■7 群（コンピュータ -ソフトウェア）－3 編（オペレーティングシステム）

5 章 メモリ管理

（執筆者：吉澤康文）［2013 年 2 月 受領］

■概要■

コンピュータの歴史を省みると（特に）主メモリ不足の時代が 1980 年代まで約 40 年間続いた．この間の創意工夫により，現在は一般的に利用されている仮想記憶が発明されるに至った. 本章では多重プロセッシングを可能とするメモリ割付け方式の発展と問題点を説明し，ページ化されたメモリの出現とプログラマに記憶容量を意識させない仮想記憶方式について解説する．

【本章の構成】

多重プログラミングを可能とするメモリ管理に必要となる基本的な考えを示し，初期の方式での課題を詳細に説明する（5-1 節）．次に，多重プログラミングを実現する具体的な方式を説明し，各々における問題点を明らかにする．同時に，主記憶サイズを超えたプログラムを実行する方法と問題点を説明する（5-2 節）．これらの方式や工夫は本質的な問題解決にはならず，ページ化されたメモリならびにメモリの論理化という発明がなされ，ハードウェアならびに OS の発展により仮想記憶が出現する．この動作原理を説明する（5-3 節）．

■7群 － 3編 － 5章

5-1 基本的な考え方

（執筆者：吉澤康文）［2013年2月 受領］

5-1-1 多重度の向上をめざして

コンピュータが誕生して以来，メモリが性能上のボトルネックになっており，ハードウェア，ソフトウェアのエンジニアは多大の努力を重ねてきた．半導体記憶が1970年代に利用可能となったが，メモリ容量の制約は1980年代後半まで続いた．ここでは，コンピュータの性能向上に対し，メモリが不足していた時代の創意工夫について説明する．大容量記憶が利用可能な状況になったときの創意工夫については，ファイル管理で説明したようなシステムバッファやRAMファイルなどがある．

メモリ不足とは，CPUの性能向上に比べると利用可能なメモリが不足しているということを意味している．つまり，コンピュータの性能向上のボトルネックが，主としてメモリ不足にあり，CPUを100％利用できないのである．具体的にいうならば，多重プログラミング環境におけるジョブ多重度を高めることができず，実行可能なプロセスが少ないため，CPUが遊んでしまうということである．したがって，目標は，いかにして多重プログラミングの多重度を高め，CPUを100％使用できるかという一点にある．

5-1-2 パーティション

多重プログラミングは1960年代から考案されていた．多重プログラミングを行うには，複数のプログラムを主記憶内にローディング（Loading）しておく必要がある．**図5･1**は最も単純な多重プログラミングを行う際のメモリ分割使用方法である．各プログラムの実行はジョブ（Job）と呼ばれており，各ジョブにメモリ領域が与えられる．このメモリ領域をパーティション（Partition）と呼ぶ．

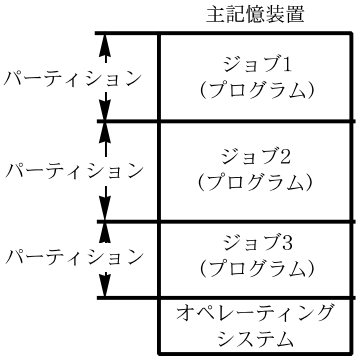

図5･1　多重プログラミングにおけるメモリパーティション

一般的に，ジョブ実行（ジョブ投入ということがある）に際していくつかの資源が必要である．その第1はジョブの要求するメモリ領域であり，第2はジョブが独占的に使用する入出力装置である．パーティション方式では，ジョブの要求する必要十分なサイズを備えたパーティションを探し，割り当てることになる．

5-1-3 メモリの節約

限られた主記憶を可能な限り有効に利用することがメモリ管理の基本的な考え方である．コンピュータ内には，メモリ割付け方法にもよるが未使用領域が必然的に生まれてしまう．ここでいう「未使用領域」とは，正確には「CPU から参照されることのない領域（Unreferenced Area）」という意味である．それらの領域は以下の部分にあり，メモリ節約の問題(a)，(b)，(c)と呼ぶことにし，**図 5･2** に示す．

(a) パーティション内
(b) ジョブ要求メモリ領域内
(c) プログラムやデータ領域内

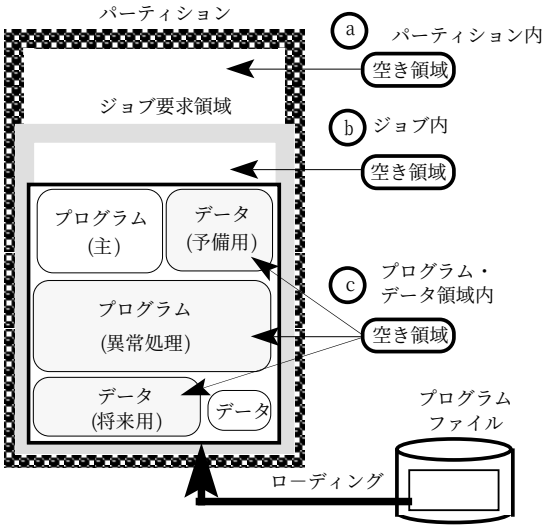

図 5・2　パーティション内に存在する未参照領域

上記(a)の部分は，ジョブ開始時にジョブの要求を満たすサイズをもったパーティションが選択されるので，固定長のパーティションの場合は，その中に必然的に空き領域が生じることになる．この領域はジョブ実行過程において CPU から参照されることはない．

次に，(b)の部分が生じる理由を説明する．ジョブの要求するメモリサイズは，プログラマが自分のジョブを実行するために必要十分な大きさを指定するために，一般的には少し余裕をもった値となる．したがって，実行するプログラムをローディングするとジョブ要求領域内に空きが生じることになる．

最後の(c)の未参照領域はプログラムやデータ内に存在する．プログラムの実行過程は外部から渡されるパラメータなどにより変化するのが一般的である．図 5･2 に示した例は，OLTP（On Line Transaction Processing）を仮定しているが，そのようなソフトウェアでは，信頼性確保のために異常処理のプログラムを多く含んでいる．もちろん，それらのプログラムはほとんどのトランザクション（取引）処理においては使用されない．しかし，ハードウェア，ソフトウェアに異常状態が生じたときのために常時主記憶内にプログラムをローディングし

