

現在多くの計算機では主記憶用メモリデバイスとしては集積度の高い DRAM が利用されている．これにより，CPU 速度に見合った大量のデータ処理を行うに足る大容量の主記憶が実現できる．しかし，そのアクセス速度は CPU の速度に比べかなり遅く，DRAM をそのまま CPU に結合したのでは，メモリアクセス時間がプログラムの実行のボトルネックとなり，CPU の高速性がほとんど活かされなくなってしまう．そこで，このギャップを埋めるため，アクセス速度の速いバイポーラなどの SRAM を利用し，しかも記憶容量がなるべく大きいように「見せかける」工夫をした技術が**キャッシュ**である．

## 3. キャッシュの原理

本章以降の説明では，単に「データ」という言葉を使った場合は，「記憶装置上のデータ」全般を示す．すなわち，CPU からみて，それがプログラムの命令（インストラクション）であるのか，データ（オペランド）であるのかという区別のないものである．

キャッシュ (cache) とは英語で「貴重品を入れておく場所」という意味であるが，計算機用語としては，「比較的利用頻度の高いものをアクセスしやすい手近なところに置いておく」という手法を指す．キャッシュの具体的イメージを身近なものを例にとって説明しよう．

今，あなたはいろいろな蔵書を使って自分の机

† The Technology of Cache and Virtual Storage by Taisuke BOKU, (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学電子情報工学系

で仕事をしているとする．あなたの机の上には数冊の本が置けるとする．それらの本は目の前にあるので，いつでもすぐに開くことができるが，その冊数は非常に少ない．一方，別の部屋には大きな本棚があり，そこには数百冊の本が置ける．その本棚にはあなたの持っている全ての本が置けるが，その部屋は机のある部屋からは遠く，1冊の本を取りに行くにも時間と手間がかかる．このような環境で効率的に仕事を進めるには，「よく使う本を選んで机の上に置き，そうでない本は本棚に置いておく」というやり方を取ればよい．頻繁に開かれる本は机の上にあり，簡単に読めるので作業の効率が高い．また，そうでない本は机の上にないため本棚まで取りにいく必要があるが，その頻度はそれほど高くないため，その労力と時間は許容できる範囲にある．さらに，仕事の進みぐあいに応じて，机の上の本が不必要になって本棚の別の本が必要になった場合，その両者の適当な組合せを入れ替える必要も生じるだろう．

計算機では上の例の本棚が主記憶，机の上の置き場所がキャッシュに当たる．**図-1** にキャッシュと CPU と主記憶の関係を簡単に示す．キャッシュはアクセス速度の非常に速い小容量のメモリで，主記憶はアクセス速度の遅い大容量のメモリである．プログラムを実行する際，CPU が参照するメモリ内容は，もしキャッシュ上に存在していればそれを使い，なければ主記憶までアクセスしにいく．キャッシュは CPU に非常に近く，しかも高速なので，CPU が必要とする多くのデータがキャッシュ上にあれば，非常に高速な処理が期待できる．

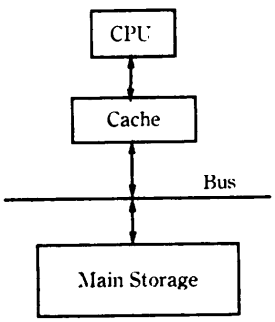



**図-1** CPU，キャッシュ，主記憶の関係

## 4. 局所原理とキャッシュの構成

キャッシュが有効に働くには，アクセス頻度の高いアドレスのデータをキャッシュに置いておくことが重要である．一般的なプログラムのメモリアクセスのパターンを解析すると，次のような特徴が現れる場合が非常に多い．

1. ある一定時間内にアクセスされるデータは比較的近いアドレスに分布する．

2. ある一定のアドレスのデータに対するアクセスは比較的短かい時間内に再発する．

前者は「空間的局所性」，後者は「時間的局所性」と呼ばれる法則で，これらを合わせて**局所原理 (Principle of Locality)** と呼ぶ．

これらはプログラムの持ついくつかの典型的な構成（ループ構造や，よく使う値をいったん変数に入れておくなど）の結果生じるものである．キャッシュでは特に後者の時間的局所性が重要になる．時間的局所性が現すところは，「一度アクセスされたアドレスは近い将来再びアクセスされる」ということである．すなわち，今アクセスされたアドレスをキャッシュにとっておけば，次回そのアドレスが再び参照されたときにはそのデータはキャッシュにあるためアクセス時間が短かくてすむ．そのようなことが何度も生ずることによって処理速度が向上するわけである．

