17 ページングの方式

一口にページングと言っても、

- フェッチの方式 · · · どのページをいつ主記憶に移すかを決める
- 置換えの方式 · · · 空きのページフレームを確保するために
  主記憶からどのページを退避するかを決める

に関して、いくつかの選択肢がある。

17–1 プリページング vs. デマンドページング

ページングの際のフェッチの方式としては、9.8節で説明したデマンドページングの他にプリページングと呼ばれる方法もある。

デマンドページング (または要求時ページング, オンデマンドページング):

- プロセス起動時,スワップイン時はページフレームの割り付けは一切行わず、
- プログラム実行時に参照されページフォールトになったページだけを随時主記憶に読み込で行く。

プリページング (または予測ページング):

- プロセス起動時,スワップイン時に、将来実行に必要となりそうなページの集合を前もってまとめて主記憶に読み込む。

デマンドページングの利点・欠点：

(利点) 利用しないページを主記憶に読み込むことがない。

(欠点) まとめて主記憶に読み込む訳ではないので、
1ページ当たりの(主記憶への)読み込み時間は、それ程少なくはならない。

プリページングの利点・欠点：

(利点) ディスク上の連続した領域から複数の領域をまとめて読み込むので、1ページ当たりの(主記憶への)読み込み時間は、1ページずつ読み込む時と比べてかなり少なくて済む。

(欠点) 将来必要なページを予測するのは一般には困難なので、利用しないページを主記憶に読み込んでしまう可能性がある。

プリページング vs. デマンドページング：

どちらが有利かは、プログラムのメモリ参照動作に依存して決まる。

- プリページングにおいて、プロセス起動時に将来必要なページを主記憶に読み込むのにかかる時間 $\mathbf{P(N)}$ は次の様に見積もることが出来る。

  $\mathbf{P(N)=\alpha N+c}$　　但し、$\mathbf{N}$ = プロセスの要求するページ数

- デマンドページングにおいて、ページアウトやスワップアウトによる再読み込みがないと仮定した場合に、ページ読み込みにかかる時間 $\mathbf{D(n)}$ は次の様に見積もることが出来る。

  $\mathbf{D(n)=\beta n}$　　但し、$\mathbf{n}$ = 参照するページ数 $(\leq \mathbf{N})$

⇨ 実メモリが十分にありページアウトやスワップアウトによる再読み込みがないと仮定した場合

デマンドページングの方がプリページングより有利

$\Longleftrightarrow \mathbf{D(n)<P(N)}$

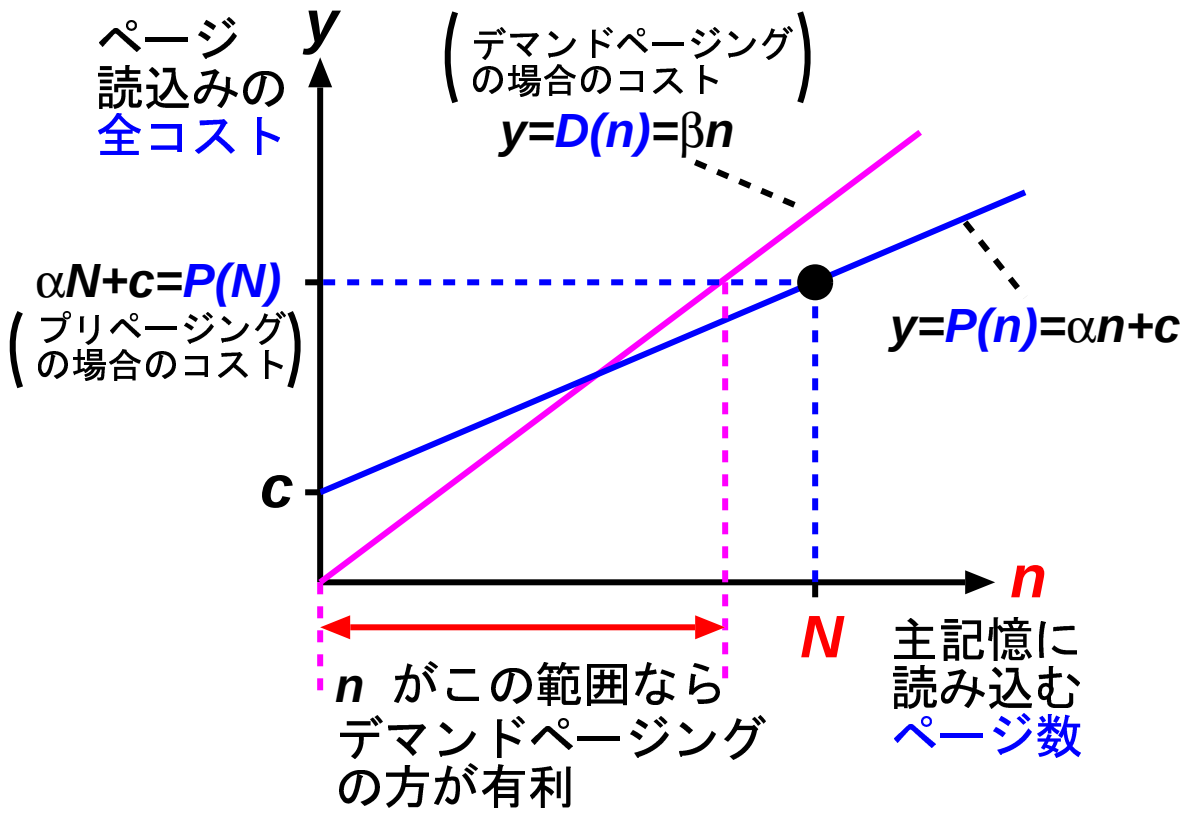
$\Longleftrightarrow \mathbf{n} < \frac{\mathbf{1}}{\beta}(\alpha\mathbf{N+c})$

⇨ 実メモリが十分にありページアウトやスワップアウトによる再読み込みがないと仮定した場合

デマンドページングの方がプリページングより有利

$\Longleftrightarrow \mathrm{D(n)<P(N)}$

$\Longleftrightarrow \mathrm{n} < \frac{1}{\beta}(\alpha\mathrm{N}+\mathrm{c})$



ページの再読み込みの頻度は、プログラムの動作する環境やページ置換えの方式 (i.e. 主記憶からどのページを退避させるかの方式) だけでなく、プログラムのページ参照の順番にも依存する。

17–2 プログラムのページ参照動作の傾向
——ワーキングセットと局所参照性——

ページ参照動作のパターン :

(ページングに有効なモリ参照の仕方)

. . . ページ再読み込みをあまり発生させない。

小さな領域を密度高く参照 する。

(無駄の多いメモリ参照の仕方)

. . . ページ再読み込みを頻繁に引き起こす。

広い領域を密度低く参照 する。例えば、

- 各ページ内の1バイトだけを参照する。
- 一度は参照するが、同じ領域を2度と参照しない。(逐次処理に多い。)
- 参照した後同一部分の再参照まである程度の時間がかかる。
- 広い領域をランダムに参照する。

ページ参照動作のパターン：

(ページングに有効なモリ参照の仕方)

...ページ再読み込みをあまり発生させない。

小さな領域を密度高く参照する。

(無駄の多いメモリ参照の仕方)

...ページ再読み込みを頻繁に引き起こす。

広い領域を密度低く参照する。例えば、.....