ておかねばならない．

同様に，実用化されているソフトウェアには新しい要求が生まれ，適用規模の拡大も伴うのが常である．したがって，ソフトウェア設計者は現時点の要求を満たすだけでなく，将来を見通してデータ領域を大きく確保し，また将来予定されている機能拡張のためのデータ領域をあらかじめ確保している．

このように，プログラムならびにデータ領域は通常の処理においては未参照な領域を含んでいるのが一般的である．メモリ管理の究極の目的は，上記 3 種類に分類された未参照領域のすべてをジョブから取り上げて，多重プログラミングの多重度向上に寄与することである．

5-1-4 記憶保護機構

多重プログラミングを実現するためにメモリ管理と密接な関係にあるのは，メモリ保護機構である．ここでは，最も基本的な記憶保護機構について説明する．

図 5･2 に示すように，多重プログラミング環境では，主記憶を複数のジョブがパーティションごとに分割して共用している．このため，各ジョブに与えられた領域（空間と呼ぶこともある）以外の参照は禁止されねばならない．特に，ジョブに与えられていない空間に対して書込みを行うと，書き込まれた領域のジョブが正しく動作する保障が失われる．最悪のケースは，オペレーティングシステムの領域に書込みがなされる場合であり，ときにはシステムダウンにつながる可能性が高くなる．

そこで，**図 5･3** に示すような記憶保護機構が初期のコンピュータには採用されていた．この機構では，プロセス A がディスパッチされる際に，プロセス A の領域外に対する書込みを禁止するためにメモリ保護境界レジスタ（Boundary Register）を設定する方法である．このレジスタにより，プロセス A が自分の領域外に対してストア（Store）系の命令を実行すると記憶保護例外の割込みが生じ，不当な書込みが防止されることになる．

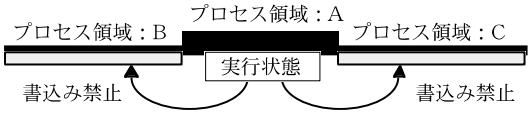

図 5・3　境界レジスタ方式による記憶保護方式

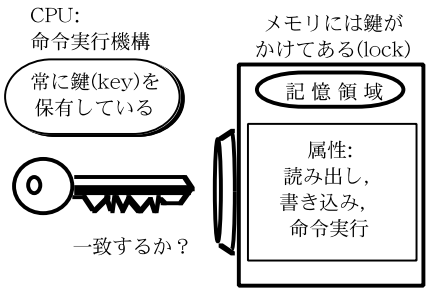

図 5・4　キーロックの一致による記憶保護方式

仮想記憶機構と共に利用されることの多い記憶保護機構がある．この方法では，主記憶を一定のサイズに区切った単位で取り扱う．そのメモリサイズの単位をページと呼び，各ページのメモリ保護を行う．ここでの基本的な考え方は，**図 5･4** に示すように，すべてのページに鍵（ロック：lock）を掛けておき，CPU が命令の実行過程でページに掛けられた鍵を CPU のもつ鍵（キー：key）で開けながらメモリ書換え，参照を行うという機構である．具体的には，ページは 4 KB や 8 KB のような 2 のべき乗であることが多い．そして鍵の値には 2 あるいは 4 ビット程度が使用され，使用するページに読込み，書込み，命令実行などの許可属性を付けることができるようになっている．

■7群 － 3編 － 5章

5-2 多重プログラミング実現法

（執筆者：吉澤康文）［2013年2月 受領］

5-2-1 静的分割方式

多重プログラミングを実現するためには主記憶内に複数のジョブ領域を確保する必要がある．最も簡単な実現方法は，**図 5･5** に示すようにシステムが起動した直後に固定的なパーティションを作成しておき，システムの運用中はその固定的なパーティションを保持することである．この方法では，ジョブをディスパッチするときに記憶保護のための記憶保護境界レジスタ値を設定するだけでよい．

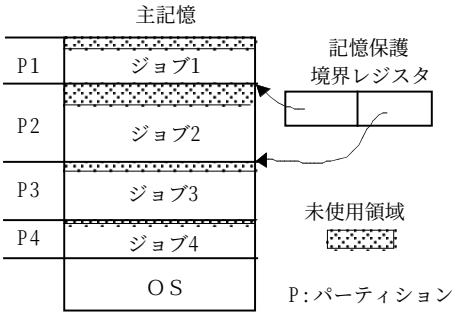

図 5･5　静的分割方式

本方式を静的分割方式（Static Partitioning）と呼ぶ．利点は容易に多重プログラミングが実現できることにある．しかし，メモリ管理の目標とするメモリ節約の問題(a)，(b)，(c)（パーティション内，ジョブ要求メモリ領域内，プログラムやデータ領域内に存在するメモリ未参照領域を排除すること）のどれひとつも解決できない．

そればかりでなく，パーティションが固定的であるため，大きなメモリを必要とするジョブの実行が困難となる．例えば，図 5･5 において，パーティション 2 と 3 を合わせた大きなパーティションを作り，大きなジョブを実行しようとしても，そこで実行しているジョブ 2 とジョブ 3 がいつ完了するか不明であることや，一方がジョブ完了してももう片方のジョブが完了するまであてもなく待たねばならないことになり，メモリを使用できない時間帯が生まれてしまう．更にいうならば，このようなパーティション再構成作業がオペレータの勘や熟練度に頼ることになり，安定的でないという欠点がある．

5-2-2 動的分割方式

静的分割方式では多重プログラミングは実現できるがメモリ管理の目標であるメモリ節約を全く達成できない．そこで，ジョブが要求するメモリ容量と一致するパーティションを実行時に作成する方式として動的分割方式（Dynamic Partitioning）が提案された．

本方式では，ジョブの要求する領域サイズを満たす空き領域を探して割り当てる．**図 5･6** にその様子を示す．ここでは，ジョブ 2 が完了した時点から開始している．ジョブが完了したときに隣接する空き領域があればそれらを併合して大きな空きのパーティションを作成す

