■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）

8 章 仮想記憶制御方式

（執筆者：吉澤康文）［2013 年 2 月 受領］

■概要■

5 章ではメモリ管理の基礎を説明し，その結論として主記憶の徹底した効率的利用のためにページ化されたメモリが発案されたこと，またプログラマからメモリ容量の制約を開放する仮想記憶の発明に至るまでを示した．この章では仮想記憶方式の利点の裏に隠れた技術的な課題を取り上げ，いかにしてそれらの課題を解決してきたかについて説明する．これらを知ることにより情報システム構築の際に問題となる課題解決の助けになると考える．

【本章の構成】

仮想記憶方式における課題を問題提起する．最初に，仮想記憶方式の基本的アイデアであるオンデマンドページングの利点と欠点の評価法を説明するが，これらがプログラムの動作に伴うメモリ参照行動に依存していることを説明する（8-1 節）．OS はプログラムのメモリ参照動作を予測してページの割付け法であるページリプレースメント方式を各種実装しているが，代表的なアルゴリズムとその具体的な実現法を説明する（8-2 節）．仮想記憶の構成法はコンピュータシステムの規模と用途により選択されるが，その種別を説明する（8-3 節）．最後に，これまでの節で紹介していなかった仮想記憶を用いた情報システム構築の際に有益な知識をシステムプログラムとメモリ管理として解説する（8-4 節）．

■7 群 － 3 編 － 8 章

8-1 基本的な考え方

（執筆者：吉澤康文）［2013 年 2 月 受領］

第 5 章ではメモリ管理の基本的な技術を説明し，仮想記憶の発明に至る過程を述べた．更に，仮想記憶方式の利点と欠点なども示した．ここでは，それらの知識を基にして，仮想記憶方式の性能に及ぼす影響について説明する．

8-1-1 プリページング

オンデマンドページング（On Demand Paging）方式ではページフォールトを起こした際に実ページを割り付けるのが基本であり，実ページ割付けごとに OS のオーバヘッドが生じる．そこで，このオーバヘッドを削減するために，主メモリ領域に余裕がある場合には，あらかじめ実ページを割り付ける方式も考えられる．このような方式をプリページング（Pre-paging）と呼んでいる．プリページングの利点はプロセスの要求する全領域を一度に主記憶にロードするので，ページ当たりのコスト（CPU 処理時間と入出力時間の合計）が少なくなると期待できるところにある．

一方，オンデマンドページング方式では，プログラムが参照したページのみをロードすればよいので，各ページ当たりのローディングコストがたとえ高くなっても，トータルではコストを安くできることが期待できる．

つまり，プリページングが有利か，オンデマンドページングが有利かはプログラムのメモリ参照動作（Program Behavior）に依存することになる．OS によっては，この両者をプログラマが選択できるようになっている．したがって，そのような OS の場合では，システム設計者は，両者のページングに関するコストをあらかじめ計算し，性能的に有利になるように工夫ができる．**図 8･1** にはその比較を行うための式である．

P(N) = αN＋c ：プリページングのコスト
D(n) = β n ：オンデマンドページングのコスト

N: プロセスの要求するページ数
n: 参照するページ数（≦N）
α: 1ページロードするコスト（≦ β）
β: 1 回のページフォールト処理のコスト

図 8・1 ページングのコスト比較

図 8･1 では，プリページングによるコストを $P(N)$ とし，オンデマンドページングのコストを $D(n)$ とする．ここで，N はプロセスの要求するページ数である．また，n はプログラムが実際に参照するページ数とする．したがって，両者の関係は一般的に，$n \leqq N$ である．また，各々の方式で 1 ページを主記憶に読込むコストをプリページングでは α とし，オンデマンドページングでは β とすると，$P(N)$，$D(n)$ は各々，$P(N) = \alpha N + c$，$D(n) = \beta n$ となる．ここで，$P(N)$ における c はプリページングする際に固定的に必要な処理コストである．α と β の関係は，$\alpha \leqq \beta$ となることが多い．理由は，プリページングでは複数のページをいっぺんに主記憶にローディングするので固定的なコスト c を使っても 1 ページ当たりに費やすコストは小さくできる．

$P(N)$ と $D(n)$ の比較を行った例を**図 8･2** に示す．ここで，オンデマンドページングが有利な条件とは，そのプログラムが参照するページ数が図 8･2 に示した範囲にある場合である．この図からも明らかなように，オンデマンドページングが有利な n の値の範囲は，c の値と α，β の値に依存することは明らかである．仮想記憶方式では，ページング入出力のコストが性能を左右するので，一般のファイル入出力とは区別して高速化パスを開発し，β 値を小さくする努力がなされている．

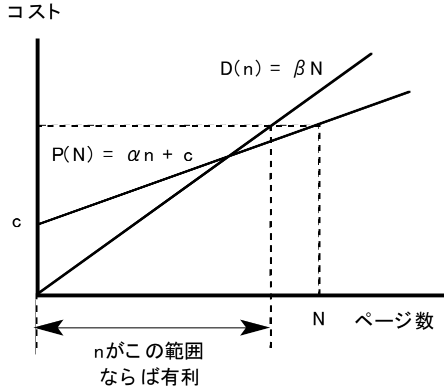

図 8・2　オンデマンドページングがプリページング方式より有利な範囲

8-1-2 プログラムのメモリ参照動作と性能

プリページング方式とオンデマンドページング方式はどちらが有利かは図 8･1 ならびに図 8･2 の評価で決まる．このように仮想記憶方式では，プログラムのページ参照動作が性能を左右することになる．8-1-1 項では，一つのプログラムにおけるプリページングとオンデマンドページングの評価であり，そこでは，参照されるページ数が主たる要因なので比較的静的な評価で済む．つまり，全プログラム容量の N とプログラム実行で参照するページ数 n だけが問題であった．しかし，多重プログラミング環境で複数のプロセスが同時に実行している状況では，そのような静的な評価ではなく，プログラムが参照するページの動的な評価を必要とする．

多重プログラミング環境で仮想記憶を利用している状況では，プロセスの要求する全メモリ量（V）が主記憶容量（R）を越える可能性がある．このような状況でプリページングを行ったとしても，他のプロセスでページフォールトが発生するとページインのために空きのページを作らなければならないので，その際にプリページングで割り付けられたページがその対象になる可能性がある．基本的にオンデマンドページングを採用していると，あるプログラムがプリページングをしたとしてもプログラム終了まで初期に割り付けられたページが確保される保証はない．したがって，ページフォールトの発生を少なくするだけでなく，発生したときは，その処理を高速化することが求められる．

そこで，ページフォールトが発生したときに，他のプロセスから空きのページを作るという方法は，ページアウトの操作（5-4-4 項）があるため高速な処理とはならない．つまり，ペ