⇨ 近い将来使うページを優先的に主記憶に残すことが出来れば、ページ再読み込みはより少なくなる。

しかし、一般にはプログラムのメモリ 参照パターンは時間に伴って変化する。

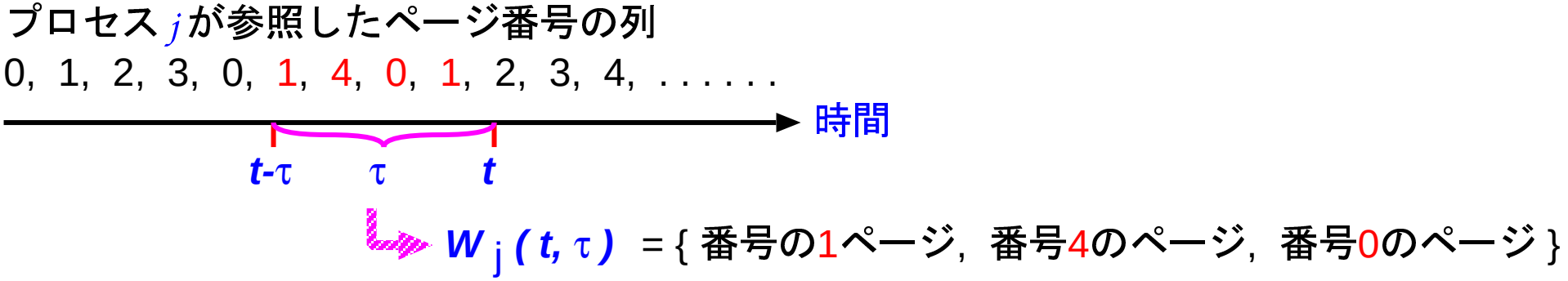
⇨ 各プロセスに割り付けた実ページの活用度を常時監視しないと、それらが将来参照される見込がどの程度あるかの判断が出来ない。

どういう基準で各ページの活用度を測るか：

最近の τ 時間(但しτはウィンドウサイズと呼ばれるパラメータ)の間にプロセスが参照したページは活用度が高く近い将来参照される可能性も高いと考え、それらの集合をワーキングセットと呼ぶ。

時刻 t におけるプロセス j のワーキングセット

$W_j(t,\tau)$＝{x| 時刻t以前にプロセスjに割り当てられた最近の
τ時間の間にプロセスjがページxを参照した}



$W_j(t,\tau)$=｛x| 時刻 t 以前にプロセス j に割り当てられた最近の
τ 時間の間にプロセス j がページ x を参照した｝

当然、ワーキングセット $W_j(t,\tau)$ はプログラムの実行に伴って変化する。

例えば、コンパイラは
(1) ソースコードの読み込み,
(2) 字句解析/構文解析,
(3) 中間言語 (e.g. アセンブラコード) への変換,
(4) オブジェクトコードの生成,
. . . . . .
という順に処理を進める。 この様に実行フェーズが変わる時点で、一般にワーキングセットは大きく変化する。

では、ワーキングセット内のページは本当に近い将来も参照される可能性が高いのか?

⇩

ページ参照の一般的な傾向 : 経験的な話

プログラムを実行した時のページ参照は多くの場合局所的である。
すなわち、

> プログラムは関連するページを均等に参照し続けることはまれで、通常は幾つかのページにアクセスが集中する。
>
> [但し、アクセスが集中するページの集合は時間の経過と伴に(特にプログラムの実行フェーズが変わる場合は大きく)変化する。]

このページ参照の傾向は多くの研究/経験から広く認識されており、プログラムの局所参照性(またはページ参照の局所性, ...)と呼ばれている。

⇨ ウィンドウサイズτをある程度以上に選べば、ワーキングセット $W_j(t,\tau)$ はあまり変動しない。

⇨ ワーキングセット内のページは近い将来も参照される可能性が高い。

補足：

ここで言う「ページ参照」とは、プログラムが変数領域のページを参照するということだけでなく、プログラムカウンタが実行コードの置かれたページを参照するということも含む。

結局のところ、局所参照性は次の2種類の局所(参照)性に起因する。

- 時間的局所性:

  一度参照された場所(データまたは命令)はすぐまた参照されることが多いという性質。

- 空間的局所性:

  一度参照された場所の近くがすぐまた参照される傾向にあるという性質。

補足：

局所参照性のほとんどないプログラムももちろん存在する。この様なプログラムは一般に仮想記憶方式では効率的な実行が出来ない。

17–3 ページ置換えアルゴリズムの必要性

多重プログラミング環境では、ページアウトの可能性は避けられない。

⇨ 主記憶から退避するページを決める必要がある。このための
アルゴリズムをページ置換えアルゴリズムと呼ぶ。

どんなページ置換えが良いか：

- 将来使う可能性の少ないページをページアウトの対象として選んで、ページの再読み込みを出来るだけ少なくすることが大事。
- 退避ページを決める計算の手間が大きくなり過ぎてもいけない。
- ページフォールトの度にページアウトとページインを行うというのでは、ページアウトを頻繁に行うことになり兼ねない。

⇨ 実ページの活用度を常時監視し、活用度の低い実ページをまとめて適宜ページアウトすることによって空きページを常時用意しておく。

⇨ 実ページの活用度を常時監視し、活用度の低い実ページをまとめて適宜ページアウトすることによって空きページを常時用意しておく。

ページ置換えアルゴリズムの起動時機：

UNIX では、空きのページフレームの割合が一定の値より小さくなると、ページデーモンと呼ばれるシステムプロセスが起動され参照されていないページを回収する。そして、

ページデーモンが使用するCPU時間が一定の割合を越えると、ページデーモンは待ち状態にあり主記憶を長時間使っているプロセスを選んでスワップアウトする。

17–4 グローバル方式 vs. ローカル方式

ページ置換えアルゴリズムはその適用範囲を主記憶内の局所的な範囲に限定するかどうかによって、次の2つの種類に分けることが出来る。

- グローバル方式 (i.e. 大域的な適用) :
  主記憶内の全てのページの中から退避ページを一定の基準で決める方式。
- ローカル方式 (i.e. 局所的な適用) :
  ページを取り上げるアドレス空間をまず選択し、その空間に属するページの中から退避ページを決定する方式。

例えば、
各プロセスに固定長のメモリを割り当て、各々のプロセスに割り当てられたメモリ領域内でページフォールトに対する処置を施す、というのはローカル方式に属する。

17–5 ページ置換えの基本的な方式

基本的なページ置換えアルゴリズムを以下に列挙する。

(1) **OPTアルゴリズム** (**OPTimum**; または**MIN**):

最も遠い将来まで参照されないページを退避する方式。

⇨
- その名の通り、最適なページ置換えアルゴリズムである。

  しかし、
- プログラムの将来の動作についての完全な情報を必要とするので、実装不可能である。
- 提案されたページ置換えアルゴリズムの性能評価の基準を与えている。