る．ここに，少し大きなジョブ 5 が入ってきて，空いたばかりのパーティションが容量的に満たされるなら直ちに割付けを行う．

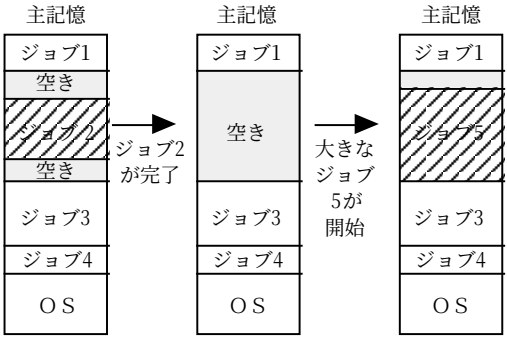

図 5・6　動的パーティション構成方式

この方式では，要求されたジョブ領域と全く同じサイズのパーティションを作成するために，パーティション内に無駄な領域が作成されることはない．ということは，メモリ節約の問題(a)を解決したことになり，メモリ管理としては進歩である．動的割付け方式は 1960 年代の後半における汎用大型コンピュータで利用され，その代表的な OS が IBM 社の MVT で

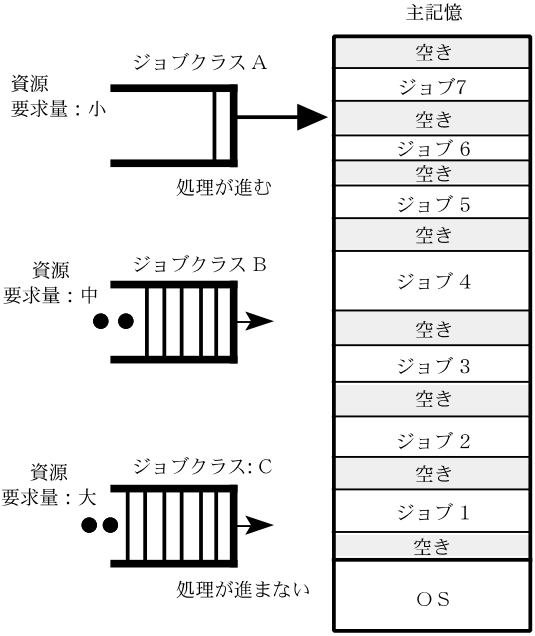

図 5・7　動的分割方式でのフラグメンテーション問題

ある．しかし，(b)，(c)の問題は解決されなかった．更に，動的割付け方式は以下のようなフラグメンテーションという新しい問題に直面し，そのために発展することはなかった．

当時，コンピュータの利用法は，ジョブをため込んで一括処理するバッチ処理が主流であった．そこでは，ジョブの要求する資源量（CPU 時間，要求するメモリ容量，プリンタへの出力行数など）に応じてジョブクラス（Job Class）を作り，ジョブ待ち行列を作る．**図 5･7** にジョブクラスとその待ち行列の様子を示す．

一般的にこのようなジョブクラスを設定する目的は，資源要求量の小さなジョブが多く存在するので，それらを優先して処理し，ターンアラウンドタイム（Turn Around Time）の短縮を図り，サービスを向上させることにある．しかし，このような運用を行うと，図 5･7 に示すように，小さなジョブクラス A の処理は進むが，ジョブクラス C のような資源要求量の大きいジョブはいつまでも実行されないことになる．その理由は，動的割付けにより小さなジョブが実行されると，その結果として小さな空き領域が主メモリ内に増えてしまうためである．このように，小さな空きパーティションがメモリ内にたくさんでき，利用できない領域となることをフラグメンテーション（Fragmentation）問題という．

5-2-3 大容量記憶への願望

プログラマにとって記憶容量の制限は設計上大きな制約になっていた．物理的なメモリ容量の上限が実行できるプログラムの限界サイズになっていたからである．大きなプログラムは一般的に，複数のサブルーチン（Subroutine）から作られている．それらのサブルーチンはいくつかがまとまって一つの機能の集団となっている．また，機能の集団はそれらが相互に独立に動いたり，選択的に使われたりしていることが多い．これらのプログラム構造を考察すると，大きなプログラムは必ずしもすべてが同時に主記憶内に存在する必要はないことが分かる．

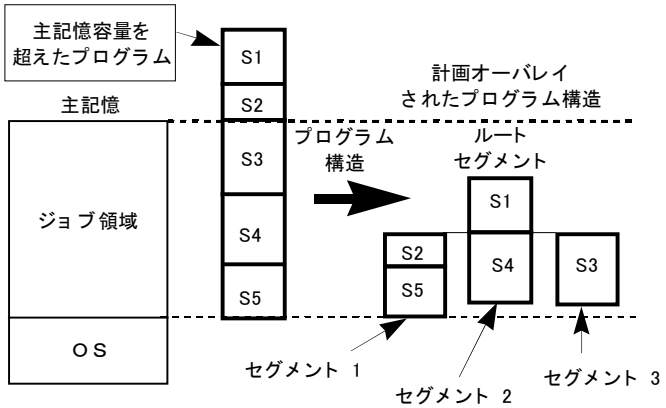

図 5・8　計画オーバレイによる主記憶サイズ以上のプログラム実行

例えば，**図 5･8** に示すような大きなプログラムがあり，このプログラムは S1, S2, ･･･, S5 までの機能集団からできている．このプログラムをよく見ると，S1 は主プログラムであるの

で，常に主記憶に存在しなくてはならないが，それ以外のプログラムの動作を分析すると，必ずしも同時に主記憶に存在しなくてよい場合がある．この例でいうと，S3 と S4 は同時に存在する必要がない．S2 と S5 は同時に存在する必要があるが，しかし，それ以外のプログラムと同時に存在する必要はない，などである．

