# 9 メモリ管理

## 9–1 主記憶を管理する上での、課題の変化

主記憶の容量によって、管理の目標が少し変わってくる。

- 初期のコンピュータ： 主記憶の容量が小さい。

⇨ OSの課題：

◇ 大きなプログラムを実行できる様にする。

◇ 限られたメモリ領域を有効に活用した上で、
多重プログラミングの多重度を向上する。

⇨ 仮想記憶

- 現在： 半導体の集積化技術の急激な進歩によって、メモリが大容量化した。

例えば、
HITAC 8350: 200KB $\Longrightarrow$ 普通のPC: 4GB,
(1975頃教育用電算機) 情報工学科実習室のWS: 64GB

⇨ 大容量メモリを用いた性能向上技術が開発された。
例えば、

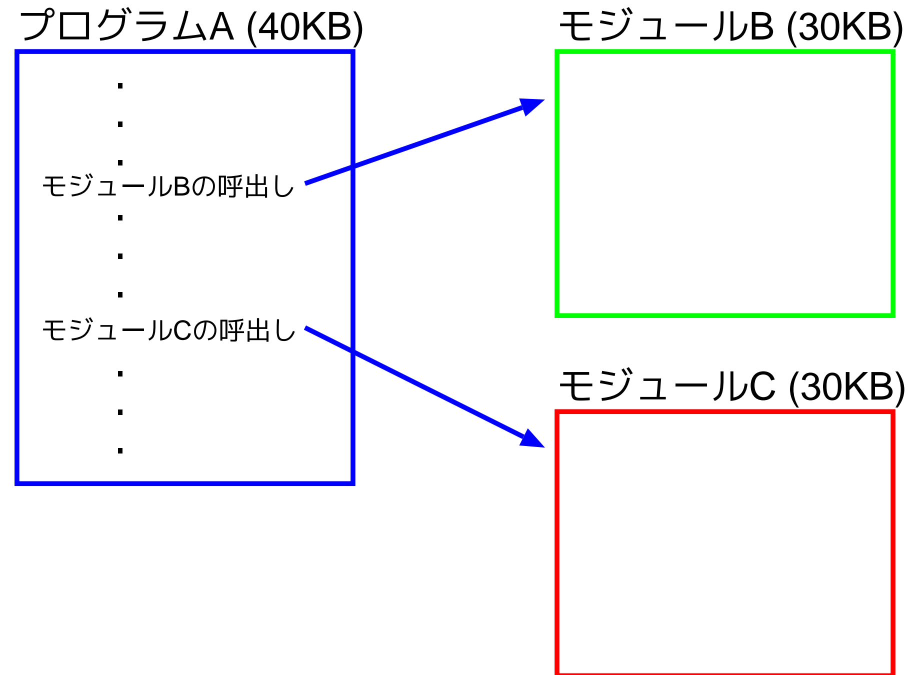
キャッシング … 主記憶に使用しているRAMの一部を磁気ディスク(RAMディスクという)と見なして、頻繁にアクセスされる情報をRAM内に格納しておく。

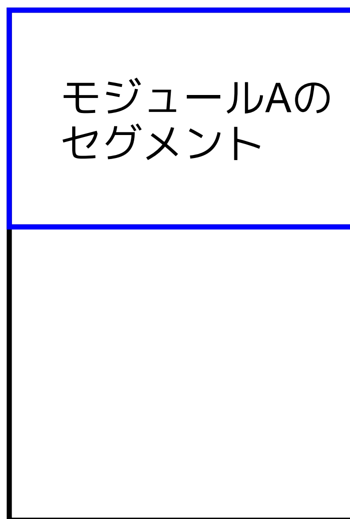
# 9–2 オーバレイ
## —大きなプログラムを実行するための昔の技法—

仮想記憶の機構が備わっていない計算機においては、(実メモリ空間より)
大きなプログラムを走行させるために、
オーバレイ (**overlay,** 重ね割付け, プログラムの重ね合せ)
と呼ばれる手法が用いられていた。

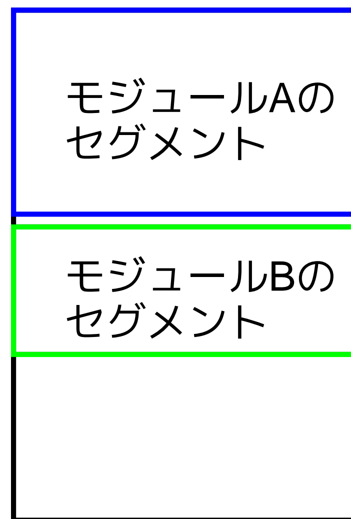
## オーバレイの基本的な考え方：

もしBとCの2つのモジュールが同時に主記憶に存在する必要が無ければ、2つのモジュールBとCには主記憶上の同じ記憶領域を割り当て、モジュールBとCのセグメントをその領域に切替えて格納することが出来る。

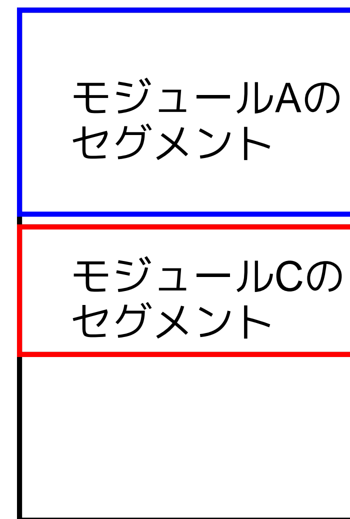



(a) モジュールBを呼び出す前の主記憶の状況




(b) モジュールBを呼び出した直後の主記憶の状況




(c) モジュールCを呼び出した直後の主記憶の状況

## オーバレイ構造の決定：

- オーバレイ構造はプログラムの呼出し関係によって定まる。
- リンケージエディタが呼出し関係を基に自動的に作成していた。
- ユーザが細かな指示を与えることも出来た。