(2) RANDOMアルゴリズム：

主記憶内からランダムに選んだページを退避する方式。

⇨
- グローバル方式を適用する。
- 実装は非常に簡単。
- ページフォールトの数がOPTの3倍以上になったという実験結果もある。
- アルゴリズムの性能評価の際の出発点。

⇨

RANDOM と OPT の間で出来るだけOPTに近く、かつ実行効率も良いページ置換えアルゴリズムを如何にして構成できるかが問題になる。

(3) FIFOアルゴリズム (First-In-First-Out):

主記憶内に最も長くいるページを退避する方式。

⇨
- グローバル方式もローカル方式も可能。
- 容易に実装できる。

理由：
ページインした時刻の早い順にページ番号を並べた線形リストを常に保持すれば良い。ページインの度にリストを更新するだけなので、オーバーヘッドはそれ程大きくはならない。

しかし、

- 頻繁に継続的に使われるページがあったとしても、そのうち最も古くなりその時点で退避されてしまう、という致命的な欠点がある。
- 実記憶の容量を増やしたのにページフォールトが増える、という奇妙な現象が起こり得る。(Beladyの異常現象 $\Longrightarrow$ 17.7節)
- ページフォールトの数がRANDOMと大差ないという実験結果もある。

___
- 純粋な形ではほとんど採用されていない。

(4) LRUアルゴリズム (Least Recently Used):

最も長い時間参照されていないページを退避する方式。

⇨ • グローバル方式もローカル方式も可能。

但し、
各プロセスに固定した個数のページフレームを割り当てるローカルなLRU方式では、プロセス間でページを融通し合うことが出来ないため柔軟性に乏しく、仮想記憶の利点が十分に活かされない。

• 大抵のプログラムは時間的局所性を持っているので、このLRUアルゴリズムは普通の場合良好に働く。

しかし、

しかし、

- 特殊なハードウェアがないと、このLRUアルゴリズムを効率良く実装することは困難である。

理由：
参照されていない時間の長い順にページ番号を並べた線形リストを常に保持すれば確かに実装は出来るが、これでは、メモリ参照の度にこのリストを更新する必要があるためオーバヘッドが大きくなり過ぎてしまう。

⇨ 一般には近似方式が用いられる。 (⟹ 17.6節)

- グローバルLRUアルゴリズム(の近似方式)は比較的よく用いられる。

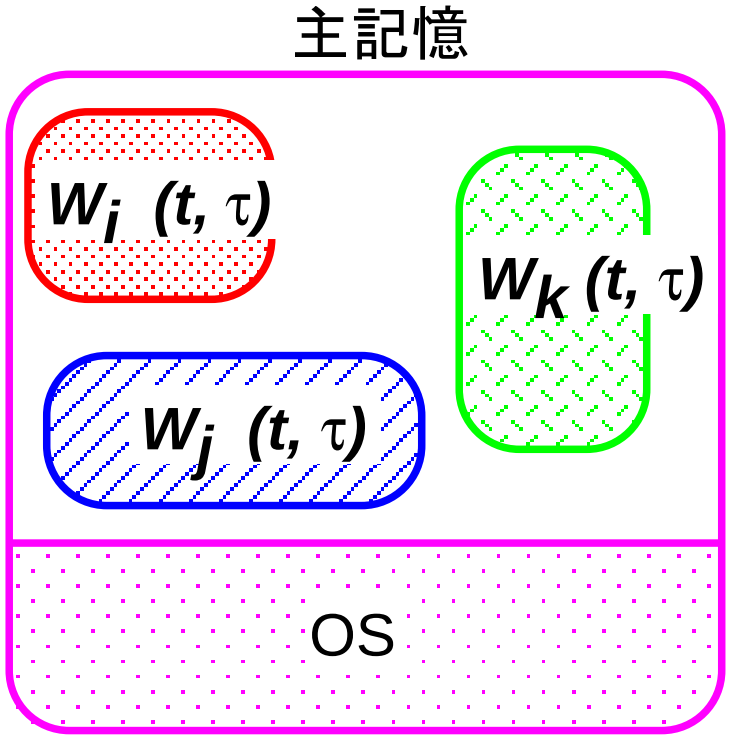
(5) ワーキングセット法：

各プロセスのワーキングセットを参考にして、退避ページを決定するだけでなく適宜スワップアウトも行い多重プログラミングの多重度の調節も行う方法である。

具体的には、実行可能状態にあるプロセスのワーキングセット

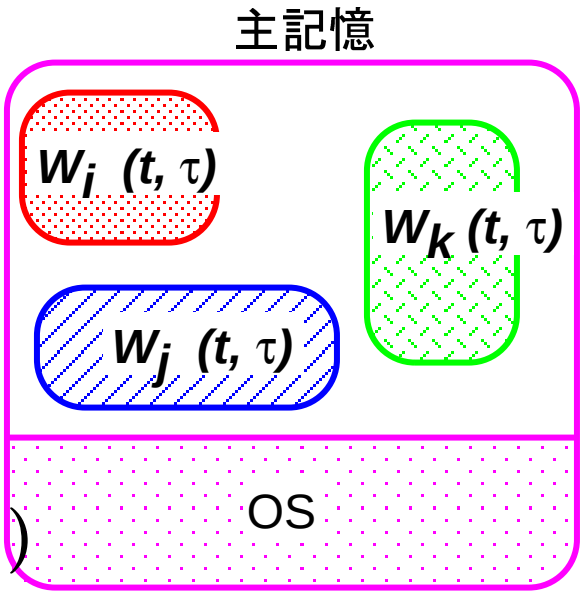
$$\bigcup_{\text{j:実行可能状態のプロセス}} W_j(t,\tau)$$

内にあるページは全て主記憶内に保持することに決め、残りのページから退避ページを(例えばLRU法で)選ぶ。



その際、「残りのページ」が無ければプロセスをどれかスワップアウトする。

(その後、メモリに余裕が出来ればスワップイン。)



その際、「残りのページ」が無ければ
プロセスをどれかスワップアウトする。

(その後メモリに余裕が出来ればスワップイン。)

ウィンドウサイズτの選び方：

個々のプログラムの実行時間を考えるとτは大きい方が良いが、

多重プログラミングの多重度を上げるためにはτは小さい方が良い。

⇨ 実際には、ウィンドウサイズτは経験的に適当な値に設定され、システム共通に使われることが多い。

⇨ • プロセスに割り当てられるページフレームの個数は動的に変化する。

• グローバル方式もローカル方式も可能。

その理由：
時々、使われてなさそうなページ群をページアウトする。その際、プロセス毎の基準で (local に) ワーキングセット外のページを選んで退避することも、全体に統一された基準でワーキングセット外のページを選んで退避することも可能である。退避ページのないプロセスがあっても良い。

• 大抵のプログラムは時間的局所性を持っているので、このアルゴリズムは普通の場合良好に働く。

しかし、

• ワーキングセット法を厳密に実装することは計算効率の点で難しい。

⇨ 一般には近似方式が用いられる。 (⟹ 17.8 節)

17–6 LRUアルゴリズムの近似的な実装
—NRU法,FINUFO法,LFU法,...—

p.250からの引用：
特殊なハードウェア(e.g. A.S.タネンバウム3.4.6節)がないと、このLRUアルゴリズムを効率良く実装することは困難である。
⇨ 一般には近似方式が用いられる。