キャッシュは主記憶上の全アドレス空間の部分集合のデータをもっている．キャッシュはメモリであるから，アドレスによってそのデータの位置が指定される．ここで問題となるのは，何を単位として主記憶上のデータとキャッシュ上のそれを対応させるかである．最も簡単な方法は1バイトごとに対応づけを行う方法であるが，これではキャッシュの管理の単位が細かすぎ，そのために大量のハードウェアが必要となり好ましくない．そこで，キャッシュはある一定長のデータごとに分割され，それぞれのデータが主記憶上のどのアドレスから始まるデータに対応するかが管理されている．この一定長のデータのかたまりを**ブロック**あるいは**ライン**と呼ぶ．ある一つのデータを読むためにはそれを含む1ブロックのデータ全てをキャッシュにもってこなければならないが，空間的局所性により，ブロック内の他のデータも近い将来にアクセスされる可能性は高く，ブロックご

とに管理を行ってもその効率はかなり高くなる．

ブロックは主記憶とキャッシュの間のデータの交換を行う単位である．ブロックの大きさ（ブロック長）はキャッシュの構成法によって異なるが，一般には十数バイトから百バイト程度であることが多い．ブロック内の一連のデータは主記憶上で連続しているアドレスのそれに対応する．しかし，キャッシュ中の各ブロックは必ずしも連続したアドレスであるとは限らない．すなわち，キャッシュは主記憶の連続した一部をキャッシュの容量分だけまとめてもってきたものではなく，ブロックごとにまったく異なったアドレスのデータを保持している．

キャッシュ内の高速メモリには各ブロックのデータが並んで収められている．その一つずつを**エントリ**と呼ぶ．キャッシュ内の各エントリには，それが主記憶のどのアドレスから始まるデータに対応するかを示すアドレス情報がつけられている．全エントリのアドレス情報を格納している場所を**ディレクトリ**と呼ぶ．このイメージを**図-2**に示す．CPU から与えられるアドレスのビット長を $n$ とすると，これはキャッシュブロック単位のアドレスを示す上位 $n-m$ ビットと，キャッシュ内でのアドレスを示す下位 $m$ ビットに分けられる（この場合，ブロック長は $2^m$ バイトである）．CPU がメモリをアクセスすると，まずアクセスされたアドレスがディレクトリと比較され，アクセスされたアドレスがどれか一つのエントリの中に収まっていれば，求めるデータはその対応するブロック中にあることになる．この状態をキャッシュに**ヒット**したという．ヒットした場合，アクセスはキャッシュ中のそのブロックのデータに対して行われる．ミスヒットの場合は主記憶までアクセスしにいかなければならない．

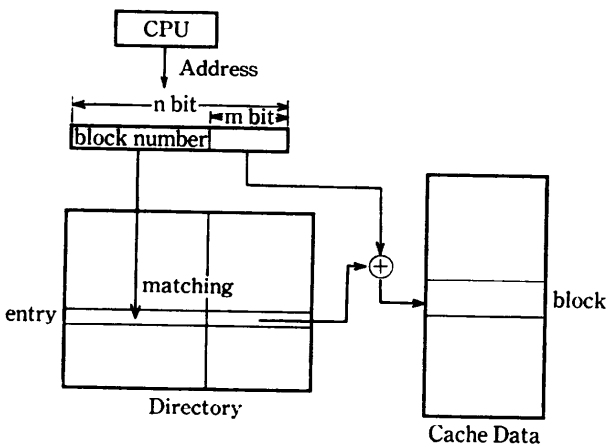



**図-2**　ディレクトリの参照

## 5. キャッシュの実装法に関する分類

ここではキャッシュの具体的実装法について，幾つかの方式を分類する．

### 5.1 主記憶とキャッシュのアドレス・マッピング

キャッシュの各エントリには主記憶のブロックが入るが，ここで問題となるのが「主記憶のどのブロックがどのエントリに入るか」である．この対応づけを**キャッシュのアドレス・マッピング**と呼ぶ．

理想的には，どのブロックがどのエントリに入ってもよさそうであるが，キャッシュの制御と組み合わせて考えるとそれは難かしいことが分かる．もし主記憶のブロックとキャッシュのブロックのアドレスの対応に何の制約もなければ，CPU から与えられたアドレスがキャッシュ内に存在するかどうかを調べるには，それをディレクトリ中の全エントリと比較しなければならない．これを高速に処理するためには，連想メモリ (8.1 参照) のような高価で複雑なハードウェアを用意する必要があり，キャッシュのブロック数を増やした場合，コストがかかり過ぎる．

そこで，一般的にはキャッシュのアドレス・マッピングにはなんらかの制約条件がつけられている．代表的なマッピング方法を以下にあげる．

**(a) ダイレクト・マッピング**

キャッシュのエントリ数を $N=2^d$ としたとき，主記憶上のブロックの並びを $N$ ブロックごとにインタリーブすることを考える．すなわち，主記憶のブロック番号が 0 から始まっているとすると，主記憶の第 $i$ ブロックをキャッシュの第 $N \bmod i$ エントリに入れるように決める．こういうマッピングを**ダイレクト・マッピング**と呼ぶ(**図-3**)．

ダイレクト・マッピングの利点は，主記憶上のブロックがどのキャッシュ・エントリに入るかが一意的に決定されるため，ディレクトリの中を一つ一つ調べなくても簡単にヒットしたかどうかが分かる点である．また欠点は，あるブロックから $N$ ブ