## オーバレイ手法の欠点：

- 各々の時点でどの程度メモリが使えるかによって、どの程度オーバレイする必要があるのかが変わって来る。
- **2**つのモジュールがオーバレイ出来るかどうかはプログラムの進行状況にも依存するので、一般には予め正確には分からない。
- オーバレイ機能を有効に利用するには、プログラムを適当な個数のセグメントに分割し、それらが全体として効率良く動作するようなオーバレイ構造を与えなければならない。リンケージエディタへの細かな指示も必要。

⇨ 「セグメンテーション」による仮想記憶へと繋がる。

# 9–3 メモリ管理の初期の課題

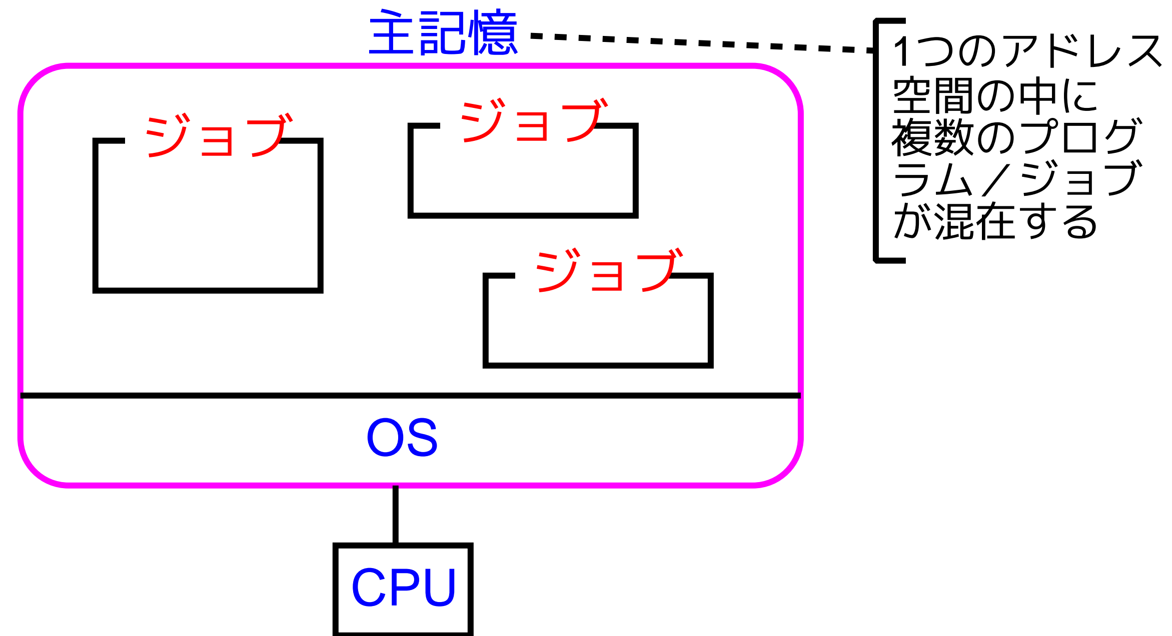
OS第2世代：多重プログラミング … (p.28からの引用)

- 入出力とCPUの実行を独立に並行して行わせるハードウェアが出現。
- チャネルからCPUへの連絡は(入出力完了)割込みによって行う。
- チャネルと割込みによって、複数のジョブを並行して動作させ(多重プログラミングと言う)CPUの有効利用が出来るようになった。

(入出力になったジョブは休止させCPUに別のジョブを実行させる。これによって、CPUが遊び状態のまま入出力の終了を待つという非効率を回避できる。)

⇨ 多重プログラミングの多重度をどうやって上げるかが、CPUの効率的な利用の鍵を握る重大な問題となった。

- CPU資源を使いこなすには
  多重プログラミングの多重度を充分に上げる必要がある。 そのためには、同時に多数のプログラム／ジョブを主記憶内に置かなければならない。




- しかし、一方では主記憶の容量は有限である。

> 補足：
> 1960年代(~1970年代)は主記憶(コアメモリ)は高価で空間的な制約もあるので、大容量は望めなかった。 それゆえ、多重度を充分上げてCPUを100%使い切ることは出来なかった。