ここでは、LRUアルゴリズムの近似方式を列挙する。

(1) **LFUアルゴリズム** (Least Frequently Used):
参照頻度が最小のページを退避する方式。

⇨
- グローバル方式もローカル方式も可能。
- ページ毎のアクセス数を数えるハードウェアがないと効率的な実装は出来ない。

(2) NRUアルゴリズム (Not Recently Used):

ページングによる仮想記憶を採用しているコンピュータでは、各ページの使用状況を記録するために次の2つのフラッグビットがページ表の中に用意されることが多い。

$$\text{参照ビット } R=\begin{cases}1 & \text{if ページが参照された}\\ 0 & \text{otherwise}\end{cases}$$

$$\text{更新ビット } C=\begin{cases}1 & \text{if ページが修整された}\\ 0 & \text{otherwise}\end{cases}$$

通常、これらのフラッグはページ参照の度にハードウェアが自動的に更新する。また、

参照ビットの方は最近使われていないページを明らかにするためにオペレーティングシステムが定期的に0にセットすることもある。

このNRU方式では、.....

このNRU方式では、
これらの情報を使ってアルゴリズム適用の範囲内のページを

下位 ↑↓ 上位

(クラス0) 参照も修整もされていない,
(クラス1) 参照されてないが修整されている,
(クラス2) 参照されているが修整されてない,
(クラス3) 参照も修整もされている,

補足：
参照ビットの方は時々0にセットされるのでこういうことも起こる。

の4つに分類し、最下位の空でないクラスからランダムにページを選んで退避する。

⇨
- グローバル方式もローカル方式も可能。
- 容易に実装できる。
- 性能は最適ではないが、大抵の場合十分である。

(3) セカンドチャンス アルゴリズム：

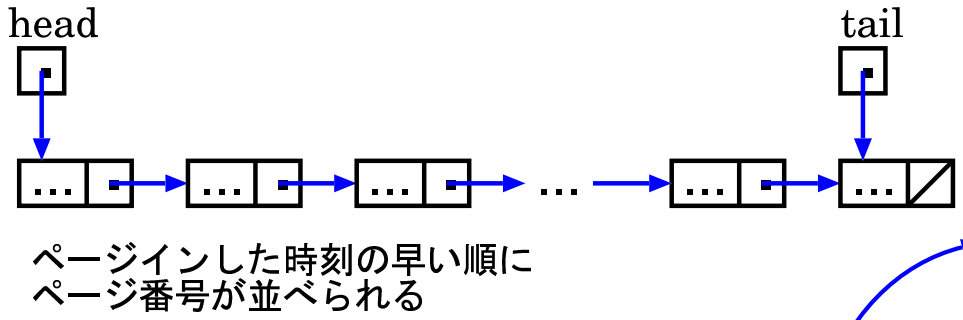
FIFO法の場合と同様に、ページインした時刻の早い順にページ番号を並べた線形リストの様なものを常に保持する。
しかし、
FIFO法と違ってこの方式では、
①リストの先頭ページの参照ビットが 1 である間そのビットを 0 に修正した上でそのページをリストの末尾に移す、という作業を繰り返す。 そして、

②リストの先頭に参照ビットが 0 のページが現れれば、それを退避ページとして選ぶ。

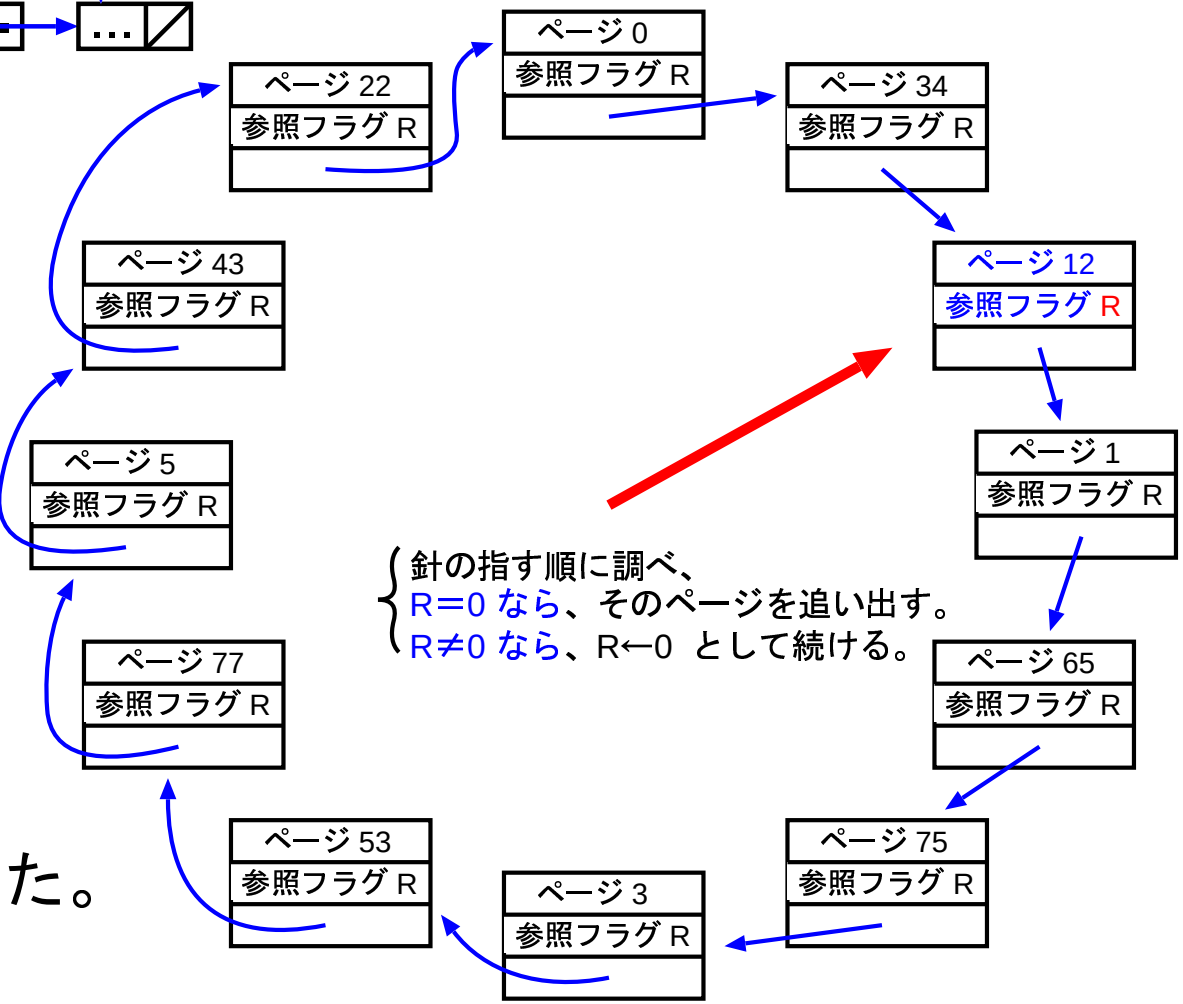
⇨
- グローバル方式もローカル方式も可能。
- 比較的容易に実装できる。
- 合理的だが、必ずしも計算効率が良いとは言えない。
- 全てのページが参照されていれば、FIFO法と同じ判断結果をもたらす。