ロック離れたブロックは元のブロックとキャッシュ内で共存することはできない点である．図-3でいえば，2次元的に並んだブロックのうち，横の列が同じブロック同士は共存できない．

**(b) セット・アソシアティブ**

ダイレクト・マッピングにおける制約は非常に厳しいので，それを少し緩めることを考える．今度はキャッシュの隣り合う二つのエントリを一組とし（総エントリ数を $N=2^d$ とすると，$N/2$ 組のエントリ・ペアができる），主記憶のブロックの並びを $N/2$ 組ごとにインタリーブする．そして，第 $i$ ブロックはキャッシュの第 $(N/2) \bmod i$ エントリ・ペアのうちどちらにでも入れるようにする．こういうマッピングを**2-ウェイ・セット・アソシアティブ・マッピング**と呼ぶ（**図-4**）．

この手法は一般的に $n$-ウェイに拡張できる．すなわち，キャッシュのエントリを $n$ 個で一組とし，主記憶の第 $i$ ブロックを第 $(N/n) \bmod i$ エントリ組のどれか一つにマッピングするのである．逆に，これを1-ウェイにしたものがダイレクト・マッピングである．$n$-ウェイ・セット・アソシアティブ・マッピングでは，ダイレクト・マッピングにおける制約が緩められ，図-3で同じ列に含まれるブロック同士でも $n$ 個までのブロックが共存できる．

2-ウェイの場合を例に，ヒットしたかどうかの判定法を示す．まずCPUからのアドレスでブロック番号が決定されたら，ディレクトリ中の対応するエントリ・ペア（二つのブロック番号を含む）が調べられる．2組の比較演算器を並列に使って，対象となるブロック番号がどちらかのエントリに存在するかどうかを調べる．存在すればヒットしたことになり，対応する側のエントリのブロックがアクセスされる．どちらにもヒットしなければミスヒットである．$n$-ウェイの場合はこの並列動作する比較演算器が $n$ 個必要になる．

セット・アソシアティブ方式は，比較的簡単なハードウェアでヒットの検出が可能で，かつダイレクト・マッピングに比べ制約条件が緩くなっているため，2-ウェイ，4-ウェイ程度のものがよく用いられる．

**(c) 完全アソシアティブ**

$n$-ウェイ・セット・アソシアティブの $n$ をキャッシュのエントリ数に等しくすれば，主記憶の全ブロックが一つのセットに収まるので，結果としてどのブロックでもキャッシュの任意のエントリに対応することができるようになる．つまり，マッピングの制約条件はまったくなくなり，置換アルゴリズム（**5.2** 参照）なども自由に選べる．このようなマッピングを**完全アソシアティブ・マッピング**と呼ぶ．

完全アソシアティブでは，キャッシュのエントリ数だけの比較器が必要となり，連想メモリを用意することになる（連想メモリについては **8.1** 参照）．これは非常にコストが高くなるので，一般にはほとんど用いられない．実際，統計的にはセット・アソシアティブでほとんど十分であることが知られているため，

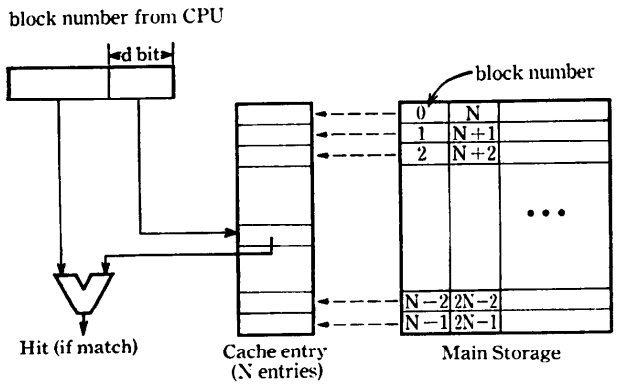



**図-3** キャッシュのマッピング（ダイレクト・マッピング）

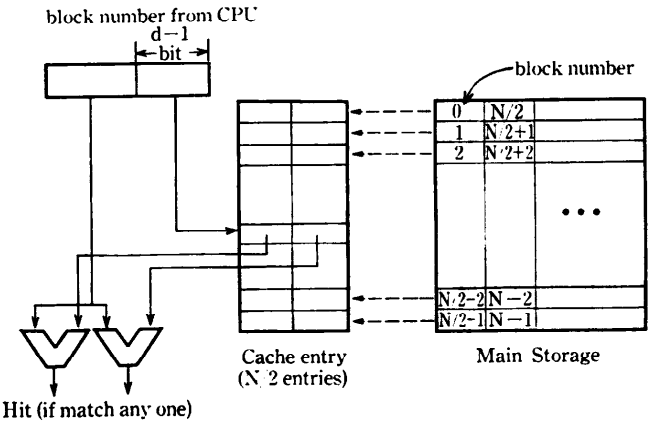



**図-4** キャッシュのマッピング（セット・アソシアティブ・マッピング）

多くのシステムではそれを採用している.