⇨ メモリ管理の基本的な課題：
高価で限られた容量の主記憶の中に充分多くのジョブを詰めこむ。

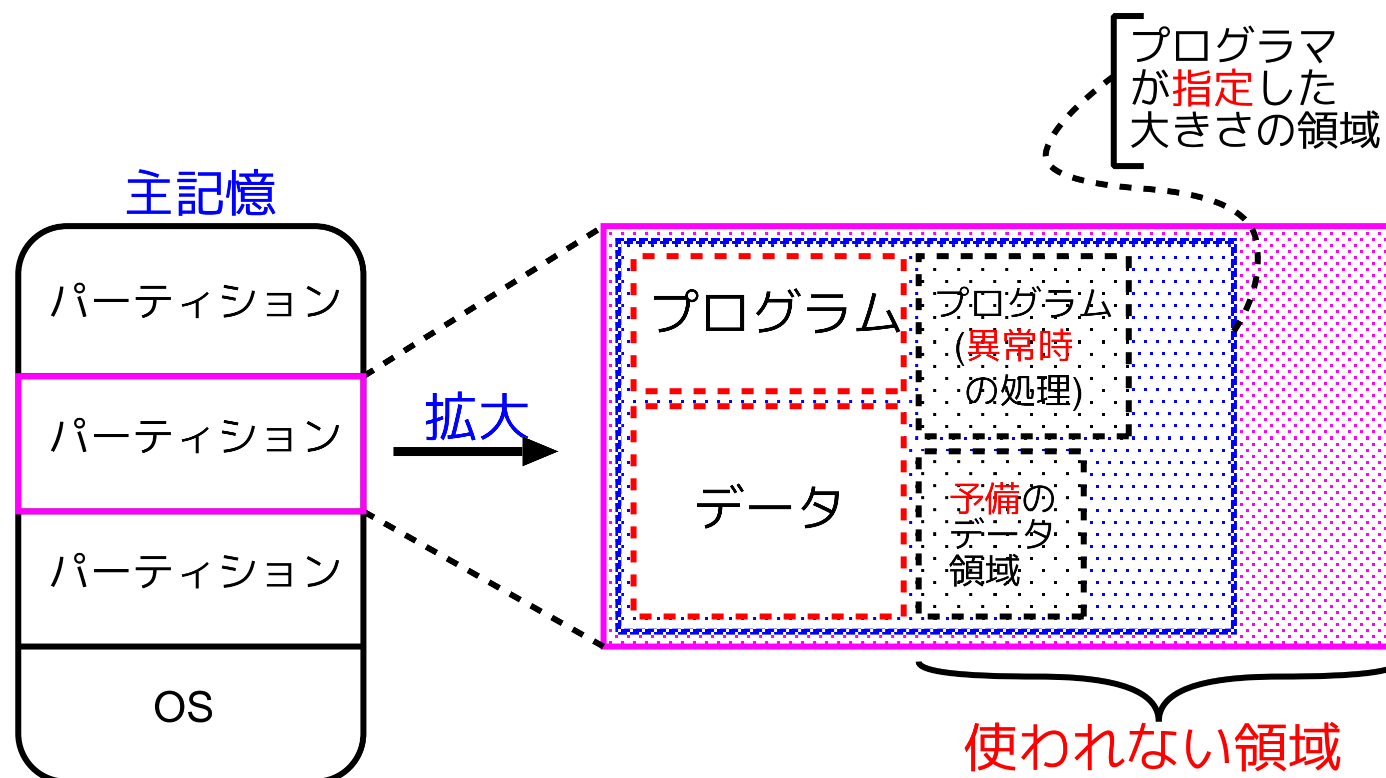
# 9–4 主記憶の中にジョブを詰めこむ際にどういう問題が発生するか

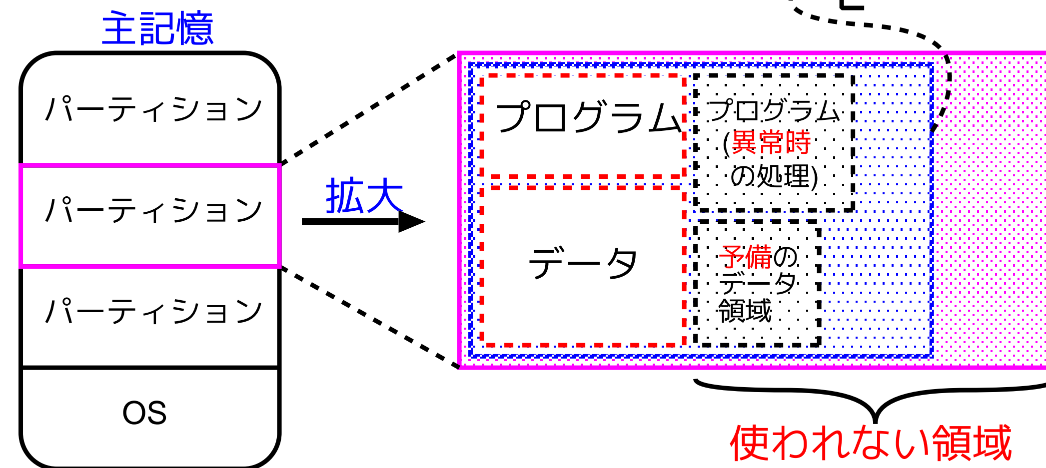
多重プログラミングを行うためには、...

⇨ 並行して走らせる複数のプログラムが主記憶を分け合って使う。

メモリ節約の問題：

主記憶を単純に固定的に分割し各々の小領域(パーティションと呼ぶ)に1つのプログラムを格納するというのでは、使われない領域が主記憶内の次の様な場所に出来てしまう。






- パーティション内に … パーティションの大きさを固定したために、…
- ジョブ空間内に … 昔はプログラム実行のために必要なメモリ容量をプログラマが指定していた。
- プログラム・データ領域内に … 異常処理のルーチンは通常は使われない。
  また、将来の機能拡張のためにデータ領域を大きめに確保しておくこともある。

⇨ これらの「使われない領域」が大きくならないように主記憶を分割しなければならない。

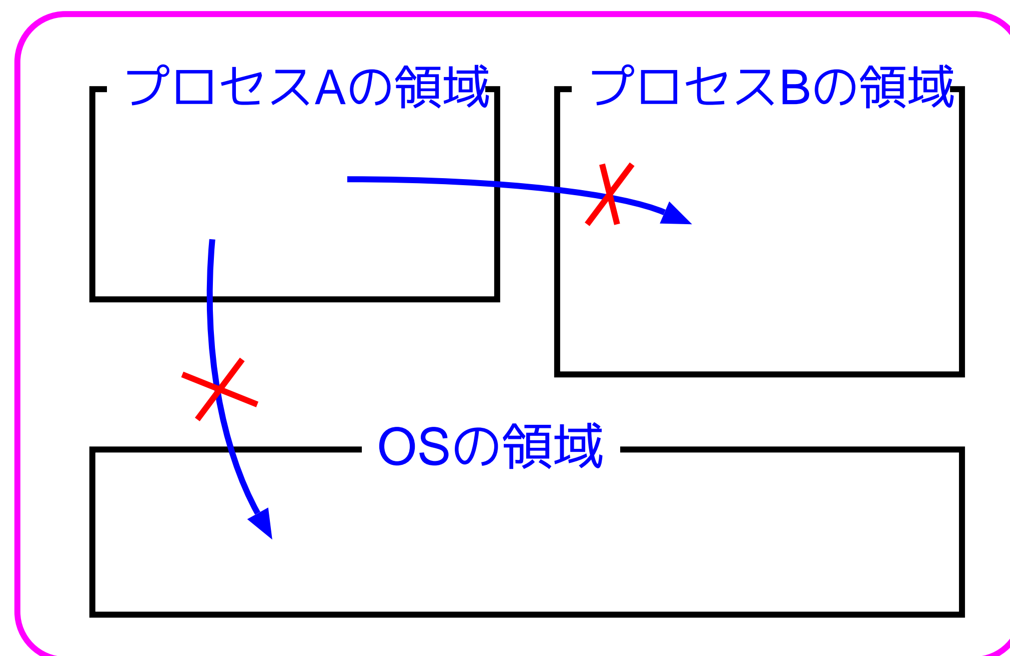
---

## 記憶保護の問題：

多重プログラミング環境においては複数のジョブが同時に主記憶領域を使うことになる。

⇨ ジョブの実行が他のジョブによって影響を受けるようなシステムは、信頼性の点で大きな問題がある。
更に、一般ジョブが**OS**の領域に勝手に書き込みを行ってしまうと、システムダウンにつながる。