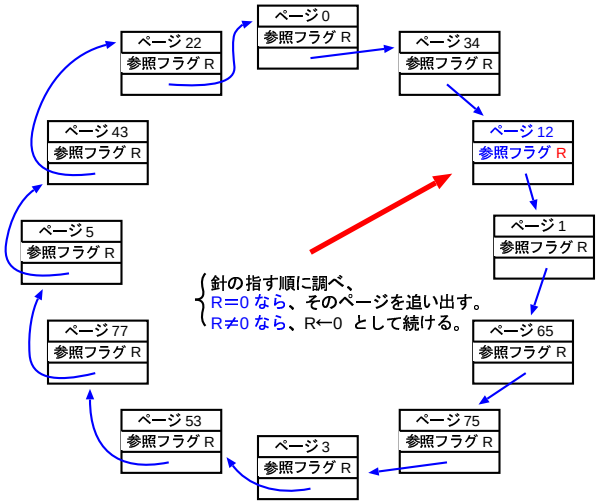
(4) クロック アルゴリズム (FINUFO法, First-In-Not-Used-First-Out):

セカンドチャンスアルゴリズムの計算効率を改善した方法である。



ページ番号を並べた線形リストの先頭にあるページ(番号)を絶えずリストの末尾に移動させる非効率を避けるために、リストをリング状に構成し、ポインタ(時計の針)で次の退避候補を指す様にした。





具体的な処理内容としては、

①針の指すページの参照ビットが 1 である間

そのビットを 0 に修正した上で針を順方向に進める、という作業を繰り返す。 そして、

②針の先に参照ビットが 0 のページが現れれば、

それを退避ページとして選び(次の処理機会のために)針を1つ進める。

⇨ • セカンドチャンスアルゴリズムとの違いは実装方法だけである。

• グローバル方式もローカル方式も可能。

• MulticsやVAX上のUNIX, IBMのVM/370等で採用。

(5) 未参照カウンタを用いた擬似LRU法：

ページ毎に非負整数を記憶するためのカウンタを用意する。そして、将来の退避ページを選出する時に備えて次の作業を定期的に繰り返す。

> 各々のページの参照ビットを調べて、
> もし参照ビットが0ならそのページのカウンタを1増加させ、
> また、　もし参照ビットが1ならそのページのカウンタと参照ビットを0にクリアする、

こうしておいて、
退避ページを選ぶことになれば、その時点でカウンタ値の最も大きなページを選んで退避させる。

⇨
- カウンタ値の大きい順にページを並べたものは、ほぼ参照されない時間の長い順に並ぶことになるので、確かにLRU法の近似になっている。
- グローバル方式もローカル方式も可能。
- 実装は、比較的容易。

- 実装は、比較的容易。

例えば：
カウンタ値の大きな順にページ番号を並べた線形リストの様なものを常に保持すれば良い。

17–7 スタックアルゴリズム

一般に、実記憶の容量を増やしてもページフォールトが減るとは限らない。ページ置換えアルゴリズムによっては、逆にページフォールトが増える場合もある。これをBeladyの異常現象と呼んでいる。

例17. 1 (Beladyの異常現象;FIFOアルゴリズムの異常現象)
実記憶の容量を増やしてページフォールトが増えることもある。
例えば、
番号が0〜4の5個の仮想ページを持つプログラムが

$$0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4$$

というページ番号の順に仮想ページを参照する場合を考えよう。

FIFOページ置換えにおいては、.....

(もしページフレームが3個なら)

実記憶内に蓄えられるページは次の様に変わって行くので、ページフォールトの回数は9 回である。

ページ参照の列

		0	1	2	3	0	1	4	0	1	2	3	4
		↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
		page fault	page fault	page fault	page fault	page fault	page fault	page fault			page fault	page fault	
最新のページ		0	1	2	3	0	1	4	4	4	2	3	3
			0	1	2	3	0	1	1	1	4	2	2
最も古い...				0	1	2	3	0	0	0	1	4	4

⟶ 時間

一方、

(もしページフレームが4個なら)

.....

一方、

(もしページフレームが4個なら)

実記憶内に蓄えられるページは次の様に変わって行くので、ページフォールトの回数は10 回となり、ページフレームが3個の場合より多い。

ページ参照の列

	0	1	2	3	0	1	4	0	1	2	3	4
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
	page fault	page fault	page fault	page fault			page fault	page fault	page fault	page fault	page fault	page fault
最新のページ		0	1	2	3	3	3	4	0	1	2	3	4
			0	1	2	2	2	3	4	0	1	2	3
				0	1	1	1	2	3	4	0	1	2
最も古い...					0	0	0	1	2	3	4	0	1

⟶ 時間

Beladyの異常現象を起こさないページ置換えアルゴリズムも存在する。例えば、スタックアルゴリズムと呼ばれる種類のページ置換えアルゴリズムはそういう性質を持つことが数学的に保証されている。

定義17. 2 (スタックアルゴリズム)

ページ参照要求の順に参照ページの番号を並べた列をページ参照列と呼ぶ。また、ページ置換えアルゴリズム A とページ参照列 w, ページ容量を表す正整数 m に対してページの集合 S(A,m,w) を

$$S(A,m,w)=\begin{cases} m \text{ ページを記憶できる容量の主記憶 (最初は空) において、ページ参照列 } w \text{ に対してデマンドページング／ページ置換えアルゴリズム } A \text{ が適用された直後に、主記憶に存在するページ (番号) の集合} \end{cases}$$

と定める。 この時、もし任意のページ参照列 w と正整数 m に対して

$$S(A,m,w) \subseteq S(A,m+1,w)$$

であるなら、ページ置換えアルゴリズム A はスタックアルゴリズムであると言う。

例17. 3 (スタックアルゴリズム)

例17.1から、FIFOアルゴリズムにおいては

$$\begin{aligned} & S(\mathrm{FIFO},\ 3,\ \text{"0123014"}) \\ & = \{4,\ 0,\ 1\} \\ & \not\subseteq \{4,\ 3,\ 2,\ 1\} \\ & = S(\mathrm{FIFO},\ 4,\ \text{"0123014"}) \end{aligned}$$

であることが分かるので、FIFOはスタックアルゴリズムではない。

一方、LRUアルゴリズムにおいては、任意のページ容量 m とページ参照列w に対して

$$\begin{aligned} & S(\mathrm{LRU},\mathrm{m},\mathrm{w}) \\ & = (\text{wの中で最近参照された互いに異なる m 個のページの集合}) \\ & \subseteq (\text{wの中で最近参照された互いに異なる m+1 個のページの集合}) \\ & = S(\mathrm{LRU},\mathrm{m}+1,\mathrm{w}) \end{aligned}$$