### 5.2 キャッシュの置換アルゴリズム

キャッシュの容量は有限であるため，ミスヒットが生じた場合，新しいブロックを読み込むためには現在キャッシュ内にあるブロックのどれかを追い出さなければならない．これを**キャッシュの置換**と呼び，どのブロックを置換対象とするかを決定するアルゴリズムを置換アルゴリズムと呼ぶ.

ダイレクト・マッピング方式の場合，新しいブロックが入るエントリは一意に決定されるので，当然そのエントリに入っていたブロックが追い出される．しかし，セット・アソシアティブや完全アソシアティブでは，幾つかの追い出し候補ブロックが存在する（後者の場合は全ブロックが対象となる）ので，どれか一つを決定しなければならない.

置換アルゴリズムには以下のようなものがある.

**(a) FIFO**

最も単純な置換アルゴリズムは **FIFO** (**First In First Out**) である．これは，候補エントリ中で最も古くからあるエントリを追い出す方法である．FIFO 方式を実装するには，各エントリに「そのブロックが主記憶から取ってこられた時刻」が記録されている必要がある.

たとえば，ミスヒット時に値が1増えるカウンタを用意し，主記憶からブロックをもってきたときにその値をエントリに記録する．この方法だと，候補の中から最もカウンタ値の小さいものを選ぶ必要があるので，2-ウェイ・セット・アソシアティブ程度なら二つの値を比較するだけなので簡単だが，完全アソシアティブとなると比較を並列に行おうとすれば膨大なハードウェア量が必要になってしまう.

もう一つの方法は，候補の群（セット）の中に循環バッファのようなものを用意し，エントリが埋められたときにその番号を記録する方法である．そして，最も古いエントリ番号が記録されている場所を指すポインタを用意し，それを参照する方法である（ポインタはエントリへの読み込みのたびに一つずつ循環バッファ内で移動する）．この方法は候補数が幾つあっても同じ負荷で決定ができるので，セット数が多い場合や，完全アソシアティブの場合に有効である.

どちらの方法を用いるかは，セット数に大きく依存する.

**(b) LRU**

FIFO の欠点は，最も古く主記憶からもってきたブロックが，最も使用頻度の低いブロックであるとは限らないという点である．古くからあっても何度も参照されるようなブロックがあった場合，FIFO でそれが追い出されたとしても，またすぐにそのブロックがアクセスされてミスヒットを生じることが多い.

そこで「最も古くからあるブロック」ではなく，「最後にアクセスされた時刻が最も古いブロック」を選ぶというアルゴリズムを考える．これは時間的局所性を反映しているので，一般的には FIFO よりも優れていると考えられる．この方法を **LRU** (**Least Recently Used**) と呼ぶ．LRU ではそのブロックがもってこられた時刻は関係なく，最後にアクセスされた時刻で比較を行って追い出しブロックを決める.

LRU の実装法としては，FIFO で使った二つの方法をそれぞれ改良する方法が考えられる．前者の改良としては，カウンタをキャッシュ参照のたびに1増やすようにし，ヒットしたエントリの値を更新する．特に，2-ウェイ・セット・アソシアティブの場合は非常に簡単で，一つのセットの中のエントリに1ビットのフラグを一つ用意し，最後にどちらのエントリがアクセスされたかを0か1の情報で覚えておけばよい．それ以上のセット数では順序づけのための機構が必要になり，現実的には 4-ウェイ・セット・アソシアティブ程度がハードウェアで実装する場合の限界である.

後者の改良は若干難かしくなる．ヒットしたエントリの番号を単純に循環バッファに書き込んでいくと，同じブロックばかりが連続してアクセスされた場合，循環バッファの中はそのブロック番号だけで埋まってしまう．そこで，この場合はエントリ番号をリスト構造で管理し，アクセスされたエントリをリストの最後尾にもっていくようにすれば，リストの先頭に最も古くアクセスされたエントリが残るようになる．しかし，このようなハードウェアは実装が難かしく，完全アソシアティブ方式に対応する LRU 制御装置は大変高価になる.

### 5.3 主記憶への書き込み方法

キャッシュからエントリが追い出される際，もしキャッシュに対するデータ書き込みがあった場合は主記憶とキャッシュのブロックの内容が違ってしまうため，それを主記憶に反映させる必要がある．逆に，書き込みがなかった場合はキャッシュのブロックの内容は捨ててしまえばよい．キャッシュの内容が主記憶と異なっている場合，そのブロックは**汚れている（dirty）**という．

このような場合，なんらかの方法でキャッシュ内容を主記憶に書き込まなければならない．これには以下の二つの方法がある．

**（a） ストア・イン方式**

ブロックが追い出されるとき，ブロック全体をまとめて主記憶に書き込む方式．このために各エントリにはダーティ・ビットと呼ばれる1ビットのフラグを用意しておく．そのブロックに対して1度でもCPUからの書き込みがあったら，そのフラグは1にセットされる．そして，追い出されるときにダーティ・ビットが1なら主記憶への書き込みを行う．