このように，プログラムが主記憶上に同時に存在する必要性を分析すると，図 5･8 の右に示すような木構造にしたプログラムの構造が考えられ，これを計画オーバレイ構造（Planned Overlay Structure）と呼ぶ．主記憶にローディングする単位をセグメントと呼び，主記憶に常駐して各セグメントを主記憶にローディングするために制御をする部分をルートセグメントと呼んでいる．ルートセグメントは，プログラムの進行に合わせてセグメントを主記憶に読み込む制御を行う．このとき，OS はルートセグメントのプログラムに対して，オーバレイを可能とするシステムコールを提供しなければならない．

計画オーバレイ方式は大きなプログラムの実行に有効で，1960 年代の後半にはトータルバンキングシステムに代表される OLTP や，巨大な計算を行う科学技術計算プログラムなどに多く使用されることになった．しかし，ソフトウェアは利用されている限り機能拡張がなされるのが一般的宿命であり，そのたびにオーバレイ構造の妥当性を検討しなければならない．つまり，セグメント長の見積りや，最終的にできあがったセグメントが図 5･8 のジョブ領域に入るか否かの検討などである．この作業には，プログラムの保守拡張という本来の作業以外の工数を要するという欠点があり，それを無視できなくなってきた．このための抜本的な解決が長く望まれていたが，それは仮想記憶の出現まで待たされることになる．

■7群 － 3編 － 5章

5-3 ページング機構

（執筆者：吉澤康文）［2013年2月 受領］

5-3-1 従来のメモリ割当ての制約

多重プログラミングの多重度向上はOSの使命である．5-1-3項にて述べた動的分割方式ではメモリ節約の(a)の問題は解決したが，(b)，(c)の問題は未解決であった．また，フラグメンテーションという問題に直面し，その解決法がハードウェア，ソフトウェア共に行われてきたが必ずしも成功はしなかった．

フラグメンテーションは深刻な問題で，避けて通れない問題である．この問題の本質的な原因は「ジョブに対する記憶割付けは連続的な領域でなくてはならない」という制約にある．つまり，ジョブの要求するメモリ領域は常に連続的に割り当てねばならないという条件である．もし，この条件がなければ，主記憶上にちらばった空き領域を寄せ集めて割り当てることができるので，多重度の向上が図れることになる．

5-3-2 ページ化された記憶

そこで，最初から主記憶を一定の小さな単位（これをページ：Pageと呼ぶ）に分割しておき，主記憶の要求に対して任意の未割当てのページを必要なだけ割り付ける，という今までとは発想の異なるコンピュータが出現した．これが，1960年英国マンチェスタ大学とFerranti社が共同開発したATLASコンピュータであり，M. V. Wilkesらによって開発されたのである．

ページ化されたメモリのアイデアをまとめると以下の三つである．

(a) メモリを一定のサイズ（ページ）に区切る．
　これによりメモリ内の空き領域をページサイズよりも小さくできる．

(b) アドレス変換テーブルを用意する．
　物理的に離れているページを論理的に連続領域にする．

(c) アドレスの論理化を行う．
　論理アドレスを物理アドレスに変換する．

上記のアイデアを理解するために，**図5･9**にページ化されたメモリの概念的なモデルを示す．ページ化されたメモリにおいても，プログラマには連続したアドレス領域（空間）が与えられる．プログラマは従来とおり，一次元に連続したメモリが存在すると仮定してプログラミングすればよい．このように，物理的な主記憶と切り離して記憶領域をプログラマに提供するので，各プロセスに与えられたアドレス領域を論理アドレスと呼ぶことにする．この領域には，テキスト＜プログラムの手続き部分＞，データ領域，スタック領域などが存在する．ここではページサイズを4KB（4,096バイト）とする．

図5･9において，プロセスAが必要とするメモリは18KBとする．その内訳は，テキストが8KB，データが4KB，そしてスタック領域が6KBである．プロセスAに割り付けられた論理アドレスを4KB単位に区切って考える．すると，テキストは，A0，A1なる各々4KBの内容であり，データはA2の4KB，そしてスタックの領域はA3，A4となる．

ここで，ページ化されたメモリにこれらのテキスト，データ，スタックが割り付けられている様子を見る．論理アドレスの0番地に位置しているテキストのA0は，物理アドレス

の 4 K 番地（正確には 4,096 番地）に内容がある．この 4 K 番地のアドレスはアドレス変換テーブルの最初のエントリに書き込まれている．そして，テキストの第 2 ページの内容 A2 は物理アドレスの 12 K 番地（49,152 = 12×1,024）に存在し，12 K 番地がアドレス変換テーブルの第 2 エントリに書かれている．

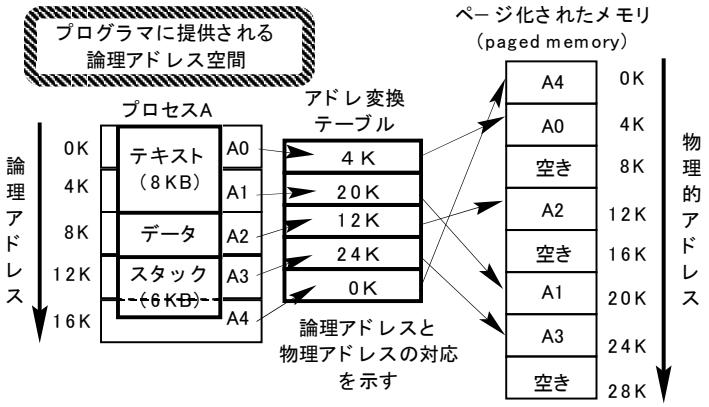

図 5・9　ページ化された主記憶を連続領域に変換するアドレス変換機構

上記のように，アドレス変換テーブルの第 1 エントリは論理アドレスの第 1 ページに割り付けられた物理アドレスを示し，第 2 エントリは論理アドレスの第 2 ページに割り付けられた物理アドレスを示すようにする．つまり，論理アドレスに対応してアドレス変換テーブルのエントリに物理アドレスを書き込んでおけばよいのである．これで，不連続な領域を物理アドレス上に割り付けてもアドレス変換テーブル上で論理アドレスの連続性が保てることになる．