であるのでLRUはスタックアルゴリズムである。

他に、LFUやOPTもスタックアルゴリズムである。

—

命題17. 4

スタックアルゴリズムにおいては、実記憶の容量を増やした時ページフォールトの回数が増えないことが保証される。

[証] スタックアルゴリズム A においては定義から必ず

$$\mathbf{S(A,m,w)} \subseteq \mathbf{S(A,m+1,w)}$$

であるので、

> ページ参照列 $\mathbf{w}$ の処理を行った直後に
> 容量が $\mathbf{(m+1)}$ の実記憶でページフォールトが起きるなら
> 容量が $\mathbf{m}$ の実記憶の時もページフォールトが起きる

はずである。 これが全てのページ参照列 $\mathbf{w}$ に対して成り立つ訳だから、

容量が $\mathbf{(m+1)}$ の時のページフォールト回数
$\leq$ 容量が $\mathbf{m}$ の時のページフォールト回数

でなければならない。 □

主記憶内のページ構成：

ページ置換えアルゴリズムAがスタックアルゴリズムなら当然

$$|S(A,m,w)| \leq |S(A,m+1,w)| \leq |S(A,m,w)|+1$$

であるから、

$$S(A,m+1,w)=S(A,m,w)$$

または

$$S(A,m+1,w)=S(A,m,w)\cup\{p\} \quad \text{for some } p$$

である。そこで、i番目に参照されるページ番号がa_iで, ページ参照列が

$$w_t = a_1 a_2 a_3 \cdots a_t$$

と書けたとして、

$$p(A,m,w_t) \stackrel{\text{def}}{=} \begin{cases} \text{空白} & \text{if } S(A,m,w_t)=S(A,m\text{-}1,w_t) \\ S(A,m,w_t)-S(A,m\text{-}1,w_t)\text{内の要素} & \\ (\text{i.e. } S(A,m,w_t)=S(A,m\text{-}1,w_t)\cup\{p\}\ \text{となる様な } p) & \\ & \text{if } S(A,m,w_t)\neq S(A,m\text{-}1,w_t) \end{cases}$$

とすると、逆に主記憶内のページ(番号)の集合 $S(A,m,w_t)$ は

$$S(A,m,w_t)=\{p(A,1,w_t),\ p(A,2,w_t),\ \ldots,\ p(A,m,w_t)\}$$

と表すことができる。

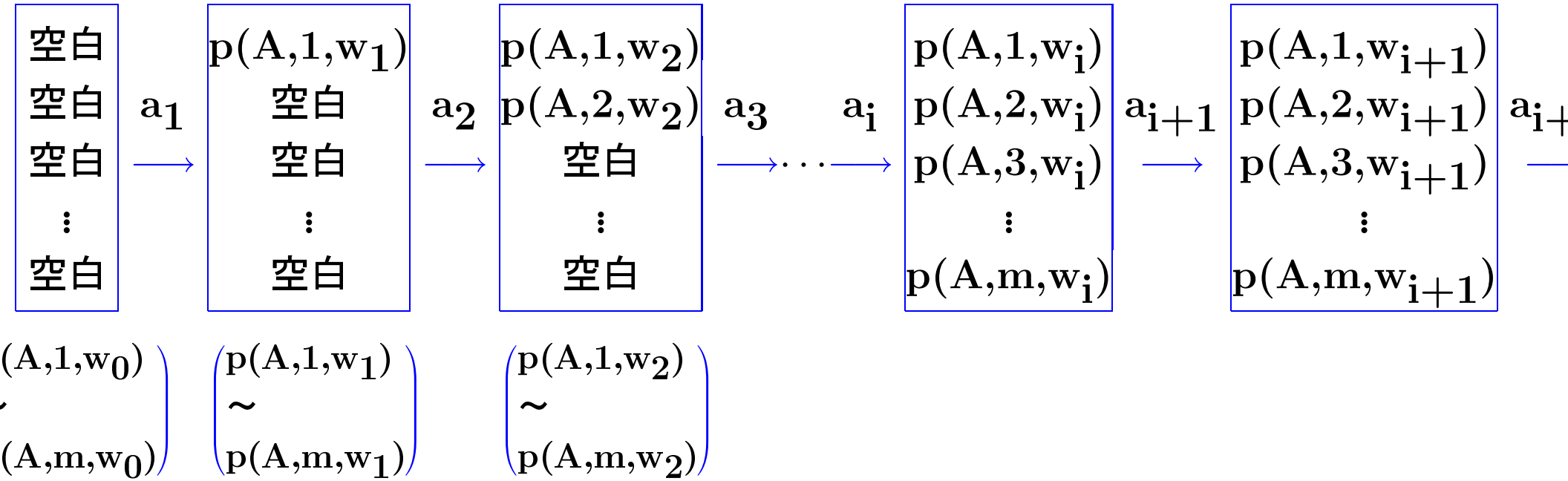
.....

とすると、逆に主記憶内のページ(番号)の集合は

$$S(A,m,w_t)=\{p(A,1,w_t),\ p(A,2,w_t),\ \ldots,\ p(A,m,w_t)\}$$

と表すことができる。

⇨ 主記憶の容量が m で、番号が$a_1,a_2,a_3,\ldots$ のページを次々と参照した時、主記憶のページ構成の変遷は次の様に表すことができる。



命題17. 5　$p(A,m,w_t)$=空白 $\Longleftrightarrow$ $|\{a_1,a_2,\ldots,a_t\}|<m$

LRUスタック：

スタックアルゴリズム A の場合に、
ページ参照列 $w_t = a_1 a_2 \cdots a_t$ で参照
された $n_t \stackrel{\text{def}}{=} |\{a_1, a_2, \ldots, a_t\}|$ 個の
ページの番号をスタック状に積んだ右の
形のものを(ここでは)A-スタックと呼ぶ。

$p(A,1,w_t)$
$p(A,2,w_t)$
⋮
$p(A,n_t,w_t)$

特にLRUアルゴリズムの場合は、これはLRUスタックと呼ばれ、最終参照時刻の古い順にページ番号を積んだスタックになる。

なぜなら、

$$S(LRU,m,w_t) = \left(\begin{array}{l} w_t \text{の中のページを最終参照時刻の遅い順に並べ} \\ \text{た時の上位 } m \text{ 番目までのページ(番号)の集合} \end{array}\right)$$

となり、それゆえ

$$p(LRU,m,w_t) = \begin{cases} \left(\begin{array}{l} w_t \text{の中のページを最終参照時刻の遅い} \\ \text{順に並べた時の} m \text{番目のページ番号} \end{array}\right) & \text{if } m \leq n_t \\ \text{空白} & \text{otherwise} \end{cases}$$