ダーティ・ビットはそのブロックのどのバイトが汚れているかという情報まではもたないので，主記憶に書き込まれるのはブロック全体である．このため，ブロック長が長い場合，追い出し時に要する時間が長くなる．

**（b） ストア・スルー方式**

この方式は，CPUからの書き込みがあった場合，キャッシュのブロックにそれを書くと同時に，主記憶にも同じ内容を書き込みにいく方式である．この方式ではキャッシュの内容と主記憶の内容は常に一致しているため，追い出し時の書き込みの必要がない．したがって，ブロックの高速な置換が可能である．

ストア・スルー方式の問題点は，キャッシュへの書き込みアクセス時間が遅くなることである．キャッシュと主記憶への書き込みは同時に行われるが，主記憶のほうがアクセス時間が長いため，キャッシュはその間待たされてしまう．これではキャッシュの高速性が活かされない．この解決法としては，主記憶への書き込みは，その要求だけを出し，別のハードウェアが書き込み作業を行っている間にCPUは処理を先に進めてしまうという方法（置いてきぼり制御）がある．しかし，この制御方法は複雑なハードウェアを必要とし，またCPUからの書き込みアクセスが連続した場合などの制御も複雑になるため，コストが高くなる．

## 6. 仮想記憶

大規模なプログラムを実行するためには，大容量の主記憶が必要となる．かつては主記憶容量以上の記憶領域を必要とするプログラムをなんとか実行しようとする場合，オーバレイという技術が用いられた．これは，処理すべきプログラムが主記憶に入り切らない場合，現在可能なデータ処理を終えた後で次に処理すべきプログラムをプログラム自身がディスク装置のような二次記憶装置から主記憶に読み込んでその後で処理を継続する，というような一連の処理をプログラム中に組み込んでおく手法である．このような作業を繰り返すことによって，「見かけ上の記憶容量」を増やすことができる．

実際のオーバレイは，プログラムの配置や処理手順のスケジューリングなどを細かく考えなければならず，非常に面倒で高度なプログラミング技術を必要とする．**仮想記憶（Virtual Storage）**は簡単に言えば，オーバレイのこういった繁雑な処理をOSの管理下で自動化し，プログラマにそれを意識させずにすませる技術である．

さらに，ここではプログラムに見えるアドレス空間は**論理アドレス空間**と呼ばれ，プログラムではそれがどのように実際の主記憶に対応しているかを意識しないですむ．こうして，プログラムは仮想記憶システムが許すかぎり，主記憶容量をはるかに越えるような大きなアドレス空間を利用できる．また，たとえば主記憶の大きさだけが異なる複数の計算機間でプログラムを共有することもきわめて簡単になる．さらに，ユーザ・プログラムの利用するアドレスは全てOSの管理下にあるので，複数のユーザ・プログラム同士が互いに邪魔をすることのないような管理も可能である．このように，仮想記憶システムは現在のマルチユーザ環境の計算機には必須の技術となっている．

## 7. 仮想記憶の原理と仕組

仮想記憶の原理は単純で，基本的には主記憶上の記憶領域が不足したら，現在使用されていない

主記憶の一部を二次記憶に追い出し，主記憶上に新しい領域を確保する，というものである．同時に，もし二次記憶に追い出されている領域がプログラムによってアクセスされたら，その部分を再び主記憶上にもってくる（その際はそれまで主記憶のその部分にあったデータが二次記憶に追い出される）．主記憶の内容を二次記憶に追い出すことを**スワップ・アウト**，二次記憶の内容を主記憶に戻すことを**スワップ・イン**という．

仮想記憶システムはハードウェアとソフトウェアの組合せで実現される．ある記憶領域をスワップする際，その領域の決定や実際の二次記憶装置への命令発行などはソフトウェア（OS）によって行われる．しかし，そもそもアクセスされたアドレスが主記憶上に存在しているかどうかというチェックは，ユーザプログラムの命令が実行されるたびに行わなければならないため，ハードウェアの支援が必要となる．このために MMU (Memory Management Unit：記憶管理装置）と呼ばれるハードウェアが CPU と主記憶の間に設置され，CPU から出されるアクセス要求に対するチェックを行い，必要に応じてアドレスの変換や CPU への割り込みなどを行う（**図-5**）．