ージフォールトのたびにページアウトとページインの 2 回の入出力を繰り返すことになるので，プロセスの進行を妨げてしまう．そこで，仮想記憶方式では，ページフォールトの高速化のために，適度な，空きのページを用意し，ページフォールト発生時に即座に実ページの割当てが可能となるようにしておく必要がある．

このために，仮想記憶方式を実現するには，各プロセスに割り付けた実ページの活用度を評価し，ページアウトを適宜実施することによって空きのページを常時作っておく必要がある．ページアウトを行うことで，限られた主記憶を有効に使うことが可能となる．この実ページの活用度の評価をページリプレースメントアルゴリズム（Page Replacement Algorithm）と呼ぶ．

プログラムのページ参照動作にはいくつかのパターンが予想される．**図 8･3** にその典型的なパターンを示す．主記憶を有効に利用するメモリ参照動作は，小さな領域を密度高く参照するプログラムである．逆に，主記憶を無駄に使用するパターンは，広い領域を参照するが，密度は低く参照する場合である．各ページ内の 1 バイトだけを参照するようなプログラムがその極端な例である．また，一度は参照するが同じ領域を 2 度と参照しないようなパターンも有効なメモリ利用ではない．この例は，領域を逐次参照する例であり，このようなメモリ参照をするプログラムは意外に頻繁にみることができる．そのほかのパターンとして，一度参照するが同一部分の再参照するまで，ある程度の時間を必要とする場合や，領域をアトランダムに参照するようなパターンもある．

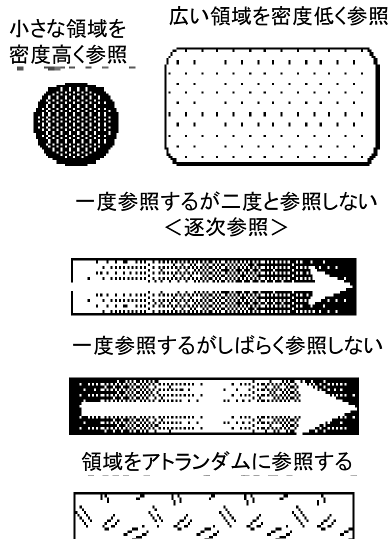

図 8・3　プログラムのメモリ参照動作の例

図 8･3 にはいくつかのメモリ参照パターンの例を示したが，プログラムがこのどれかにいつも当てはまるということは少なく，むしろ時間とともにこれらのメモリ参照パターンを移動すると考えられる．そこでメモリ管理は，時々刻々と変化するプログラムのメモリ参照動

作を監視し，無駄に使用しているメモリ部分をページ単位で検出し，それらをページアウトする必要がある．

5 章 5-1-3 項では，コンピュータ内で参照されないメモリ領域を節約する三つの目標を掲げた．これらの目標はオンデマンドページング方式の発明によりすべて解決されたのであるが，ここで述べているページリプレースメントアルゴリズムは，当初目標としたメモリ節約を更に一歩進める方法である．つまり，一度参照されたことのあるページ化されたメモリ領域の中の参照アクティビティをページ単位で常時監視し，もし参照アクティビティが低いページと判断されたならプロセスから該当するページを取り上げてしまうのである．このようなメモリの節約を行うことによって，空きのページを作り，更にマルチプログラミングの多重度を向上しようとするのである．

8-1-3 ワーキングセットと局所参照性

ページの利用度を測定するには尺度が必要である．P. J. Denning は 1968 年にワーキングセット（Working Set）の考えを発表した．ワーキングセットとは，その名のごとく活躍しているメモリの集合という意味である．つまり，プロセスのページ参照列（Page Reference String）（命令ならびにそのオペランドなど）の履歴を観察したとき，現在の時刻を t とすると，過去の τ 時間の間 $[t-\tau, t]$ に参照する相異なるページの集合を意味する．ここで，τ はウインドウサイズ（Window Size）と呼ばれ，ワーキングセットを定めるパラメータである．

ワーキングセットは各プロセス単位のページ参照アクティビティとして測定すべきものであるので，プロセス j のワーキングセットを Wj (t, τ) と表記することにする．その具体的なイメージを**図 8･4** に示す．この例では，ワーキングセットは Wj (τ, t) = {a, b, c} であり，3 ページからなる．

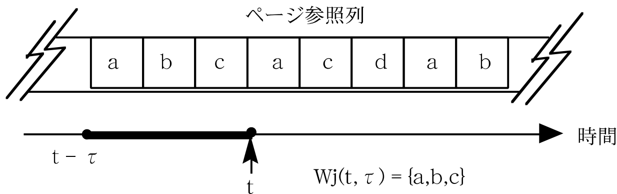

図 8・4　プロセス j のワーキングセット

図 8･4 から明らかなように，ワーキングセットは各プロセスが現時点で参照しているホットスポット（Hot Spot）のページの集合を測定していることになる．このような測定を行うのは，メモリ参照動作の予測を行う可能性がある．つまり，図 8･4 の例でいうならば，次の時点 $(t+1)$ において参照するページが Wj (τ, t) 内に存在する確率が高いと判断できることである．

プログラムのメモリ参照については多くの研究がなされてきたが，このなかで，以下の二つのことが確認されている．

(a) ワーキングセット内のページは近い将来参照される確率が高い．

(b) プログラムの実行過程で参照するページの範囲は特定の部分に集中する．

上記(b)はプログラムの局所参照性（Program Locality）と呼ばれるもので，概念的には**図 8･5** に示すような特性を意味している．つまり，プログラムを実行させるとメモリ領域のある特定の部分に参照が集中するという性質を示している．上記の二つの特性は多くのプログラムに当てはまるが，もちろん例外的なプログラムも存在する．それらは，一般的に局所参照性のないプログラムと呼ばれ，仮想記憶方式では取扱いにくく，性能を保障することができない．

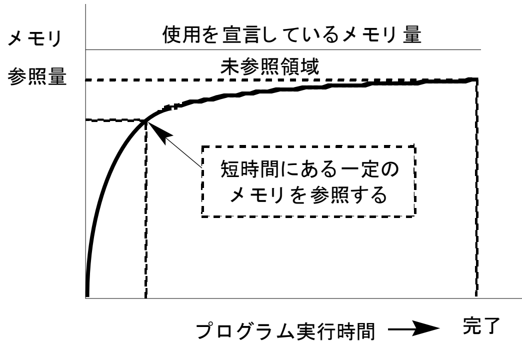

図 8・5　プログラムの局所参照性

ワーキングセットは常に定まっているわけではなく，一般的にプログラムの実行過程で変化する．概念的な表現をすると**図 8･6** のようになる．プロセスは複数のプログラムを次々と実行していくのが一般的である．例えば，コンパイラなどは，ソースコードを読み込み，シンタックスチェックを行い，中間語を生成し，そしてオブジェクトコードの生成やその最適化コード生成を行うなど，機能の異なる処理をする．したがって，ワーキングセットはプログラムのフェーズが変わることにより大きく変化することもある．