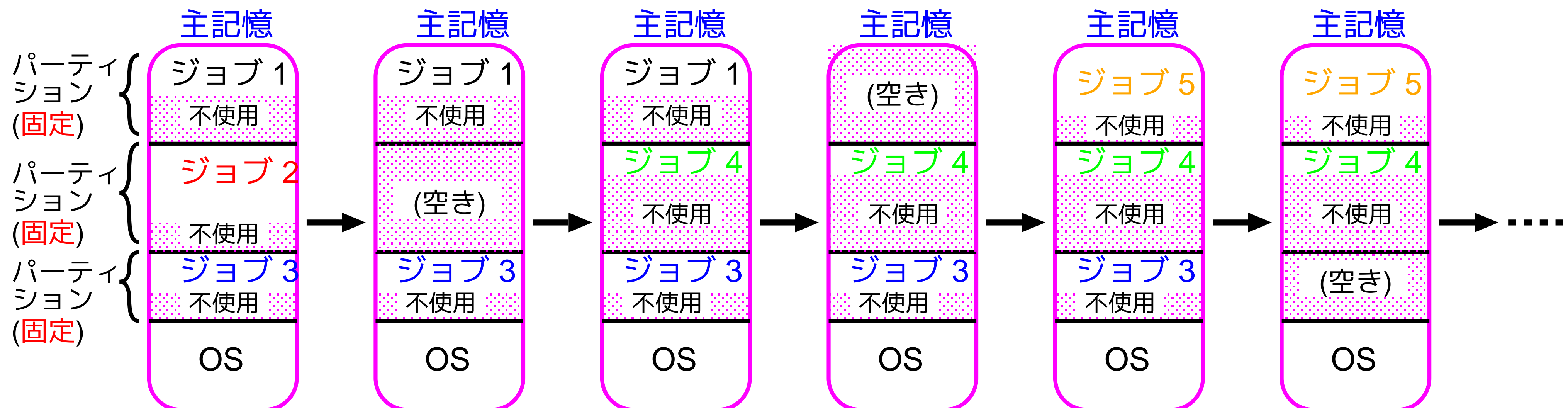
⇨ **1** つのジョブから別のジョブに割り当てられた主記憶領域への不当アクセスは絶対に避けなければならない。