仮想記憶システムでは，ユーザ・プログラムが実行されているアドレス空間を**論理アドレス空間**（**Logical Address Space**）と呼ぶ．そして，そのアドレス空間上のアドレスは，そのまま主記憶のアドレスに対応するのではなく，MMU によって主記憶上の本当のアドレスに変換される．主記憶上の本当のアドレス空間を**物理アドレス空間**（**Physical Address Space**）と呼ぶ．このアドレス変換作業が MMU の最も大切な仕事である．このために，MMU は論理アドレスと物理アドレスの変換表をもっており，CPU からのメモリ・アクセスのたびにこの表を使って変換を行う．もし対応する物理アドレスが表に見つからなかった場合，それはその論理アドレスが現在主記憶上にない（スワップ・アウトされている）ことを意味するので，割り込みによってそれを OS に知らせる．OS は MMU に問い合わせて，どの論理アドレスがアクセスされたかを調べ，そのブロックを二次記憶からスワップ・インする．そして，MMU のアドレス変換表を書き換え（今アクセスされた論理アドレスは新しい物理アドレスに対応し，逆にスワップ・アウトされたアドレスに対しては，「存在せず」が書き込まれる），プログラムの処理を続行する．

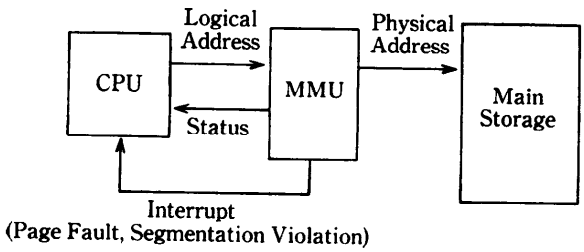



**図-5** MMU と仮想記憶

## 8. 仮想記憶の実現方法と記憶管理

仮想記憶を実現するうえで最も重要なことは，スワップ・イン/アウトや記憶管理をどのような単位で行うかである．これは，仮想記憶の使い方に大きく依存する．たとえば，複数のユーザ・プログラムを同時実行するために仮想記憶を用いるのであれば，プログラム単位でスワップを行うことも考えられるが，一つのユーザ・プログラムの記憶容量を増やそうとするならば，これではうまくいかない．そこで，現在多くの仮想記憶システムでは，ページング及びセグメンテーションと呼ばれる2種類の基本的な記憶管理方法を用いている．また，これらを組み合わせた方法も存在する．以下，これらについて順に説明する．

### 8.1 ページング

主記憶と二次記憶の間のスワップを行う単位の最も極端な例は，それをバイト単位で行うことである．しかし，スワップを行うべきアドレスを決定したり，実際にディスク装置を起動したりする時間を考えると，これはあまりにもオーバヘッドが大き過ぎて実用的でない．そこで，システムで固定の適当な大きさのブロックを設定し，そのブロックの単位でスワップを行うのが一般的である．このブロックを**ページ**と呼び，ページを単位とした記憶管理を行う仮想記憶の方法を**ページング**と呼ぶ．

**図-6** にページングによる仮想記憶の原理を示す．一つのプログラムのメモリ領域は複数のページに分割される．論理アドレスはページ番号を示す部分（p）と，ページ内でのアドレスを示す部分（d）に分けられる．MMU ではページを単位とし

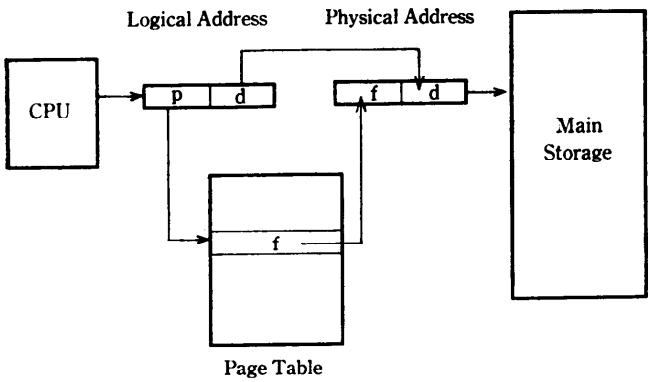



図-6 ページングによる記憶管理

たアドレス変換表をもち（ページ表），与えられた論理ページ番号を主記憶上の物理ページ番号（f）に変換する．そして，物理ページ番号とページ内アドレスを合成したものを物理アドレスとする．もし論理ページ番号に対応する物理ページ番号がない場合は，そのページがスワップ・アウトされていることを示す．この場合，MMU から割り込みがかかり，OS に知らされる．これを**ページ・フォールト**と呼ぶ．OS ではまず，物理ページの中から一つを選び，その内容をディスクに書き出す．そして，ページ・フォールトを起こしたページをディスクから読み込み，最後にページ表の書き換えを行う．

実際の OS では，ページングにおけるディスクへの読み書きをできるだけ減らすため，ワーキングセットという概念を用いて処理の効率を上げている．これは，スワップ・アウトされるページをすぐにはディスクに書き込まず，とりあえずメモリ上に保存し，近い将来再びアクセスされたときにすぐに取り出せるように管理しておく方法である．OS は Modified Page List および Free Page List という二つのリストを用意する．あるページをスワップ・アウトする際，もしそのページにこれまでになんらかの書き込みがあったならば（dirty ならば）前者のリストへ，そうでなければ後者のリストへ追加される．このページはまだメモリ上に残ってはいるが，MMU 上ではそのページへのアクセスは禁止される．Modified Page List 中のページは，捨てられる際にはディスク上に書き込まれなければならないが，Free Page List 中のページはディスクの内容と一致しているため，そのまま捨てられても構わず，他のページによってそのまま再利用可能である．