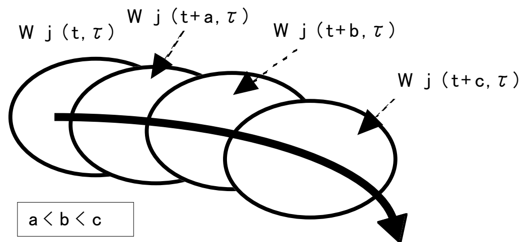

図 8・6　ワーキングセットの移動

■7 群 - 3 編 - 8 章

8-2 ページリプレースメント

(執筆者：吉澤康文)［2013 年 2 月 受領］

8-2-1 代表的なアルゴリズム

ページリプレースメントの目的は，近い将来参照される確率が最も低い実ページを求め，空きページとすることにある．逆のことをいうならば，空きページとしたページが近い将来再参照されてページフォールトを発生してしまったとき，ページリプレースメントは失敗と判断される．

このように，ページリプレースメントは将来のページ参照を予測することになるため，絶対的に成功するアルゴリズムは存在しない．したがって，各 OS では，プログラムのメモリ参照モデルを仮定してリプレースメントアルゴリズムを実現している．ここでは，上記のワーキングセットの考え方とプログラムの局所参照性が存在することを前提にして実現されている，代表的な 3 種のページリプレースメントアルゴリズムを説明する．

(1) LRU アルゴリズム

メモリ内で最も昔に参照された実ページからページアウトの候補とするアルゴリズムである．この方式では，参照されて最も時間の経っているページは近い将来参照される確度が低いと判断する．本方式を LRU（Least Recently Used）方式と呼び，この名のごとく，最も最近でなく使用されたページを選ぶという意味である．

メモリ管理は**図 8･7** に示すような管理テーブルによって LRU を実現する．ここでは，各実ページを管理するテーブルを作成し，そこに該当ページの最終参照時刻を入れておく．このような値をもとにして実ページ管理テーブルをソート（Sort）し，昇順に並べることにより，先頭のエントリが最も昔に参照されたページとなる．したがって，LRU アルゴリズムではページアウトすべきページ数をこのリストの先頭から取り出し，空きページとする．このように，プロセスに割り付けられた実ページをページアウトの候補として取り上げることをページスチール（Page Steal）と呼んでいる．

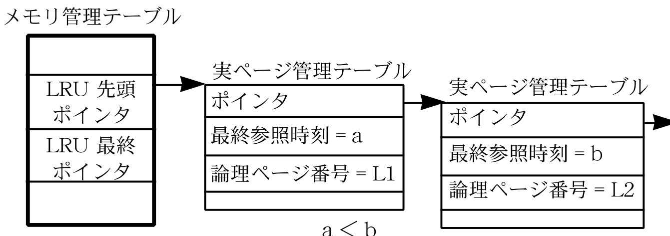

図 8・7　LRU におけるリプレースメント方式

現実のコンピュータでは，実ページを参照した時刻を記録する機構を備えているものは極めて少ない．その代わり，仮想記憶方式を実現しているコンピュータでは，実ページを参照（読込みや書込み）した際に，該当ページを読み込んだこと及び書き込んだことを示すフラ

グ（あるいはビット）がオンになる機構を備えているものが多い [*1]．メモリ管理は参照フラグ（あるいは参照ビット：Reference Bit）がオンかオフかを定期的に検査することにより，最後に参照した時刻に相当する値を計算する．この計算値に基づき，図 8･7 にあるような実ページ管理テーブルを管理し，LRUアルゴリズムを疑似的に実現している．

(2) ワーキングセットアルゴリズム

ワーキングセットの概念を基本とするアルゴリズムである．先に 8-1-3 項に示したように，プロセス j のワーキングセット Wj (t, τ) はウインドウサイズτが与えられると各プロセスの実行時間 t により定まる論理ページ集合である．そこで，ワーキングセットに含まれる論理的なページ集合に対して実ページの割付けを保障する方式がワーキングセットアルゴリズムである．このモデルの考え方は，プログラム実行にあたり，ワーキングセットに実ページ割付けを保障することでページフォールトの発生を最小化できるという仮定に基づいている．

上記の仮定から，本アルゴリズムでは，プロセスに割り付けられている実ページの中で，ワーキングセットの外に割り付けられているページをページアウトの対象とする．ワーキングセットアルゴリズムの概念的な説明を**図 8･8** に示す．ワーキングセットアルゴリズムでは，ページスチールの要求に対して，プロセスのワーキングセットを主記憶内に維持できない場合がある．つまり，スチールの対象となる実ページが全くなくなるときである．このような現象をオーバイニシエーション（Over Initiation）状態と呼び，この場合は，実メモリ不足の状態であり，マルチプログラミングの多重度が高すぎて性能低下をきたしている状況と判断される．したがって，プロセスのスワップアウトと試みるのが一般的である．

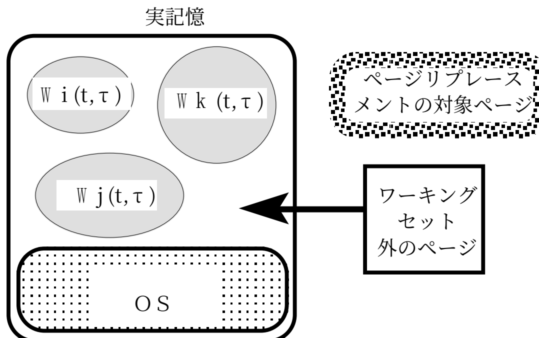

図 8・8　ワーキングセット法によるリプレースメント方式

ワーキングセットアルゴリズムを実現するにはプロセスに割り付けた実ページの最終参照時刻に相当する情報が LRU と同じように必要になる．LRU とワーキングセットアルゴリズムとの基本的な違いは，この最終参照時刻の定義の違いにある．つまり，LRU での時刻はこ

---

*1 参照ビット（Reference Bit）は該当ページが CPU によって参照された記録となる．同様に，実ページに書込みを行うと変更ビット（Change Bit）がオンになるマシンもある．これらの情報はページテーブルエントリに存在することが多い．OS はこれらの情報を基にしてリプレースメントやページアウトなどの処理を行う．

の世の時間，つまりリアルタイム（Real Time）である．したがって，LRU でページスチールの対象となるページは実時間上で最も昔に参照されたページということなる．

一方，ワーキングセットの時間は，プロセスに与えられた時間軸上での時間である．つまり，プロセスがディスパッチされて CPU を与えられた時間上での最終参照時刻ということであるので，プロセス固有の時間軸が基本である．ワーキングセットアルゴリズムはこのように，プロセス固有のページ参照動作を基本にしているため，ローカルポリシィ（Local Policy）のアルゴリズムと呼び，LRU のようにプロセス個々のページ参照特性は考慮しないアルゴリズムをグローバルポリシィ（Global Policy）と区別している．