# 9–5 主記憶を分割して多重プログラミングを実現する方法

静的メモリ分割 ・・・・・・・・・・・・ {各ジョブに主記憶の一部を割当てる最も簡単な方法

- システム起動直後に主記憶を固定的に分割
- 実行すべきジョブが現れたら十分なサイズのパーティション(の中で最小のもの) を適宜割り当てていく。

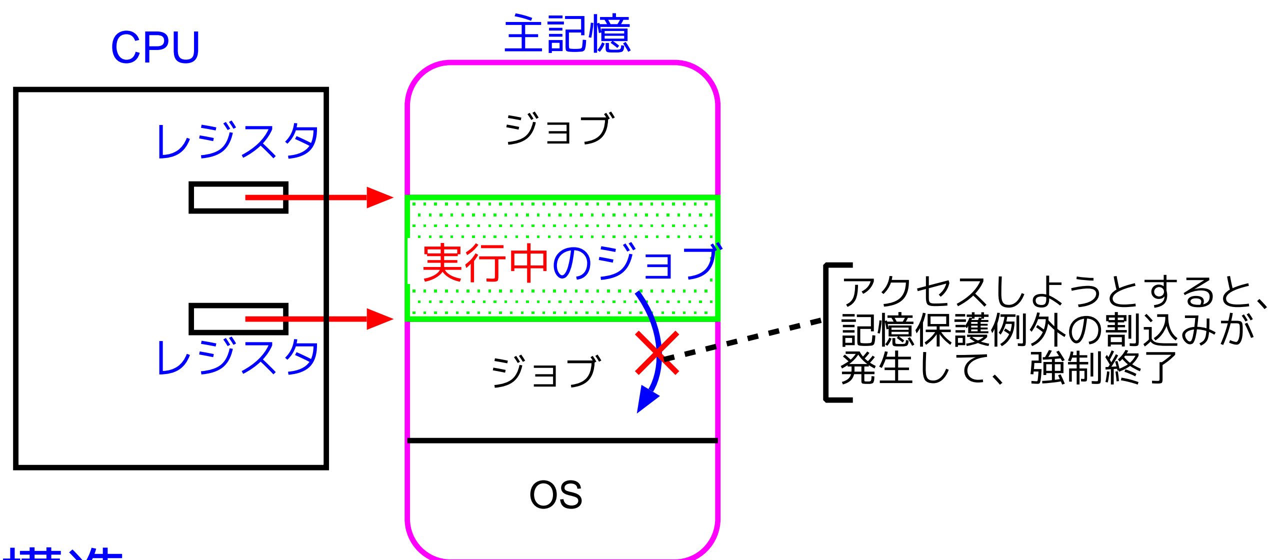


# 静的メモリ分割の利点・問題点：

- 利点(1) 容易に多重プログラミングが実現できる。

## 例えば、記憶保護のためには、

ジョブがCPU資源を割り当てられた時点で、そのジョブの使っている領域の境界を特別なレジスタ(メモリ保護境界レジスタと呼ぶ)に記憶する。これで、……




## データ構造：

どの領域が使われているかを掌握するためには、各々のパーティションが使用中かどうか(1か0)を記録したビットマップ表があれば良い。

- 問題点 (1) パーティション内に不使用領域 (他ジョブに割当て不可) ができ、メモリ節約の問題を解決できない。

  この様に、パーティション内に不使用な領域が出来ることを内部断片化 (内部フラグメンテーション) と呼ぶ。

- 問題点 (2) 大きなメモリを必要とするジョブの実行が困難になる。

補足： 連続する領域を同時に確保できれば実行できないこともないが、

◇ 同時に確保できる機会は少ない。

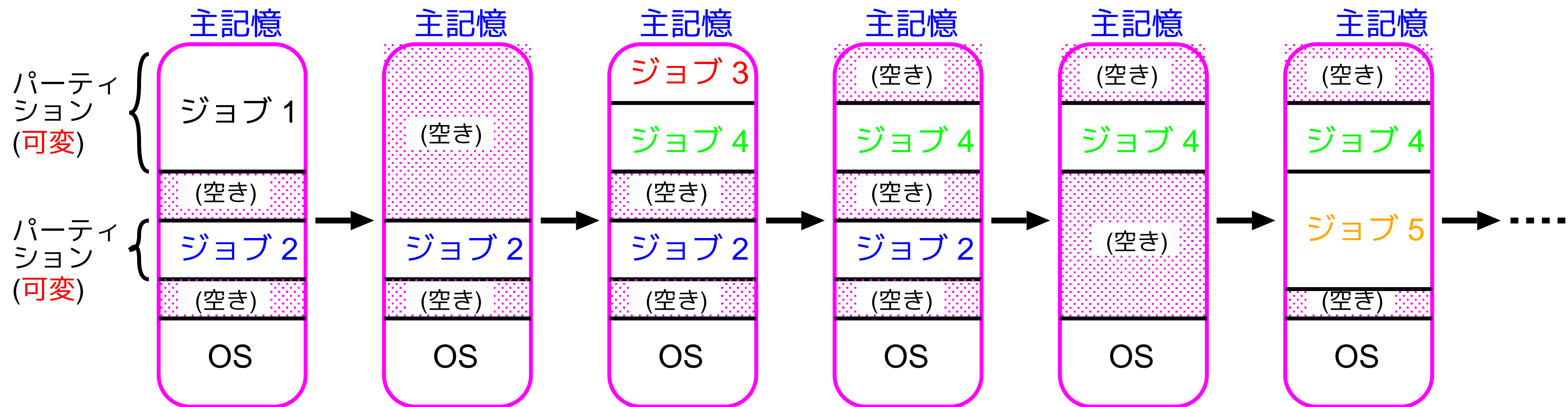
◇ 同時に確保しようとすると、使っているパーティションが空くのを待っている間、何にも使われないパーティションが出てくる。

◇ このようなパーティション再構成の作業は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。

# 動的メモリ分割：

各ジョブへの主記憶領域の割当てを次のように動的に行うこともできる。

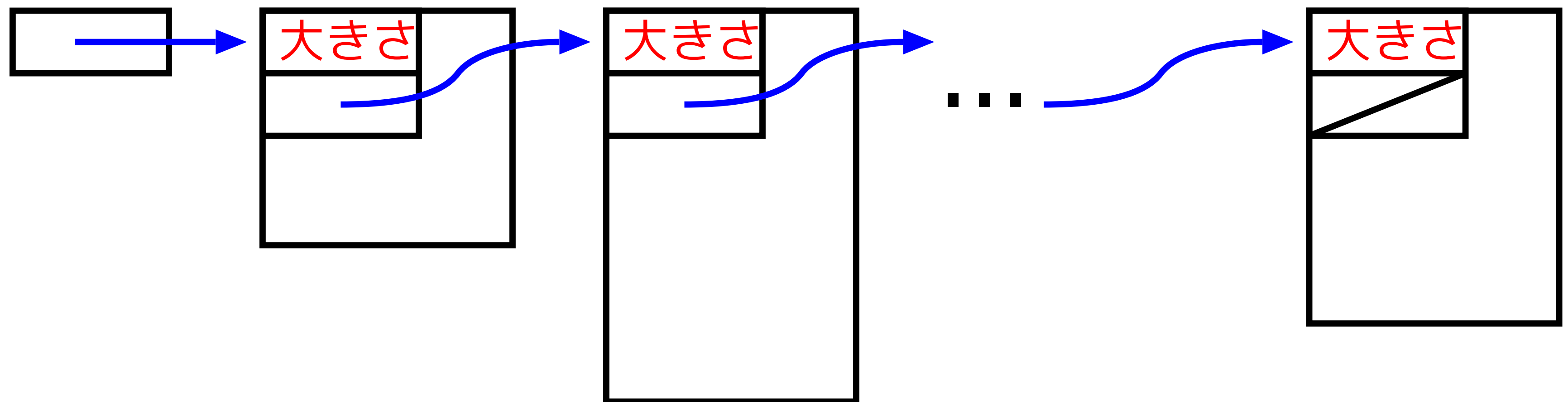
- 空き領域の所在を常に把握する。
- 必要な分だけメモリを空き領域から切り出して割り当てる。
- ジョブの実行が終了したら、使っていた領域は空き領域として戻す。その際、隣接する空き領域があれば即座に併合する。



補足：

1960年代後半の汎用大型コンピュータで採用

データ構造： —空き領域のリストを保持—



## 動的メモリ分割の利点・問題点：

- **利点 (1)** パーティション内 (ジョブ空間外) に不使用領域が発生することはない。
- **問題点 (1)** ジョブ空間内、プログラム・データ領域内に不使用領域ができる問題は解決できない。
- **問題点 (2)** ジョブに割り当てられるメモリの大きさが一定でないため、空き領域が細分化され小さな空き領域が多数でき、空き領域は全体では充分大きくても連続的でないのでジョブの割り当てには利用できないという現象が起こり得る。