新しいページが必要になった場合，もしそのページがこれらのリスト中に残っていれば，それをそのまま利用する．そうでなければ当該ページはディスク上にあるため，まず Free Page List からページを一つ取り出し，そこにディスク上のデータを読み込んで，MMU の表を変更し，プログラムの実行を継続する．また，Modified Page List が一定の大きさより大きくなったり，Free Page List が一定の大きさより小さくなった場合，前者のリスト中のページをディスクに書き出し，それを Free Page List に登録する．こうして OS は常に一定数以上の Free Page を確保し，新しいページが必要になった場合に素早くそれを提供できるように管理する．このような管理の下で，ディスクへのアクセスをできるだけ少なくし，しかも新しいページを高速に用意できるようになる．

実際の論理ページの数はかなり多いので，ページ表は MMU の中に収まらず，システムが管理する主記憶上の特別な場所に置かれている．しかし，メモリ・アクセスのたびに毎回，主記憶上のページ表を見ていたのでは，速度が極端に低下してしまう．そこで，これを高速に処理するために MMU 内にも比較的小さなページ表をもち，頻繁にアクセスされる論理ページの情報を置いておく．そしてその中で論理ページ番号が見つからなかった場合のみ，主記憶上の本当のページ表を見にいく．つまり，MMU 内の変換表は一種のキャッシュとして利用されるわけである．この変換表を **TLB (Translation Lookaside Buffer)** と呼ぶ．

TLB には全ての論理ページが登録されているわけではないから，論理ページ番号から物理ページ番号が直接引き出せるような機構が必要となる．これは一種の連想メモリ (Associative Memory) である．連想メモリとは，普通のメモリのようにアドレスとそれに対応するデータからなるのではなく，二つのデータを対にしたデータの組の集合からなるメモリである．そして，一方のデータを与えると，それに対応するもう一方のデータが出力される．この場合，論理ページ番号と，そ

れに対応する物理ページ番号を対にして記憶しておけばよい．連想メモリは非常に高価なハードウェアなので，TLBの容量はそれほど大きくすることができない．そこでキャッシュ化して用いることにより利用効率を上げるわけである．

### 8.2 セグメンテーション

セグメンテーションは厳密に言えば，仮想記憶というより記憶管理の手法の一つであるが，ページングと関連づけて用いられる場合も多く，重要な概念なのでここで紹介しておく．ページングではページを単位とした記憶管理を行っていたが，プログラムを実行する際の記憶領域の使用法を考えると，それらは幾つかの論理的な集合，たとえばプログラムの命令が格納されている領域とデータが格納されている領域に分けられる．さらにデータ領域は静的なデータ，スタック，動的なデータなどに分けられる．そこで，これらを一つの単位とした記憶管理が考えられる．ここではこういう論理的な領域の単位を**セグメント**と呼び，これに基づく記憶管理の方法を**セグメンテーション**と呼ぶ．

**図-7** にセグメンテーションにおける記憶管理の原理を示す．一つのプログラムは複数のセグメントをもち，同一セグメント内のアドレスは連続しているが，異なるセグメントは異なるアドレスに割り当てられる．CPU から出されたアドレスはセグメントを示す部分(s)と，セグメント内でのアドレスを示す部分(d)に分けられる．セグメント番号によりセグメント表が引かれる．セグメント表にはそのセグメントの主記憶上での先頭アドレス (base) とセグメントの長さ (size) が記されている．セグメントの長さはセグメントごとに異なり，これをはみ出した場合は割り込みによって OS にエラーが通達される．これをセグメンテーション・バイオレーションと呼ぶ．

セグメント表にはこのほかにも，そのセグメントが書き込み可能か（プログラム本体は通常は書き込み不可能となっている），そのセグメントを使用するプロセスの番号は何番か，などの情報をもっている．これらを利用することにより，誤った記憶領域への書き込みによる破壊や，ユーザ・プロセス間のアクセス保護などが実現できる．