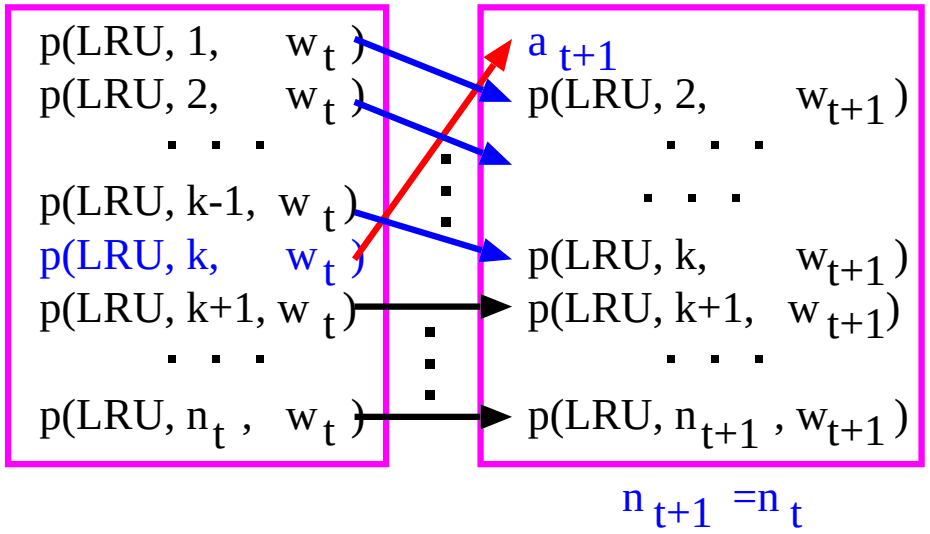
となるからである。

—

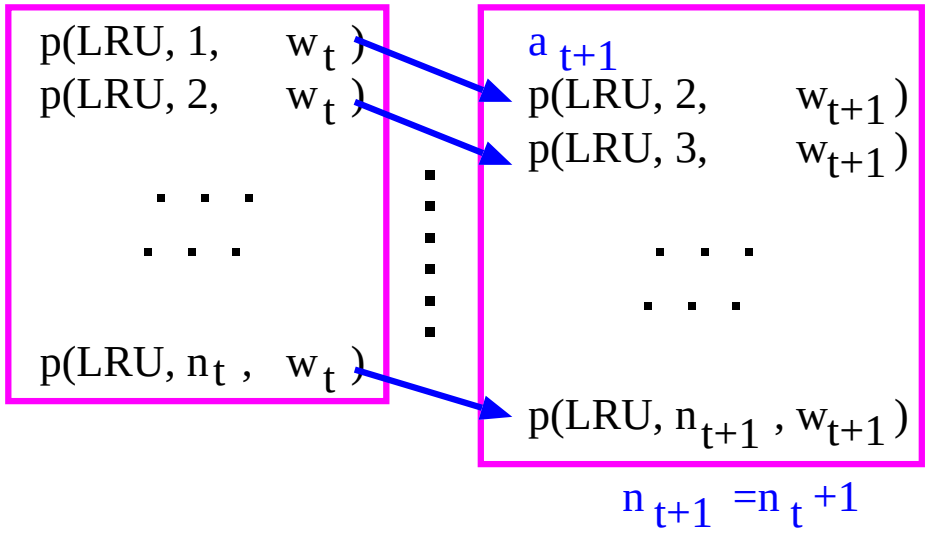
LRUスタックの変遷は次の漸化式に基づいて簡単に行うことができる。

$$p(LRU,m,w_t) = \begin{cases} a_t & \text{if } m=1 \\ p(LRU,m-1,w_{t-1}) & \text{if } m \geq 2,\ a_t \notin \{p(LRU,1,w_{t-1}),\ldots, p(LRU,m-1,w_{t-1})\} \\ p(LRU,m,w_{t-1}) & \text{otherwise} \end{cases}$$

($\exists$k) $p(LRU,k,w_t) = a_{t+1}$ の場合

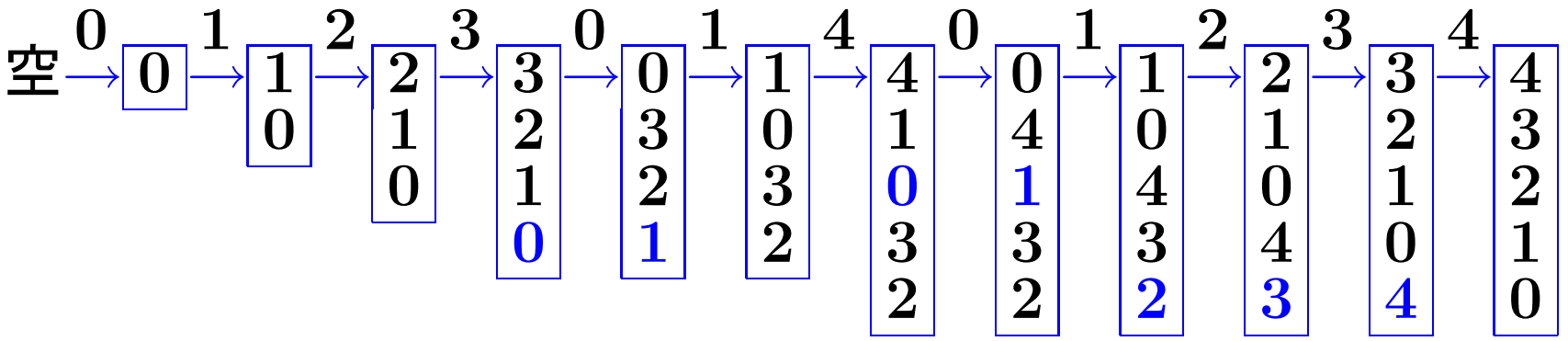


($\forall$k) $p(LRU,k,w_t) \neq a_{t+1}$ の場合



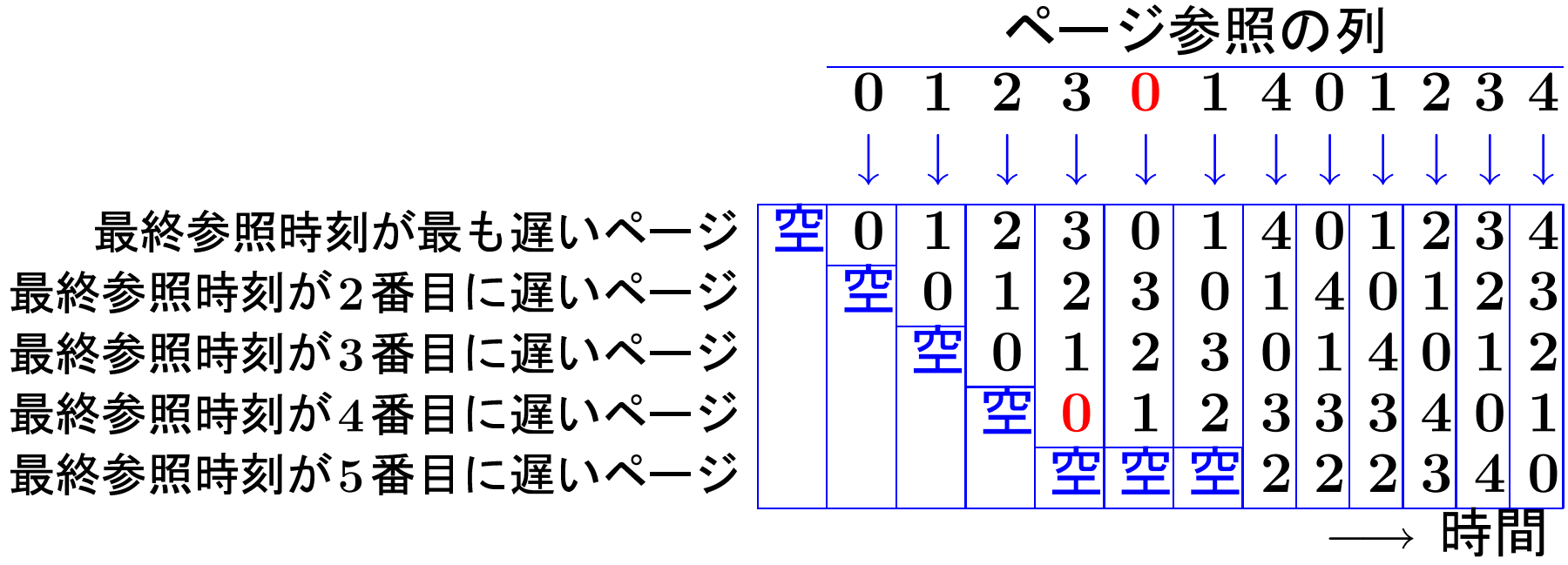
例17. 6 (LRUスタックの変遷)