(3) FINUFO アルゴリズム

LRU アルゴリズムやワーキングセットアルゴリズムの実現は最終参照時刻に相当する情報収集が煩雑な処理となることが多い．そこで，実ページのアクティビティを単純に測定する方法として考案されたのが，FINUFO（First In Not Used First Out）アルゴリズムである．

本アルゴリズムを実現するための前提条件は，実ページに対して参照が行われたときに各実ページに対応した参照フラグ（Reference Flag）が利用できるということである．そのフラグを R とする．参照が行われたときにオンとなり，R＝1 となる．このフラグは特権命令によってオフにすることができるとする．

図 8･9 で本アルゴリズムリズムを説明する．図にある a，b，c，… は実ページ管理テーブルを示す．各管理テーブルは固定的なリスト構造になっているとする．そして，実ページのアクティビティを検査するためのポインタがあり，このポインタはページスチールの要求が発生した時点で検査を開始する実ページ管理テーブルを示している．

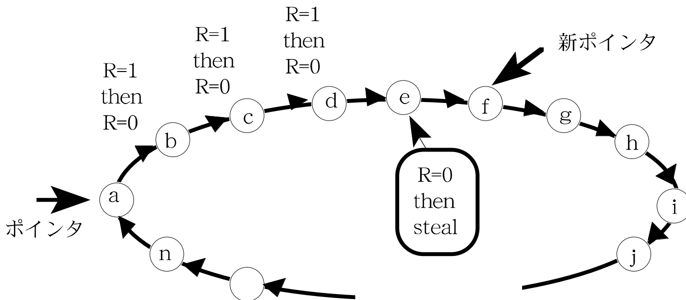

図 8・9　FINUFO によるリプレースメント方式

ページスチールの要求がくると，検査開始のポインタを基にして，該当の参照フラグがチェックされる．この例では，ページ a の参照フラグ R はオン（R＝1）であり，参照があったことを示している．したがって，今回はページスチールの対象外と判断される．そしてページ a の参照フラグはオフにされる（R＝0）．つまり，今回の検査では参照があったのでアクティビティが高かったとみなされ，ページスチールされなかったが，ポインタがもう一周してくる間にページ参照がなければ次回の検査時はページスチールされることを意味する．

それから，次の実ページ管理テーブルをたどり，ページ b の検査を行う．同様の操作を行

い続けていくと，参照フラグ R＝0 なる状態にある実ページ e を見出す．つまり，このページはポインタが一周する間に一度も CPU から参照がなかったことを意味しており，アクティビティが低いとみなされる．したがって，実ページ e はページアウトの候補となる．仮に，空きページの要求が 1 ページであったときは，検査ポインタは次のページ f を示して操作を完了する．

FINUFO アルゴリズムはグローバルポリシィであるが実現が容易であること，正確な LRU とは異なるが LRU のような要素を含んでいることなどからそれを採用しているメモリ管理がある．

8-2-2 仮想記憶を支えるハードウェア

仮想記憶方式を実現するには以下の機構をもつことが望ましい．

・アドレス変換機構（DAT）

・TLB（Table Look aside Buffer）による高速アドレス変換

・ページ参照記録機能（読出し，書込みを区別したものが望ましい）

DAT はページ化された記憶を用いる際の最低限のハードウェア機構であるが，その処理過程においてアドレス変換テーブルを参照するため遅延が生じる．そこで，主記憶参照時間を短縮する方法として，TLB が考案されている．

TLB は，**図 8･10** に示すように，アドレス変換を行ったことのある論理ページアドレスと物理ページアドレスの対応を格納した高速な CPU 内のメモリである．DAT はアドレス変換時に TLB を参照することで高速に物理ページアドレスを出力することが可能となる．TLB は高速な CPU の中に存在する記憶であるため，一般的に大量のエントリを用意することはできない．また，入力の論理ページアドレスに対して TLB エントリの対応付けをどのように行うかなどの技術的な課題もあり，プロセッサ設計の一つの重要なポイントである．

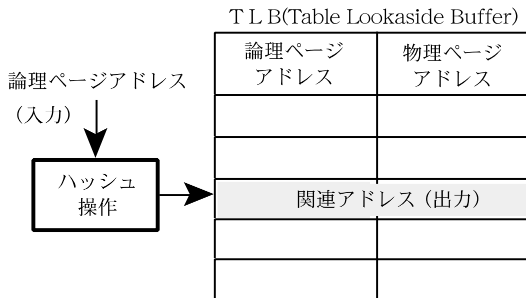

図 8・10　TLB による DAT の高速化

仮想記憶方式実現のためには更に必要なハードウェア機構がある．先に述べた LRU，ワーキングセット，FINUFO などのページリプレースメントアルゴリズムの実現では，いずれも各実ページの参照アクティビティを測る参照フラグが必要となる．参照フラグの具体的な用途と働きは FINUFO のところで説明した．ページ参照フラグは命令フェッチやメモリ参照時にオンとなるが，各ページへの参照における書込みと読出しを区別するために，メモリ書込

みフラグ（Change Bit）を備えているコンピュータが多い．書込みの場合は参照フラグと書込みフラグが共にオンとなることになる．書込みフラグは，以下に説明するページスチール時に利用される．

ページリプレースメントアルゴリズムでは，スチールするページを選ぶが，その後の具体的な操作は以下の二つに別れる．

(a) 直ちにスチールしたページを空きページとする場合

(b) スチールしたページをページングファイルにページアウトする場合

上記(a)のケースは，スチールしたページが命令実行によって内容を変更されていない場合である．つまり，該当ページのメモリ書込みフラグ（Change Bit）がオフになっており，スチール対象のページと同一内容のものがページングファイルに存在する場合である．この場合のスチールの処理は，同一内容のものがページングファイルに存在するので，直ちに他のプロセスに割り当ててもよい．しかし，(b)の場合は書込みフラグがオンとなっており，ページングファイル内の該当ページとは内容が異なっているので，最新の情報を保存するためにページアウトを必要とする．

8-2-3 実現方式の概要

(1) 実記憶ページ管理テーブル

ページリプレースメントアルゴリズムの考え方を示してきた．ここでは，それを実現する一つの方法について説明する．どのアルゴリズムを実現するのもページ参照のアクティビティを測定することが基本である．参照アクティビティの測定は，参照ビットのオン・オフ状態を調べることである．これらの情報がページテーブルに格納されているときはページテーブル内を調べるが，そうでなく，特定の命令を実行することでチェックするマシンもある．