ページング機構でのメモリ管理は，メモリ要求に対して以下の処理を行う．

(i) 要求されたメモリサイズからページ数を計算する．
(ii) 空き状態にあるページ数を確認する．
(iii) 必要となるページ数のエントリをもつアドレス変換テーブルを確保する．
(iv) 空き状態のページを割り付け，各々の物理的なページアドレスをアドレス変換テーブルに書き込む．

以上の手順により主記憶上の不連続の領域をあたかも連続のように見せ掛ける準備ができる．そこで，次に，アドレス変換テーブルを基にして物理アドレスを求めるハードウェア機構を説明する．

5-3-3 アドレス変換機構

図 5･9 に示したように，ページの概念を導入したコンピュータでは，プログラマには論理的な連続したアドレス空間を提供するが，実際に割り付けられる主記憶空間は不連続である．そこで，論理アドレスを物理アドレスに変換する機構が必要になってくる．この役目を果たすのが動的アドレス変換機構（DAT：Dynamic Address Translator）であり，**図 5･10** にその機

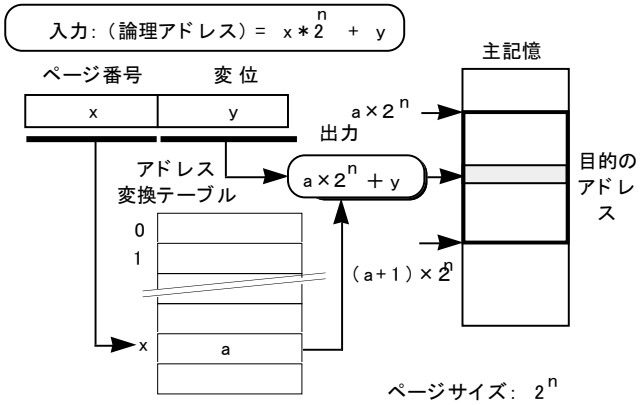

図 5・10　アドレス変換機構

能を示す.

DAT は CPU 内の機構であり，メモリにアクセスする際は常に呼び出される．例えば，命令を読み込む（フェッチ：Fetch という）などがその良い例である．このとき，DAT に対する入力は論理アドレスであり，DAT は論理アドレスを基にし，OS のメモリ管理が作成したアドレス変換テーブルを使って物理アドレスを求め出力とする.

DAT のアドレス変換過程は図 5･10 に示したとおりである．入力は論理アドレスである．ここではページサイズを 2 の n 乗とすると，（論理アドレス）= x ＊（ページサイズ）+ y と表すことができる. ここで, x をページ番号（Logical Page Number）と呼び, y を変位（Displacement）という.

x をアドレス変換テーブルのエントリ番号（インデックス）として DAT はアドレス変換テーブル内の主記憶ページ番号（Physical Page Number）a を得る．a にページサイズを掛けたアドレスが物理ページアドレスである．したがって，ページ内の変位である y の値を加えることで目的のアドレスを得ることができる. このようなアドレス変換は CPU のメモリアクセスのたびに行われ，そのたびに論理アドレスから物理アドレスへの変換が行われるので，ダイナミックリロケーション（Dynamic Relocation）と呼ぶこともある.

5-3-4 ページングの利点と欠点

論理アドレスは命令の読出しやオペランド（Operand）へのアクセスのたびに必要とされる．したがって，今までになかった操作を CPU は行わなくてはならないので処理が遅くなる．また，アドレス変換テーブルのように，今までは不要であったメモリ領域も必要となる，などの欠点がある.

ページ化されたメモリ機構には上記のような欠点があるが，動的分割方式によるフラグメンテーションを解決する唯一の方法であること，また，明らかに不要となるメモリ領域も，1 ページより小さくなる，というようにメモリの節約を達成できる．また，アドレス変換を高速化する TLB（Table Look aside Buffer）機構のようなハードウェアの改善も進み，DAT の処

理時間の遅れを解消する技術が進んだのである．

ページ化されたメモリでは，プログラマの夢であったアドレス領域の制限を大幅に緩和する仮想記憶方式が発明され，更にメモリ節約の問題(a)，(b)，(c)のすべてを解決するオンデマンドページングが発案される（6 章 6-4 節）．すなわち，動的分割方式では(a)パーティション内の無駄使いは解決したが，(b)ジョブ要求メモリ領域内と(c)プログラムやデータ領域内に存在する無駄なメモリの解決が不可能であったので，その欠点を補って余りある利点が生まるのである．

■7 群 － 3 編 － 5 章

5-4 仮想記憶方式

（執筆者：吉澤康文）［2013 年 2 月 受領］

5-4-1 仮想記憶の原理

仮想記憶方式とは，現実には存在しない記憶領域をユーザに提供する機構である．つまり，実装されている記憶容量より大きな領域をコンピュータユーザに提供することになる．もしこのようなことが可能になるならば主記憶よりも大きなソフトウェアを実行することが可能になるばかりでなく，コンピュータシステムとしては多重プログラミングの多重度向上が図れることになり，生産性の向上につながる．

仮想記憶方式を実現するには，ページフォールト割込みを起こさせる新しいハードウェア機構が必要となる．この割込みはアドレス変換を担う DAT に追加・拡張された．拡張された DAT では，アドレス変換テーブル内にページフォールトフラグ（Page Fault Flag）フィールドがオンとなっているエントリを DAT のアドレス変換過程で見出したときに割込みを発生させる．

この機構を使うことにより，プログラムが命令実行過程で参照したページだけを割り付ければよいという発想が生まれる．そこでは，以下のような手順をメモリ管理は実行する．

(a) 要求されたメモリサイズからページ数を計算する．
(b) 必要となるページ数のエントリをもつアドレス変換テーブルを確保する．
(c) このとき，アドレス変換テーブル内のページフォールトフィールドをすべてオンとし，DAT が参照したときに割込みを発生させるようにしておく．

このような処理を行った直後の状態を**図 5･11** に示す．つまり，プロセス実行開始直後には主記憶のページを全く割り付けないのである．主記憶を割り付けていない状態で，OS はこのプロセスをディスパッチする．つまり，プロセスの実行する仮想的なメモリ領域は与えるが，実際のメモリは全く割り当てることなく CPU を割り当てるのである．