  この現象を、外部断片化 (外部フラグメンテーション) あるいは単に断片化と呼ぶ。

# 9–6 仮想記憶の考え

着目点：

外部フラグメンテーションの原因は

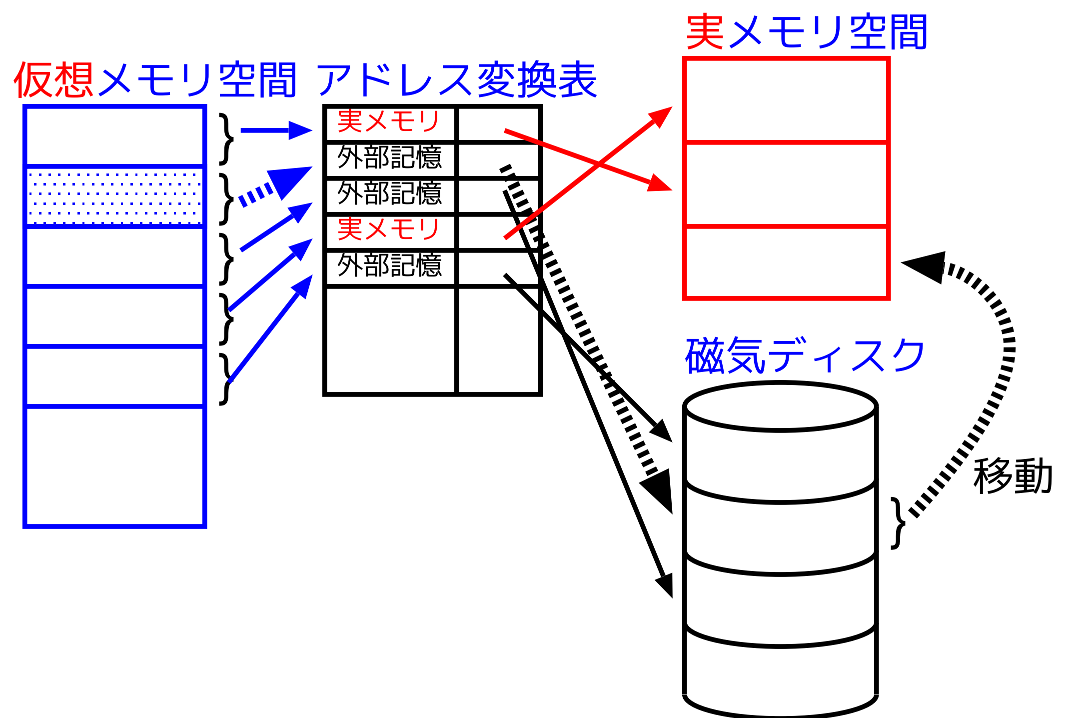
ジョブに割り付ける記憶領域は連続的でなければならない、

という制約にある。

⇨ この制約がなくなれば

外部フラグメンテーションは起こり得ない。

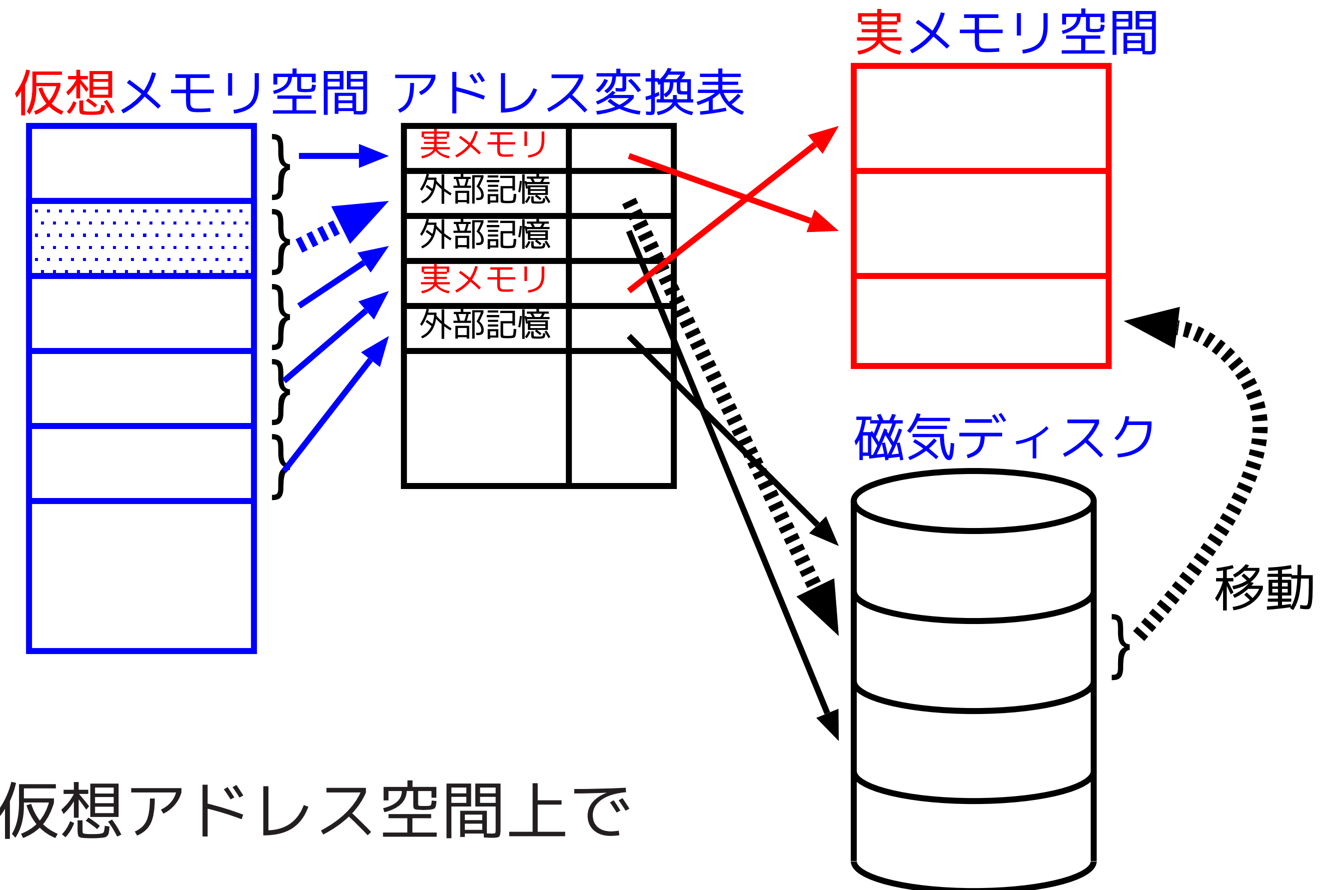
⇨ 仮想記憶の考え。



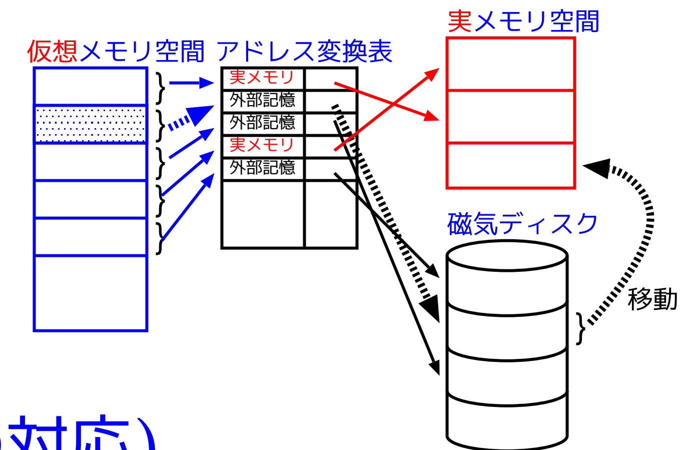

仮想記憶：、・・・(岩波情報科学辞典)
(機械語) 命令が生成するアドレスと実際に情報が存在する物理的な位置のアドレスを分離独立させる記憶方式。

# 仮想記憶の原理：

実記憶上のアドレス空間(実アドレス空間と呼ぶ)とは別の仮想的なアドレス空間(論理アドレス空間と呼ぶ)を考え、論理アドレス空間と実アドレス空間との間の対応付けを取る。




- プログラムは仮想アドレス空間上で動作させる。
  ⇨ アドレス空間は広大。
- 実メモリだけで足らない分は磁気ディスクなどを使ってまかなう。




## (仮想アドレス空間 $\Longleftrightarrow$ 実アドレス空間 の間の対応)

- 仮想アドレス空間は多数のブロックに分け、それらのブロック単位で仮想アドレス空間と実アドレス空間の間の対応をとる。