メモリ管理では，実メモリの 1 ページ単位に，**図 8･11** に示すようなテーブルを作成しておく．先頭には本テーブルのリストを作るためのポインタとする．このポインタはプロセス管理テーブルからチェインしており，各プロセスに割り当てられている実ページを管理するのに利用できる．第 2 のエントリはプロセス識別子であり，この実ページが割り当てられているときのプロセスを区別する番号を入れる．UNIX などでは PID である．そして，そのプロセスの仮想記憶アドレスを格納する．

エントリ	説明
テーブル・チェイン	テーブルの並び替えに必要
プロセス識別子	ページ・テーブル操作や リクレイム処理などに使用
仮想記憶アドレス	
未参照カウンタ(URC)	URCが大きいほど未参照間隔が長い
各種管理用フラグ	ページング中，使用目的，共有， などの状況表示

図 8・11　実記憶管理テーブルエントリの構成例

本テーブルで最も大切なのは，未参照カウンタ値である．この値は，本実ページのアクティビティを示すもので，LRU 法やワーキングセットアルゴリズムにおけるページアウト候補の判断基準となる．ここでは，この値を URC（Un-referenced Counter）と省略することにす

る．つまり，URC の値が大きいほど未参照間隔が長いことを意味し，アクティビティが低いと解釈する．最後のエントリは実ページの状態であり，ページング入出力中，何の目的で本実ページを利用しているかなどの情報表示である．

ワーキングセットアルゴリズムを実現する方法ではこれらの情報を含んだテーブルは，プロセス管理テーブルからチェインしている．**図 8･12** にはその様子を示したものである．このように管理することで，プロセスに割り当てた実ページ数やページの実体などが何であるか分かる．そして，もし URC の値で実記憶管理テーブルが並べられているならば，ページアウトの候補を迅速に選ぶことができる．

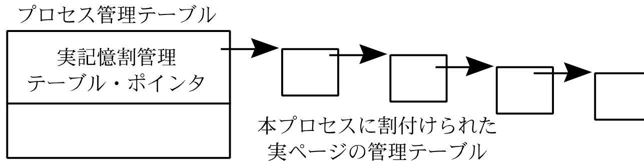

図 8・12　ワーキングセット法における実ページ管理テーブル構成の例

LRU や FINUFO などではプロセス管理テーブルから割り当てたページをリスト構造で管理する必要は特にない．

(2)　未参照カウンタの更新

上記のように実ページ管理テーブルを構成しておけば各ページのアクティビティの測定は単純である．基本的な方法は，ある時間間隔で実ページの参照ビットがオンか否かを調べるだけでよい．つまり，以下の方法をとる．

(a) 実ページの参照ビットがオンならば，URC の値をゼロとする．そして，参照ビットをオフにする．その後，実記憶管理テーブルのチェイン操作を行い，本エントリを図 8･12 に示すリストの先頭におく．

(b) もし参照ビットがオフであるならば，URC を 1 増加する．

上記の方法で各々の実ページに対する URC 値を更新することにより，プロセスの管理テーブルからチェインしている実記憶管理テーブルエントリは URC 値が昇順に並べられたリストになっている．つまり，最後のエントリが最も参照アクティビティが低いことになる．そこで，以下，LRU 法とワーキングセットアルゴリズムにおけるアクティビティの測定方法を説明する．

(3)　LRU の測定方法

LRU では**図 8･13** に示すように，仮想記憶に利用する全実ページの管理テーブルをリスト構造にしておく．一定時間間隔で未参照カウンタの更新を行い，その値で管理テーブルをソートしておくならば,最後のエントリには最大の URC 値をもったページがあるはずである．つまり，そのコンピュータシステムにおいて最もメモリ参照アクティビティの低い実ページと判定されることになる．したがって，ページリプレースメントの際には本リストの最後から必要なだけ実ページを選んでページアウトすればよいことになる．

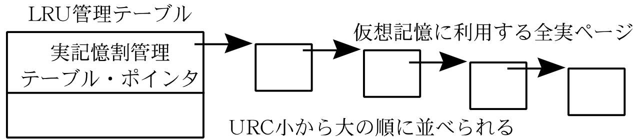

図 8・13　LRU 法における実ページ管理テーブル構成の例

LRU はこのようにシステム全体での実ページのアクティビティを測定しているので，グローバルなリプレースメントアルゴリズムと呼ばれる．測定間隔は OS 設計によって決められることになる．

(4)　ワーキングセットアルゴリズムでの測定

ワーキングセット法では図 8･12 に示したように，各プロセス単位に実記憶管理テーブルをチェイニングする．つまり，各プロセス単位の管理が基本である．URC の測定もプロセス単位に行われる．つまり，時間のインターバルがプロセスの時間の単位となるのである．つまり，プロセスがディスパッチされて CPU を使用することになるが，その時間を単位とした経過時間により実ページのアクティビティが測定されるのである．

ワーキングセットは 8-1-3 項で説明したウインドウサイズ（τ）内に参照されたページの集合である．このため，プロセスが CPU をτ時間使用するたびに URC の更新をするのが基本である．このインターバルで URC を更新したとき，URC 値が 0 ならばワーキングセット内のページであり，1 以上ならばワーキングセット外と判定される．このように，プロセスの CPU 使用時間軸のことをバーチャルタイム（Virtual Time）と呼ぶことがある．

実際には，τ時間単位で測定するのではなく，例えば，τ/3 単位に測定を行う．このようにすると，URC が 2 以下の実ページはワーキングセット内と判定され，3 以上がワーキングセット外となる．ウインドウサイズの数分の 1 を測定間隔にすることにより，より正確なワーキングセットの判定が行えることになる．

このように，ワーキングセットアルゴリズムでは LRU とは異なりプロセス単位にページのアクティビティを測定するのでローカルストラテジィと呼ばれるわけである．LRU とワーキングセットアルゴリズムでのページアクティビティの測定方法の具体的な方法を示してきたが，これらは本来の定義による LRU とワーキングセットではない．定義によると 1 命令単位で参照されたページのアクティビティである．しかし，そのような測定を行うのは現実的でない．OS オーバヘッドの塊になってしまう．そこで，上記のような測定手段を用いるのである．この意味で，現実の LRU やワーキングセット法は疑似 LRU とか疑似ワーキングセット法と呼ばれることがある．

8-2-4 LRU スタック

(1)　ページ参照列の取得方法

現実のプログラムがどのようにページを参照したかを調べる手段は命令トレーサ（Instruction Tracer）を使用して情報収集する以外にない．命令トレーサはプログラムが実行するすべての命令の実行列を記録するソフトウェアである．メモリ参照は命令アドレスだけ

でなくオペランドアドレスとその長さなどの情報が必要となる．一般的に，トレーサ情報は長大になるので，格納できる記録媒体の容量の制限を考えて限定して情報を収集しなければならない．

命令トレースの結果が得られるとプログラムが参照した論理アドレスの列が得られる．ページング機構を備えたマシンでは，メモリ参照に関する情報はページアドレスが重要であり，これをページ参照列（Page Reference String）と呼ぶことにする．