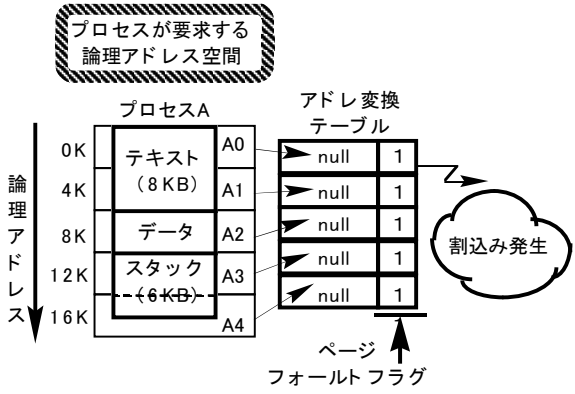

図 5･11　オンデマンドページングにおける初期のアドレス空間生成時

ディスパッチされると図 5･11 のテキスト（命令が存在する領域）が読み出される．つまり，

命令フェッチが行われるので，DAT が呼び出され，論理アドレスを物理アドレスに変換する動作が開始する．このとき，DAT はページフォールトフラグがオンのエントリを参照することになるので，ページフォールト割込みを発生させることになる．そして，制御はカーネルに渡る．

ページフォールトが発生して制御がメモリ管理に渡った時点で，初めて OS は以下の処理を行う．

(d) 未割当てのページを探して割り付け，物理的なページアドレスをアドレス変換テーブルに書込む．

(e) 必要に応じて，ページの内容をローディングする（テキストやデータなど）．

(f) ページフォールトフラグをオフにする．

この操作を行った直後の状態を示したのが**図 5･12** である．この図では，テキストの第 0 ページに対して主記憶の第 1 ページ（4,096 番地）が割り付けられている．そして，ページフォールトフラグはオフ(0)になっている．この操作を行った後に OS は再び当該プロセスをディスパッチする．すると，今度は主記憶が割り付けられているので，DAT はアドレス変換に成功し，前回の命令のフェッチを成功させ，その結果，命令が実行されることになる．ここで，プログラムに与えた論理アドレスのページを論理ページ，主記憶のページを実ページ（Real Page）と呼ぶ場合がある．また，上記の(d)に説明した空き状態のページのことを，ページフレーム（Page Frame）と呼ぶことがある．

次に，命令実行の過程で，例えばデータ領域やスタック領域を命令のオペランドとして参照するが，このとき再びページフォールトが発生することになる．ページフォールトの処理は，上記(d), (e), (f)の操作であり，該当する論理ページに対して同様の手順が繰返され，参照された論理ページだけに実ページが割り付けられてプログラムが実行されて行く．

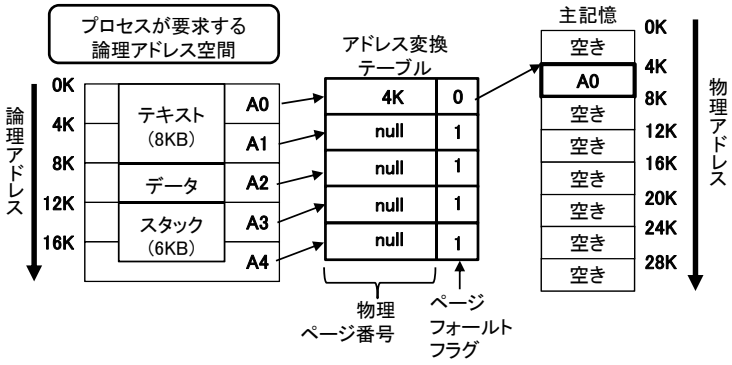

図 5・12　最初のページフォールトで割り付けられたページ

5-4-2　オンデマンドページング

上記のページ割付け方法をオンデマンドページング（On Demand Paging）方式という．これはプロセス（ジョブ）起動時のメモリ要求に対しては現実のページを全く割り付けず，命令を実行する過程で参照された部分だけに実ページを割り付ける方式である．

オンデマンドページング方式では, CPU の命令実行に伴って参照されるページだけが主記憶割付けの対象となる．したがって，メモリ節約の問題として残された二つの課題が解決したことになる．その課題とは，ジョブが要求したメモリ領域内，ならびに，プログラムやデータ領域内に存在する未参照領域（図 5･2 参照）に対する無駄なメモリの割付けのことである．

いま, プロセスが終了する直前のメモリ割付け状態が仮に**図 5･13** のようになっていたとするならば，上記に述べたメモリの節約が達成できたことになる．つまり，プログラム実行過程で未参照であったテキストの第 2 ページ(A1)とスタックの第 2 ページ(A4)の部分に対してページを割り付ける必要がなかった．つまり，従来のメモリ割当て方式では 5 ページを割り付けなければプログラムが実行できなかったのであるが, オンデマンドページング方式では, 3 ページの割付けだけでプログラムを実行できたことになる．

オンデマンドページング方式を別の角度から考察するために図 5･13 の例をもう少し詳しく考察する．プロセスの要求したメモリ容量を V とし，プロセスが完了するまでに割り付けられたメモリ量と R とするならば，参照する記憶容量の最大は V であり，プログラムが常に全領域を参照するとは限らないので，常に $V \geqq R$ の関係が成り立つ．つまり，オンデマンドページング方式では，プロセスの要求するメモリよりも少ないメモリ容量でプログラムの実行が可能となるのである．これは，主記憶よりも大きなプログラムを実行可能にすることを意味しており画期的な発明である．