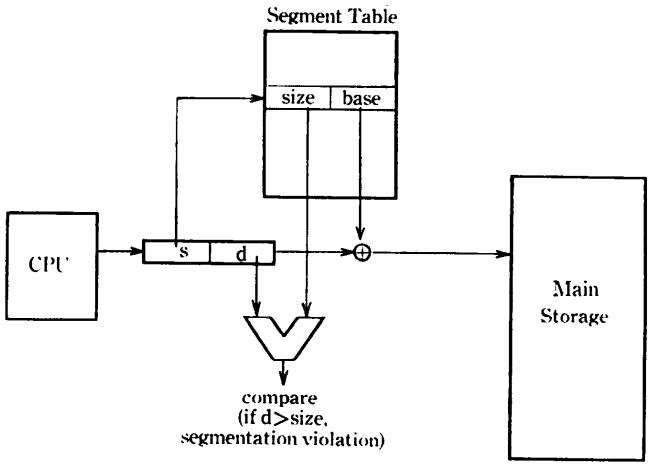



**図-7** セグメンテーションによる記憶管理

### 8.3 ページングとセグメンテーションの組合せ

ページングとセグメンテーションは基本的に独立した概念であり，その両者を適当に組み合わせることが可能である．これらの幾つかを簡単に紹介する．

最もよくみられるのは**ページ化セグメンテーション**である．セグメンテーションはスワップの対象ではなく，記憶管理の対象であり，実際にセグメンテーションの上に仮想記憶を実現するためには，各セグメントをさらにページングし，ページごとにスワップを行う必要がある．これがページ化セグメンテーションである．ここでは，セグメントの表の中には実際の主記憶上のアドレスの代わりに，そのセグメントのページ管理をしているページ表の先頭アドレスが記されている．セグメント内アドレスはこのページ表を用いて物理アドレスに変換される．

もう一つの組合せは**セグメンテーション・ページング**である．ページングにおけるページ表は基本的に全ての論理ページの分だけ用意されなければならないが，このアドレス空間が広いと非常に巨大なページ表が必要となる．多くの場合，論理アドレス空間の一部が使用されるだけなので，これはかなり無駄である．そこで，ページ表自身のセグメント化を考える．ページ番号はさらにセグメント番号とセグメント内アドレスに分けられ，セグメント番号でセグメント表を引くことに

よりそのセグメント，すなわち一つのページ表の大きさと，その先頭アドレスが得られる．そして，先頭アドレスにセグメント内アドレス（そのページ表内でのページ番号）を加えることによって本当のページ表のエントリが決定される．

どちらの方式も複雑なハードウェアを必要とするが，ページ化セグメンテーションにおいては現在走行中のプログラムのセグメント表の大きさはそれほど大きくなく，セグメントが決定した後は通常のページング処理となるため TLB の有効利用が可能である．このため，この方式は非常に多くのシステムで実用化されている．

### 8.4 ページの置換

ページングではページのスワップを行う際，二次記憶に追い出すページを決定しなければならない．これはちょうど，キャッシュにおいて主記憶に追い出すブロックを決定しなければならなかったのに似ている．したがって，ここでもキャッシュと同様に FIFO, LRU などのアルゴリズムが考えられる．基本的概念はまったく同じなので詳細については省略する．

### 8.5 記憶管理と多重プログラミング

仮想記憶を用いたシステムでは，ユーザ・プログラムによるメモリ・アクセスは全てなんらかのアドレス変換およびチェックを経て主記憶にアクセスされる．したがって，複数のユーザの複数のプログラムが同時にシステムを利用しているような状況(多重プログラミング)で，あるユーザのプログラムが他のユーザのプログラムの記憶領域をアクセスしようとしても，それは不可能である．なぜならば，通常 OS は各プログラムごとに完全に独立なページおよびセグメントを割り当て，ページ表およびセグメント表には現在走行中のプログラムでのみ有効なアドレス変換表だけが使用されるようになっているからである．こうして，このようなシステムではユーザ・プログラム間の記憶領域の保護が実現されている．

また，これとは逆に別々のプログラムが同一のページまたはセグメントをアクセスできるように MMU を設定することも可能である．たとえば，非常に頻繁に用いられる共有プログラムがあった場合，だれかがそのプログラムを使用して主記憶上にプログラムの本体が置かれているとする．別のユーザがそのプログラムを実行しようとしたとき，すでにそのプログラムは主記憶上にあるのだから，わざわざ二次記憶からもってくる必要はない．こうして，プログラム起動に要する時間が節約できる．こういう技法を共用ページ，共用セグメントと呼ぶ．

## 9. おわりに

本稿ではキャッシュおよび仮想記憶に関する基本的な概念と幾つかの実装方法について簡単に述べた．紙数の都合上，詳細な解説や実装上の技術的問題，各方式の評価，さらに各種の関連する話題(キャッシュの階層化，キャッシュと仮想記憶の関係など）について述べることはできなかった．本稿が特に初心者の読者諸氏の記憶システムに対する理解の助けとなれば幸いである．

## 参考文献

1) 高橋 茂：ハードウェア工学概論，共立出版 (1988).
2) 富田眞治，村上和彰：計算機システム工学，昭晃堂 (1988).
3) Peterson, J.L. and Silvershatz, A.: Operating System Concepts, Addison-Wesley Pub. Co. (1985).

(平成4年8月19日受付)

**朴 泰祐**（正会員）

1960 年生．1986 年慶應義塾大学大学院工学研究科修士課程修了．1990 年同大学院理工学研究科後期博士課程修了．工学博士．慶應義塾大学理工学部物理学科助手を経て現在，筑波大学電子・情報工学系講師．超並列計算機アーキテクチャおよび並列処理言語・アルゴリズムの研究に従事．