例えば，参照したページが全部で6ページ {a, b, c, d, e, f} であり，その出現したページ参照列が，R = {a, b, a, c, d, e, a, f, e, a, c, a, b, e, …} であったとすると，命令トレーサの取得した情報の中で重要なのはこの参照列ということになる．

(2) LRUスタック内の距離

このようなページ参照列が取得できると，現実の世界では実現できない正確なページリプレースメントアルゴリズムを仮想的に実現し評価することができ，ワーキングセットを求めることが可能になる．ここでは，正確なLRUページリプレースメントアルゴリズムを実現する方法を説明するが，そのためにLRUスタックを用いる．

説明のために，第 i 番目のページ参照を r(i) と書く．上記の例のようなページ参照列 R = {a, b, a, c, d, e, a, f, e, a, c, a, b, e, …} のときは，r(1) = a，r(2) = b，r(3) = c，… である．また，ページが参照されたときに，参照されたページ r(i) と同一のページが LRU スタック内に存在する位置をスタック内の距離（Stack Distance）と呼び，d(i) と書くことにする．

図8･14にはLRUスタックの具体的な例を示した．この図の左側が現在の状況であり，数字はLRUスタックの深さを示している．この図では，LRUスタックの先頭ページが a であり，第2番目の深さにページ b がある．このとき，ページ c が参照されたとすると，ページ c を LRU スタック内に求める．ページ c は深さ3に存在するので，スタック内の距離は d(c) = 3 である．

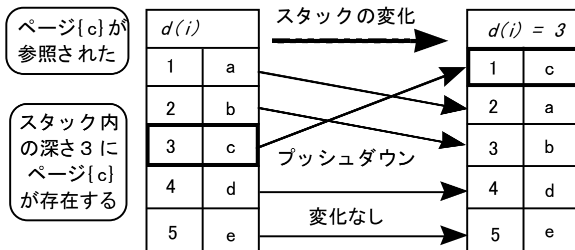

図8・14　スタック内に参照ページが存在したときの操作

LRU スタック操作では，新しいページ参照 r(i) がなされると，はじめに LRU スタック内の距離 d(i) を求める必要がある．その手順は以下のとおりである．

(a) LRU スタック内に該当ページ r(i) が存在するか探す．

(b) もし LRU スタック内に存在するならば，スタック内の深さ d(i) を求める．

(c) LRU スタック内にページ r(i) が見つからなかったときは，スタック内の深さを d(i) = ∞ とする．

具体的には，最大の整数を割り当てる．これは，ページ参照列ではじめて出現したページの場合である．

図 8･15 には LRU スタックの初期状態を左側に示す．最初のページ参照 r(1) = a であったとすると，LRU スタック内は空であるので，同一のページは存在しない．つまり，はじめてページ a が出現したので，LRU スタック内の距離 d(1) = ∞となる．

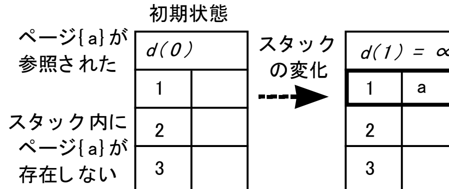

図 8・15　LRU スタックの初期状態と第 1 回目のページ参照

(3)　LRU スタックのプッシュダウン操作

ページ参照がなされると，上記の LRU スタック内の距離 d(*i*) を求める．この値を基にして，LRU スタックのプッシュダウン操作を行うが，図 8･14 に示したように，r(*i*) = c であったとき，その LRU スタック内の距離 d(*i*) = 3 であるならば，ページ c をスタックの先頭に入れ，d(1)，d(2) にあったページ (a, b) をプッシュダウンする．そして，d(i) +1 よりも深い位置にある LRU スタックには何の操作もしない．

この手順を一般的に書くならば以下のとおりである．

(a) d(*i*) ≠ ∞のとき，LRU スタックの深さ d(*i*) のページを取り出し，スタックの深さ d(1) から d(*i*)－1 までをプッシュダウンする．そして，参照されたページ r(*i*) を LRU スタックの先頭に入れる．

(b) d(*i*) = ∞のとき, LRU スタックの先頭に参照されたページ r(*i*) を入れプッシュダウンする．

(4)　LRU スタック操作の例

図 8･16 に上記のページ参照列 R = {a, b, a, c, d, e, a, f, e, a, c, a, b, e, …} と LRU スタック操作を示す．LRU スタックは初期状態では空とする．この図の下の方には LRU スタック内の距離が記されている．LRU では最近参照されたページを主記憶に残し，参照されないページをリプレースメントの対象とするアルゴリズムである．したがって，このスタック内の距離はページのアクティビティを示していることになり, 小さな値であるほど最近 CPU により参照されたことを示しており，アクティビティが高いということになる．

(5)　LRU におけるページフォールトの発生

ページが参照されたとき，ページフォールトが発生するか否かの判定は図 8･16 に示した LRU スタック操作において可能となる．重要なのはページ参照が行われた際に求める LRU スタック内の距離 d(*i*) である. もし, 主記憶として用意されているページ数を *m* としたとき，d(*i*) ≦ *m* であるならば，参照されたページ r(*i*) は主記憶に存在することになるので，ページ

参照順番: i		1	2	3	4	5	6	7	8	9	10	11	12	13	14	
参照列: r(i)		*a*	*b*	*a*	*c*	*d*	*e*	*a*	*f*	*e*	*a*	*c*	*a*	*b*	*e*	
LRU スタックの深さ	1	*a*	*b*	*a*	*c*	*d*	*e*	*a*	*f*	*e*	*a*	*c*	*a*	*b*	*e*	
	2		*a*	*b*	*a*	*c*	*d*	*e*	*a*	*f*	*e*	*a*	*c*	*a*	*b*	
	3				*b*	*a*	*c*	*d*	*e*	*a*	*f*	*e*	*e*	*c*	*a*	
	4					*b*	*a*	*c*	*d*	*d*	*d*	*f*	*f*	*e*	*c*	
	5						*b*	*b*	*c*	*c*	*c*	*d*	*d*	*f*	*f*	
	6								*b*	*b*	*b*	*b*	*b*	*d*	*d*	
	7															
	8															ページフォールト発生数
距離: d(i)		∞	∞	2	∞	∞	∞	4	∞	3	3	5	2	6	4	
ページング事象 主記憶のページ数 m	2	*	*		*	*	*	*	*	*	*	*		*	*	12
	3	*	*		*	*	*	*	*			*		*	*	10
	4	*	*		*	*	*		*			*		*		8
	5	*	*		*	*	*		*					*		7

図 8・16　ページ参照列と LRU スタックの操作

フォールトは発生しない. 逆に, d(i) ＞ m の場合はページフォールトが発生することになる.