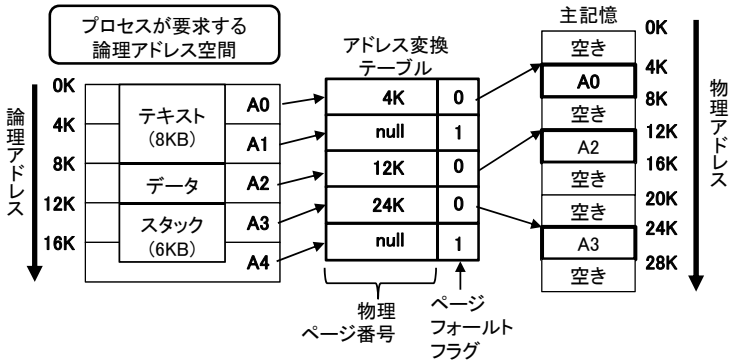

図 5・13　オンデマンドページングによるプログラム完了直前のページ割付状態

図 5･13 の例では，$V = 5$，$R = 3$ である．したがって，V/R の比を求めると，$R/V = 0.6$，この逆数は，$V/R = 1.67$ となる．これらの意味は，要求するメモリの 60 %の主記憶を用意するだけでプログラムが実行できることであり，逆の観点からすると，主記憶の 1.67 倍大きなプログラムを実行できたことを意味する．

仮想記憶を実現する際に，オンデマンドページング方式を採用する利点をまとめると以下のとおりである．

(1) 命令実行により参照されたページだけが割り付けられるので必要最低限のメモリ消費となる．

(2) 主記憶容量よりも大きなメモリサイズを要求するプロセスの実行が可能になる．

5-4-3 仮想記憶を支えるメモリ階層

オンデマンドページング方式ではページフォールトの割込み処理として，空き状態のページ割当てを行うが，その直後に，必要に応じてページの内容をローディングしなければならない．例えば，テキスト部分などは内容をロードする必要がある．このローディングの方法はOSの実現方式に依存しているが [*1]，ここでは，ページングファイル内に格納された内容をローディングする場合を考える．

一般的にこの操作をページイン（Page In）と呼び，仮想記憶の一部，あるいは全部が格納されているファイルをページングファイルとかスワップファイル（Swap File）と呼んでいる．通常は，磁気ディスクを利用することが多く，他のファイルシステムとは区別して扱うことがあるので別のパーティション（Partition）にしておくことが多い．

図 5･14 には，プロセス A の実行時の状態を示している．アドレス変換テーブルには第 2

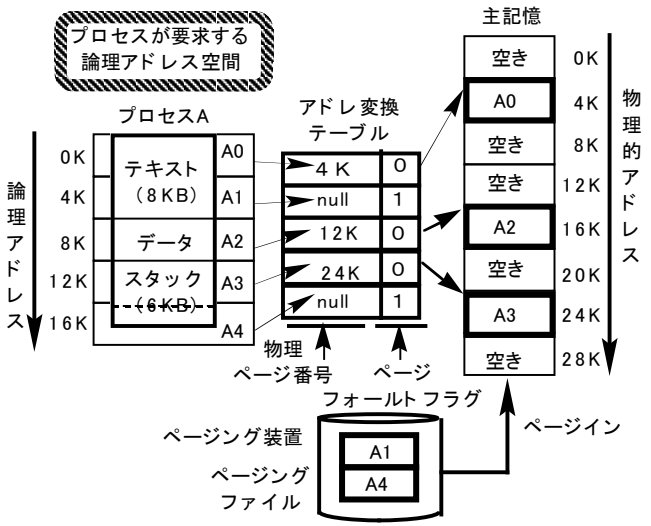

図 5・14　仮想記憶を支える 2 次記憶装置

と第 5 エントリにはまだ実ページが割り付けられていない．それらに該当するページ内容はページング装置の中に格納されている．

ページフォールト割込みが発生してページインするのは，ページの内容が存在する場合である．上記に述べたように，テキストはその代表である．また，内容が格納されているデータ領域もページインする必要がある．ページインが不要な領域の代表は，作業領域（Work Area）であり，プログラムの処理過程で動的に領域を確保する場合などが該当する．例えば，UNIX

[*1] 仮想記憶でのプログラムローディング：OS によっては，プログラムローディングの要求が発生したときにファイルシステムからすべてを読み込んでしまう場合がある．このときは，ローディングしたプログラムの内容はいったんは実ページ上にロードされているが，その後は，ページングファイルにページアウトなどで出力される．もう一つの方法は，ページフォールト発生時にファイルシステムからページフォールトを発生したページの部分だけをロードする方法がある．後者の方が仮想記憶に整合した方法である．

での malloc( )，共通メモリの shmget( ) などで確保した領域（Dynamic Storage Allocation）である．あるいは，プログラミング言語 C における自動変数の領域なども同じである．これらの領域は初期割当ての時点では内容がなく，ページフレームだけを割り付ければよい．

仮想記憶は主記憶とバッキングストア（Backing Store）としてのページングファイルで支えているという考えもあるが，逆の見方もある．つまり，コンピュータの提供する記憶はすべて仮想記憶であり，CPU がアクセスできるのは半導体メモリの主記憶なので，仮想記憶の一部をキャッシュ（Cache）して主記憶に読み込んでいるというモデルである．この考え方を**図 5･15** に示す．

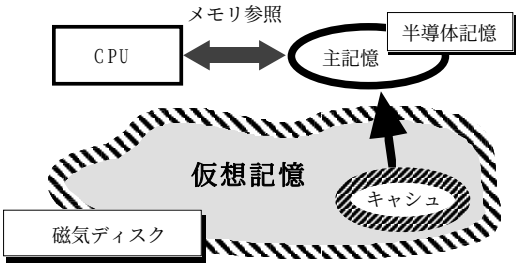

図 5・15　主記憶が仮想記憶を参照するキャッシュメモリである

5-4-4 ページングとスワッピング

オンデマンドページング方式ではページフォールトが発生した時点で空きのページフレームを割り当て，必要に応じて内容をページインする．つまり，仮想記憶は実記憶と磁気ディスクのような 2 次記憶により支えられているのである．この 2 次記憶をバッキングストアと呼ぶことがある．

実記憶容量以上の仮想記憶を実現しているとき，ページフォールトにより実ページを割り当てていくと，いつか実記憶が一杯になってしまう．このようなときは，実記憶領域から近い将来参照がないと思われる実ページを探し出し，解放する必要がある．このとき，実ページの内容は再び必要となる可能性があるので，補助記憶に書き込んでおく必要がある．この操作をページアウト（Page Out）と呼ぶ．