- 仮想アドレス空間上のブロックと実記憶上のブロックの対応付けを表(アドレス変換表と言う)の形で保持し、この表を用いて各時点での

  仮想アドレス $\Longrightarrow$ 実アドレス

  の変換を行なう。
- アドレス変換の結果、実アドレスが補助記憶上のブロック内にある場合は、そのブロックを実メモリ上に移す。

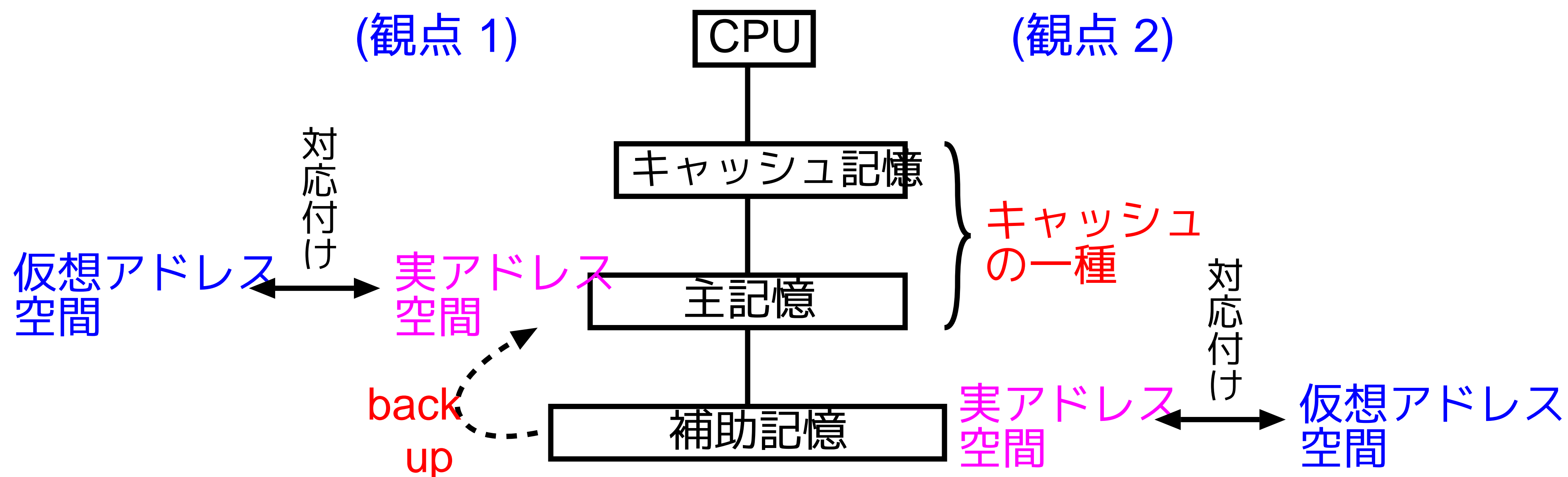
  その際、実メモリが満杯なら、実メモリ上の使わなくなったブロックを補助記憶に移す。

# 仮想記憶を支えるメモリ階層：

仮想記憶に対して、次の様な見方が出来る。

**(観点1)** 実アドレス空間は基本的には主記憶上に構成され、主記憶の容量不足を補うために後ろから支える記憶(バッキングストア)として補助記憶を用いる。

**(観点2)** 実アドレス空間は補助記憶上に構成され、主記憶をキャッシュ領域と見る。



## 仮想記憶の利点：

- 仮想アドレス空間を想定してプログラムを作るので、仮想アドレス空間をブロックにうまく分割できれば実メモリよりも大きなプログラムの動作が可能になる。
- 多重仮想記憶の場合、プログラムの動的再配置が容易。

補足：プログラムを実アドレス空間に直接入れ、単純に絶対的なアドレスを用いてプログラム内の変数の場所を表した場合、主記憶のどこにプログラムが配置されるかによって中の変数の場所が変わり、その場所を参照して作業するプログラム自身も変える必要が出てきてしまう。それゆえ、同じプログラムを違う場所に配置して実行することが出来なくなる。