図 8･16 には主メモリのページ数と LRU スタック内の距離との関係から，ページフォールトが発生するページ参照の場合は“*”を示した. そして，一番右の欄にページフォールト発生数の合計を示した. 主記憶のページ数 $m = 2$ のときはページフォールト数が 12 回発生することを示し，$m = 3$ のときはフォールト数が 10 回と減少している.

(6)　LRU アルゴリズムのシミュレーション

ページ参照列が入手できたなら，上記のような LRU スタックのシミュレータを作成し，それにより LRU 法の場合のページフォールト発生回数を求めることができる. しかし，現実の世界ではシミュレーションのように厳密な LRU を実現することは不可能であり，疑似的な LRU を行うことになるが，おおよその定量的な評価をすることが可能となる. したがって，プログラムのメモリ参照はプログラムごとに異なるので，仮想記憶の環境で性能を重視しなくてはならないプログラムについては，その特性を知るうえでこのような性能評価は必要になる.

■7 群 - 3 編 - 8 章

8-3 仮想記憶の構成法

(執筆者：吉澤康文)［2013 年 2 月 受領］

8-3-1 単一仮想記憶

仮想記憶の実現方式には 2 とおりある. その一つが単一仮想記憶方式である. この方式は, 該当するコンピュータに許される最大のアドレス範囲を仮想的に実現する. 例えば, 32 ビットのアドレス空間が可能なコンピュータならば, 4 ギガバイト (GB) のアドレスまでを提供する方法である.

単一仮想記憶方式では, 拡大化されたアドレス空間にパーティションを作りマルチプログラミングの多重度向上を実現することができる. この場合にもフラグメンテーションの問題はあるが, 仮想記憶を用いているので空き領域が発生しても実ページが割り付けられないために実害を小さくする工夫が可能である.

仮想記憶が商用コンピュータとして本格的に利用された 1970 年代のはじめには大半の汎用コンピュータにこの方式が採用された. 具体的には, IBM 社の OS/VS1 である. 当時, IBM 社の System/370 シリーズは 24 ビットマシンであり, 最大のメモリ領域は 16 メガバイト(MB) であった.

8-3-2 多重仮想記憶

プログラマにとって記憶容量の制限は煩わしい問題である. できれば無制限にメモリは利用したいと考える. 仮想記憶方式では, 論理的なアドレス空間として該当のコンピュータにおける最大容量を仮に各プロセスに与えても, 実際にメモリ参照が行われないなら資源を用意する必要はない. そこで, 各プロセスに対してアーキテクチャとして許される最大のメモリ領域をプログラマに提供する方法が考えられた. それが多重仮想記憶方式である.

多重仮想記憶方式では, 各プロセスに対して独立したプロセス固有領域を与えることができる. そのサイズはアーキテクチャの許される最大容量である. 例えば, 32 ビットアドレッシング空間をもつコンピュータでは 4 ギガバイトを各プロセスに与えることができるようにするということである. **図 8･17** にはその概念図を示す.

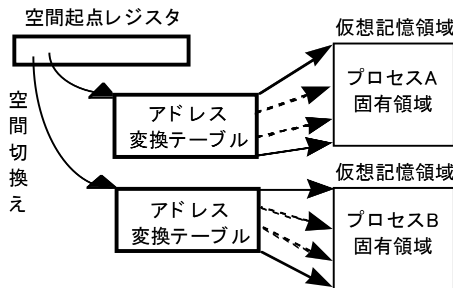

図 8・17 各プロセスが独立したアドレス空間を保有する多重仮想記憶方式

仮想記憶方式では, 5 章 5-3-3 項で説明した動的アドレス変換機構 (DAT) による論理アド

レスから物理アドレスへの変換を行うが，そのときのアドレス変換テーブルの所在アドレスを示すのは特定の制御レジスタである．つまり，アドレス空間の起点はこの特定の制御レジスタということになる．図 8･17 に示したこのレジスタを空間起点レジスタ（Address Table Origin Address Register）という．この制御レジスタはアドレス変換テーブルの先頭アドレスを示しているので，DAT はアドレス変換テーブルのアドレスをこの制御レジスタの内容によって知り，5 章図 5･10 のような変換が行えるのである．

OS は新しいプロセスが生成するときに，メモリ管理によってアドレス変換テーブルを新規に作成する．そして，空間起点レジスタに入れるべき内容をプロセス制御テーブル（PCT）内に格納しておく．このような操作を事前に行うことで，プロセススケジューラはプロセスをディスパッチするときに空間起点レジスタを設定し，そのプロセスの再開命令アドレスから命令フェッチを行わせればよい．

図 8･17 から明らかなように，各プロセスの仮想記憶領域は完全に独立している．このことから，多重仮想記憶方式では，各プロセス領域はハードウェア的に完全に保護されていることになり大きな利点となっている．また，各プロセスに対して最大限のメモリ領域を与えることが可能であることも特徴点である．更に，原理的には無限のアドレス空間を生成することが可能となる．制限は，CPU の能力，スラッシイングを起こさないだけの実記憶容量の実装，そして仮想記憶をサポートするだけのバッキングストア容量である．

以上から，多重仮想記憶を実現し，それを運用管理していくには性能評価を慎重に行う必要が生まれる．近代の OS，例えば，大形メインフレームコンピュータにおいて 1970 年代の後半から企業の情報中枢として利用されている IBM 社の MVS は多重仮想記憶を採用している．また，UNIX 系カーネルも構造的には多重仮想記憶方式を採用しているものが多く，パソコン系の OS にもその傾向がある．

■7 群 - 3 編 - 8 章

8-4 システムプログラムとメモリ管理

（執筆者：吉澤康文）［2013 年 2 月 受領］

8-4-1 インタフェース

1 章に述べたように，オペレーティングシステムには機能マシンの提供と資源管理機能という二つの大きな目的がある．機能マシンとしての代表は，プロセス管理とファイル管理である．それらについてはプログラマに提供する機能がインタフェースとして定まっている．一方，メモリ管理はプロセススケジューリング同様に OS の資源管理の重要な機能であるがプログラマとの機能インタフェースは多くは存在しない．

メモリ管理に関係するインタフェースの多くは，システム運用やシステムプログラムの開発に属する場合である．

8-4-2 動的メモリアロケーション

メモリ管理の機能のなかで，実行時に作業領域を必要とする場合がある．例えば以下のような場合がある．

・プログラム実行前に作業領域の必要性が不明の場合
・実行中に領域のサイズが決定する場合
・実行中に領域を作成した方が性能向上が図れる場合

このような場合のメモリ領域確保を動的メモリアロケーション（Dynamic Memory Allocation）と呼ぶ．UNIX では malloc() というライブラリや brk(), sbrk() というシステムコールがそれにあたる．