図 5･16 には実記憶と 2 次記憶の間でページ単位の転送を行うページイン, ページアウトの様子を示した．ページイン，ページアウトの操作は，基本的には各プロセス固有領域内の実ページを 1 ページ単位で個々に入出力することである．仮想記憶方式では，仮想的に大容量の記憶を提供し，多重プログラミングの多重度を向上することが可能であるが，後に述べるとおり，多重度を上げすぎるためにページングが多発してしまい，逆に性能低下に陥る可能性がある．このような状況に陥っていることを OS が判定すると，多重度を下げる制御機構が動作する．

プロセスの多重度を下げる最も有効な方法は，実行可能状態にあるプロセス数を少なくすることである．その手法として，プロセス空間をスワップアウト（Swap Out）する．スワップアウトとは，プロセスに割り付けられた全実ページをすべてバッキングストアに吐き出すことである．これにより，一度に多くの空きページを確保することが可能になる．したがっ

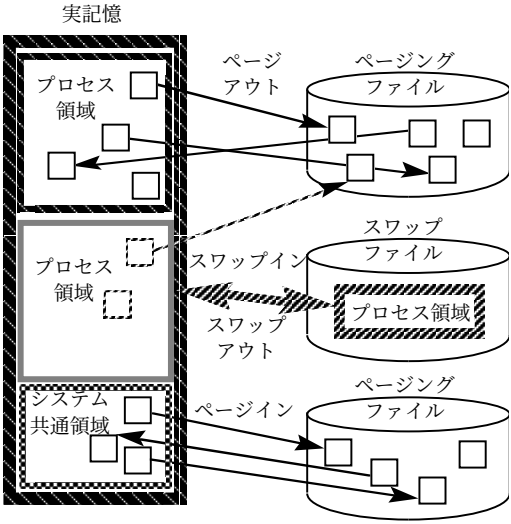

図 5・16　ページングとスワッピング

て，ページフォールトが発生しても実ページの割当て待ち状態が解消される．スワップアウトされたプロセスは，実行可能状態から「実メモリ空き待ち状態」になる．

逆に，スワップアウトされているプロセスが存在し，実記憶の空きページが増大し，ページフォールトの発生も少なく，CPU 利用率が低くなっている場合は，多重プログラミングの多重度が低いとみなされる．OS はそのように判断したときスワップアウトされているプロセスを主記憶内に読み込んで実行可能状態とし，多重度の向上を図る．この処理をスワップイン（Swap In）と呼ぶ．

このように，スワップイン，スワップアウトの操作は多重プログラミングの多重度制御に使用される．そして OS の設計にも依存するが，1 ページ単位に行うページングとプロセスの全領域を対象にするスワッピングとを区別してファイルを構成する場合がある．

図 5･16 にはページング，スワッピング，及びシステム共通領域の各々に対してページングファイルを作成しているシステムの例を示した．大規模なサーバでは，これらのページングに関係したファイルを別々の装置上に配置して性能の確保を図っている．

5-4-5 仮想記憶の欠点

オンデマンドページングにより必要最低限のページ割当てが可能になり，主記憶の有効利用が可能となった．更に，実記憶（Real Storage）容量以上の仮想記憶を提供できることができるため，多重プログラミングの多重度向上が可能になった．したがって，メモリ不足の理由で CPU の利用率を 100 %稼働できないという状況は回避できるようになった．それらはオンデマンドページング方式による仮想記憶方式の利点である．

しかし，次のような欠点が仮想記憶方式にはある．

(a) ページフォールトの発生は OS オーバヘッドである．

(b) 大容量の仮想記憶を提供する（*V/R* の比率を大とする）とページフォールトの発生頻度が一般的に高くなり，性能が極端に低くなる場合がある．

上記(a)では，ページフォールト割込み処理が CPU を消費し，またページインの操作により入出力処理を 1 ページ単位に行う必要がある．これらをまとめて OS オーバヘッド（Overhead）と称している．(b)は，*V/R* 比が大きいということは，一般的にページフォールトの発生頻度が高くなるということである．仮に，極度に発生頻度が高くなると(a)の OS オーバヘッドが無視できない状態に陥り，ページフォールト処理にほとんどの CPU 時間ならびに入出力資源が消費されてしまうことになり，プロセスに資源の配分がなされなくなる．このような状況をスラッシイング（Thrashing）状態と呼び，仮想記憶方式ではこの状態を回避する必要がある．

■7 群 － 3 編 － 5 章

5-5 演習問題

（執筆者：吉澤康文）［2013 年 2 月 受領］

(1) メモリ管理の目的を述べよ．
(2) 静的分割方式の利点と欠点を述べよ．
(3) 動的分割方式の利点と欠点を述べよ．
(4) フラグメンテーションはなぜ発生するのか述べよ．
(5) フラグメンテーションとは何か説明せよ．
(6) ページ化されたメモリにより解決されたメモリ管理の問題とは何か．また，欠点を二つあげよ．
(7) オンデマンドページングの基本的な考え方を述べ，もたらす効果と欠点を述べよ．
(8) 32 ビットのアドレス空間をもつコンピュータで，ページサイズが 4,096 バイトのとき，最大のページテーブル長はいくらか．このコンピュータでは，ページテーブルのエントリサイズは 4 バイトとする．
(9) ページテーブルは連続的な実記憶になければならない．そこで，上記のようなページテーブルの構成を改良するにはどのようにすればよいか．
(10) 上記の 2 問よりページサイズが大きい場合と小さな場合の利点，欠点を考察せよ．
(11) ページサイズが 4,096 バイトで論理アドレスが 35,594 番地のとき，論理ページ番号はいくつか．また，ページ内オフセット値はいくつか．
(12) 仮想記憶を実現しているコンピュータでは，ページテーブルなどに参照ビット，変更ビットなどが用意されている場合が多い．それらの機構と使用目的は何か説明せよ．
(13) ページフォールトはどのようなときに発生するのか述べよ．
(14) スラッシイングとは何か説明せよ．
(15) 仮想記憶を実現している状況でスワッピングを行う意味は何か述べよ．
(16) メモリ共有の利点は何か．そこでの問題は何かを説明せよ．
(17) プリページングを行うのに適したプログラムにはどのようなものがあるか．