例17.1で考えたページ参照列 0,1,2,3,0,1,4,0,1,2,3,4 に対してLRUアルゴリズムを適用した時、LRUスタックは次の様に変わってゆく。



あるいは、例17.1で示した表の様に書き表すと、

	ページ参照の列											
	0	1	2	3	0	1	4	0	1	2	3	4
最終参照時刻が最も遅いページ	空	0	1	2	3	0	1	4	0	1	2	3	4
最終参照時刻が2番目に遅いページ		空	0	1	2	3	0	1	4	0	1	2	3
最終参照時刻が3番目に遅いページ			空	0	1	2	3	0	1	4	0	1	2
最終参照時刻が4番目に遅いページ				空	0	1	2	3	3	3	4	0	1
最終参照時刻が5番目に遅いページ					空	空	空	2	2	2	3	4	0
								⟶ 時間					



命題17. 7

主記憶のページ容量が m でページ参照列 $w=a_1a_2a_3\cdots$ に対してスタックアルゴリズム A を適用した時、

時刻 t におけるページ参照 a_t に対してページフォールトになる
$\Longleftrightarrow$ a_t はその時のA-スタックのtopから m 個の中にない。

ページフォールト回数の見積り：

スタックアルゴリズム A においては、(もし関数 $p(A,m,w_t)$ が計算可能なら)与えられたページ参照列 $w=a_1a_2a_3\cdots$ を1回走査するだけで全ての主記憶容量 m に対するページフォールト回数を計算できる。

なぜなら、各々の時刻 t においてスタック距離と呼ばれる量

$$d_t=\begin{cases}\left(\begin{array}{l}a_t\text{を読む直前のAスタック内で}\\a_t\text{の現れる位置}\\\text{(i.e.topから何番目に現れるか)}\end{array}\right) & \text{if}\ \left(\begin{array}{l}a_t\text{を読む直前のA}\\\text{スタック内に}a_t\text{が}\\\text{現れる}\end{array}\right)\\ \infty & \text{otherwise}\end{cases}$$

を計算しておけば、命題17.7から、

$$\left(\begin{array}{l}\text{主記憶の容量が } m \text{ の時に時刻 } t \text{ における}\\\text{ページ参照 } a_t \text{ に対してページフォールトになる}\end{array}\right)\iff d_t > m$$

となるので、ページフォールトの回数が次の様に計算できるからである。

主記憶容量が m の時のページフォールト回数
$=$ スタック距離が m を越える時刻の回数
$=|\{t \mid d_t > m\}|$

例17. 8 (LRUアルゴリズムにおけるページフォールト回数の見積り)

ページ参照列 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4 に対してLRUアルゴリズムを適用した時、各時刻におけるスタック距離 d_t と主記憶容量 m=1～5 に対するページフォールト回数は次の様に計算できる。

ページ参照の列

			0	1	2	3	0	1	4	0	1	2	3	4	
スタック距離 d_t			∞	∞	∞	∞	4	4	∞	3	3	5	5	5	
page fault の有無	m=1 の場合		F	F	F	F	F	F	F	F	F	F	F	F	⇒fault 回数=12
	m=2 の場合		F	F	F	F	F	F	F	F	F	F	F	F	⇒fault 回数=12
	m=3 の場合		F	F	F	F	F	F	F	—	—	F	F	F	⇒fault 回数=10
	m=4 の場合		F	F	F	F	—	—	F	—	—	F	F	F	⇒fault 回数=8
	m≥5 の場合		F	F	F	F	—	—	F	—	—	—	—	—	⇒fault 回数=5
最も最近使ったページ		空	0	1	2	3	0	1	4	0	1	2	3	4	
2番目に最近使った...			空	0	1	2	3	0	1	4	0	1	2	3	
3番目に最近使った...				空	0	1	2	3	0	1	4	0	1	2	
4番目に最近使った...					空	0	1	2	3	3	3	4	0	1	
5番目に最近使った...						空	空	空	2	2	2	3	4	0	

⟶ 時間

17–8 ワーキングセットアルゴリズムの近似的な実装

(1) ページフォールト頻度法 (PFF 法):

各プロセスのページフォールトの頻度がある範囲に収まる様に、プロセスに割り当てられるページフレームの個数を調節する。

具体的には、
プロセス毎にページフォールト間のプロセス時間を測り、

.....

具体的には、プロセス毎にページフォールト間のプロセス時間を測り、

① もしそれが指定された上限の時間 UT を越えていたら、そのプロセスには沢山のページを割り当て過ぎていると判断してそれらのページフォールト間に参照されなかったページを全て退避する。

② もしページフォールト間のプロセス時間が指定された下限の時間 LT を下回っていたら、ページ退避は行わずページフォールトを起こしたページをページインするだけにする。

③それ以外の場合は、LRU アルゴリズム等によって当該プロセスのページの中から退避ページを 1 つ選ぶ。

これらの処理の際に、必要なページフレームが確保できない場合は、いずれかのプロセスをスワップアウトする。

⇨
- ワーキングセット法の変形版である。
- プロセスに割り当てられるページフレームの個数は動的に変化。
- ローカルな適用しか出来ないのでは？
- ワーキングセット法より実装が容易。

(2) 未参照カウンタを用いた擬似ワーキングセット法：

ページ毎に非負整数を記憶するためのカウンタを用意する。そして、将来の退避ページを選出する時に備えて、実行可能状態のプロセスがCPUを例えば$\frac{1}{3}\tau$時間(但しτはウィンドウサイズ) だけ消費する度に

> そのプロセスに割り当てられた各々のページの参照ビットを調べ、
> もし参照ビットが0なら
> 　　そのページのカウンタを1増加させ、
> もし参照ビットが1なら
> 　　そのページのカウンタと参照ビットを0にクリアする、

という作業を繰り返す。

こうしておけば、各時点におけるワーキングセットを次の様に近似でき、これを基にページ置換え／スワップアウトを行うことができる。

$$W_j(t,\tau) \approx \{x \mid x\text{はプロセス } j \text{ に割り当てられたページ, 時刻 } t \text{ においてページ } x \text{ のカウンタ値が2以下}\}$$