8-4-3 仮想記憶制御インタフェース

仮想記憶の環境でプログラムを実行すると，実ページの割付けがすべてオペレーティングシステムに依存してしまい，命令実行の性能が保障されないことになる．つまり，ページフォールトが発生することにより命令実行に遅延が生じることになる．このような状況は性能保障を行わねばならないシステムプログラムやオペレーティングシステムの開発者にとって重要な問題点になる．

そこで，以下のような機能をシステムプログラマに提供している OS もある．

(a) プロセス空間の仮想アドレスと実アドレスを一致させる（V = R）．
(b) プロセス空間のすべてに実ページを割当てページングの対象外とする．
(c) 指定した領域を実ページに固定しページアウトの対象外にする．
(d) 指定した領域を実ページにローディングする．
(e) 指定した領域をページアウトする．
(f) 指定した領域の内容を解放する．

(a)，(b)のような要求は，仮想記憶方式以前のプログラムを実行させるためや入出力装置との制約でページフォールトの遅延が許されない場合などのために用意されることがある．また，十分な実記憶を装備したリアルタイム処理の応用にも利用される．(c)の場合は，ある特定の処理のためにページフォールトの遅延を防止するためである．(d)はページフォールトの

遅延を防止するために行う要求である．(e)の機能は，プログラマの判断で，ある領域を二度と参照しないようなことが分かっている場合に実記憶を積極的に解放するために使用する．**図 8･18** に上記の機能の概念図を示す．

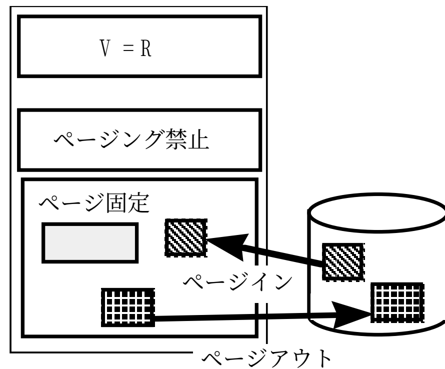

図 8・18　仮想記憶方式におけるメモリ領域制御機能

8-4-4 メモリの共用機構

プロセス間通信では UNIX の共有メモリに関するインタフェースを 7 章 7-2-7 項(3)にて示し，図 7･31 ならびに図 7･33 によって概念的な説明を行った．前節 8-3-2 項に示したように多重仮想記憶の下では各プロセスが独立したメモリ空間を保有しているが，このような環境でメモリを共有するには, **図 8･19** に示すように仮想記憶上にプロセス全体で共有する領域を設ける方法が考えられる．

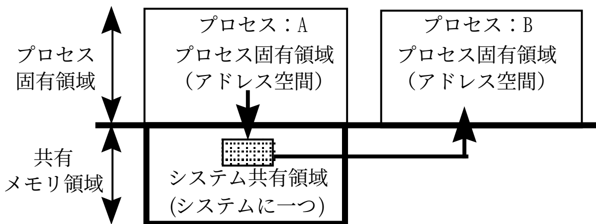

図 8・19　共有メモリによるメモリ配置

図 8･19 に示したように，プロセス A と B が共有領域をもつようにするトリックはアドレス変換テーブル（ページテーブル）の操作により行われる．一つの方法は，各プロセスを生成する際のアドレス変換テーブルのうち，システム共有領域に相当する部分は全く同一にしておくのである．**図 8･20** にその概念図を示した．共有領域用ページテーブル自身も共有させる方法をとれば管理が容易になる．この方法では，共有メモリ用のページテーブルを実際には一つにしておかないと，ページテーブルの更新時に，すべてのプロセスのページテーブルを修正しないとならなくなる欠点がある．したがって，ページテーブルを共有できるような 2 段階のページテーブル構成のハードウェアに向いている．

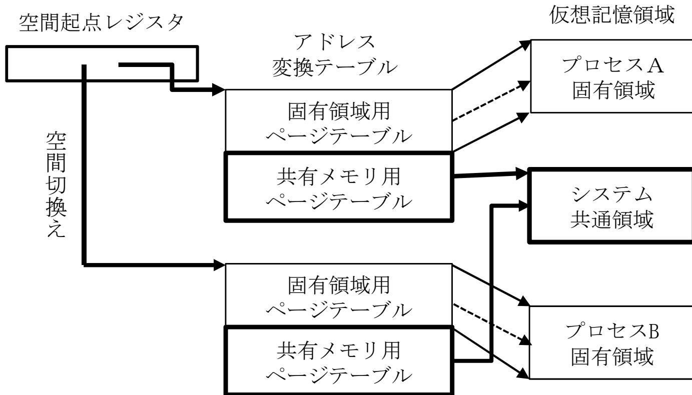

図 8・20　システム共通領域をもつ共有メモリの仕組み

もう一つの方法は，UNIX の共有メモリのような仕様に向いた方法である．つまり，共有する仮想記憶領域の参照はページ単位のウインドウを通して行う方法である．この方法は，各プロセスの空きの仮想記憶をウインドウにするので，一般的に同一の共有メモリ領域に対するアクセス時の仮想記憶アドレスは異なっている．**図 8･21** にその方式を示す．ここでは，ページテーブルが示す実ページアドレスが同じであり，実ページだけを共有しているのが特徴である．したがって，プロセス A の共有メモリ参照ウインドウのアドレスは a であり，一方，プロセス B の同一内容を参照するアドレスは b である．

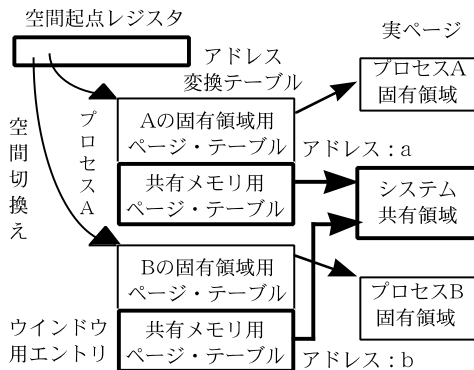

図 8・21　ウインドウを通した共有メモリの仕組み

■7 群 － 3 編 － 8 章

8-5 演習問題

(執筆者：吉澤康文)［2013 年 2 月 受領］

(1) 仮想記憶ではプログラムのメモリ参照が性能に影響を与えるが，なぜか説明せよ．

(2) ワーキングセットを定義せよ．この定義から何が言えるかを述べよ．

(3) プログラム局所性とは何か説明せよ．

(4) ページリプレースメントアルゴリズムを 3 種類あげ, 各アルゴリズムの特徴を説明せよ．

(5) LRU アルゴリズムが有効なメモリ参照と無効なメモリ参照はいかなる場合か説明せよ．

(6) ページフォールト率とは何か.

(7) 多重仮想記憶方式の利点を述べよ．

(8) スワッピングを行う主たる目的を述べよ．