⇨ この問題を解決するために次の様な方法

(方法 1) コンパイル後に出来るオブジェクトプログラムは、各命令内のアドレス部を決めない形式にし、……

(方法 2) プログラム内の大部分のアドレスはプログラムの基準点からの相対的な番地で表し、……

プログラムのアドレス空間を実記憶と独立させれば、この様な煩わしさから解放される。

# 仮想記憶の課題：

(課題1) 仮想アドレス空間をブロックに分割する仕方によっては、外部断片化や内部断片化が起こる可能性が残る。 ($\Longrightarrow$ 9.7~ 9.8節)

(課題2) 実メモリ$\Longleftrightarrow$補助記憶　の間でブロックの移動が頻繁に起こり、システム全体の性能の低下につながる可能性がある。

(スラッシングと言う。$\Longrightarrow$ 9.9節)

(課題3) 仮想アドレス$\Longrightarrow$実アドレス　のアドレス変換を常時行なわなければならないので、処理が遅くなる。

($\Longrightarrow$ 9.7節「DAT」, 9.10節)

(課題4) アドレス変換表が巨大になる可能性がある。 ($\Longrightarrow$ 9.11節)

# 9–7 ページングによる仮想記憶

ページング：

ページングにおいては、仮想アドレス空間も主記憶領域も大きさが2の冪乗byteの同じ大きさのブロックに均等に分割する。

- 仮想アドレス空間内のブロックをページと呼ぶ。
- 主記憶内のブロックをページフレームと呼ぶ。
- 仮想アドレス空間上のブロックと実記憶上のブロックの対応付けのためのアドレス変換表を、特にページ表と呼ぶ。
- 仮想アドレス空間上のブロックに実メモリ (ページフレーム) を割り当てる操作をページインと言い、逆に (必要があれば内容を補助記憶に移した上で) 割り当てられた実メモリを解放する操作をページアウトと言う。

補足：

ページングを採用した場合、実行すべきジョブへの主記憶の割り当ては次のように行われる。

(1) 必要なページ数を計算する。

(2) 空き状態にあるページフレーム数を確認する。

(3) 空きページフレーム数 > 必要なページ数 なら、

(3.1) このジョブ用にアドレス変換の表を用意する。
(エントリ数＝必要なページ数)

(3.2) アドレス変換の表に空き状態のページフレームの番号を書き込む。

---

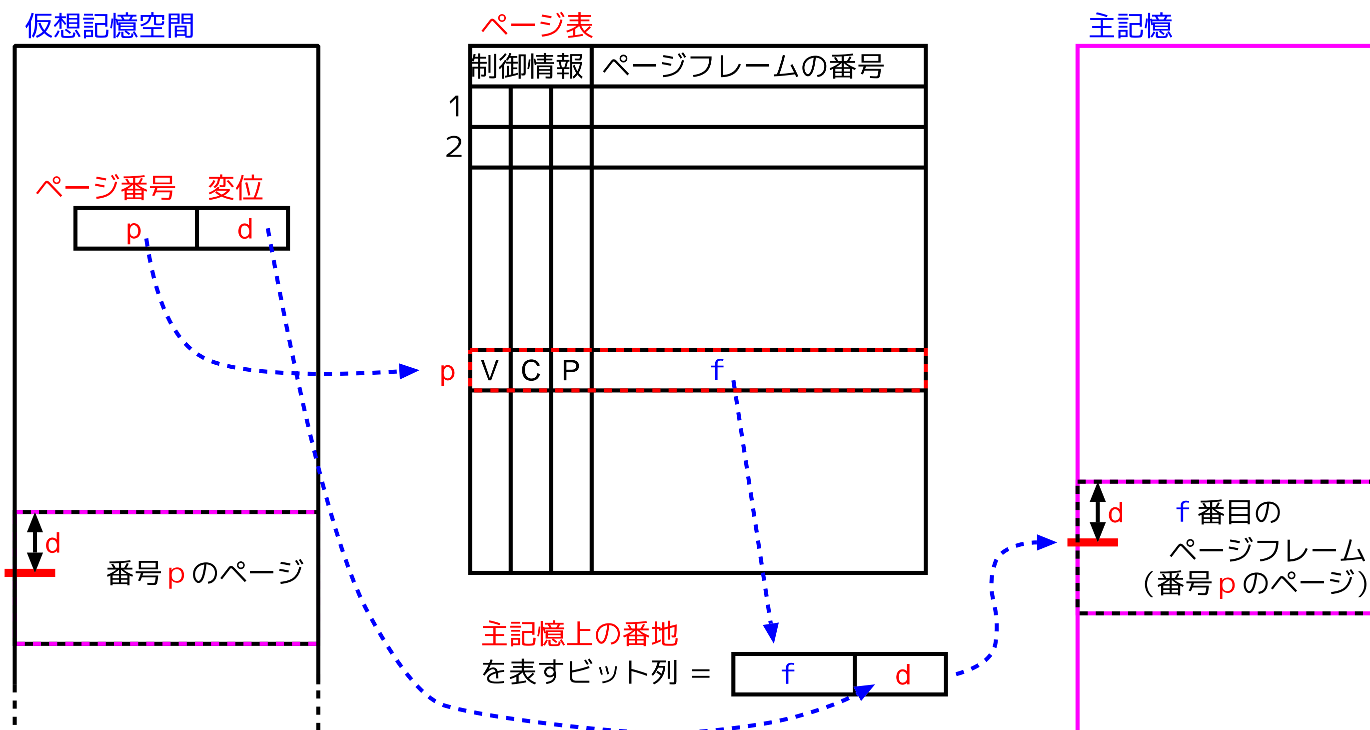
ページングの考え方自体は比較的古い。
実際、岩波情報科学辞典によれば、**1962**年**ATLAS**計算機に初めてページングが導入され、その後**1965**年頃に**MIT** の**Multics**システムに本格的に採用された。

## ページングにおけるアドレス変換機構：

仮想アドレスから実アドレスに変換する機構を一般に**動的アドレス変換機構**(**DAT**)と呼ぶ。

- **DAT**は**CPU内の機構**で、仮想アドレス空間にアクセス(**e.g.**命令読み込み)する際には必ず呼び出される。
- **DAT**は**OS**のメモリ管理部が作ったアドレス変換表を使って、入力された仮想アドレスを実アドレスに変換する。
- **ページングの場合**は、具体的には、**DATの変換は次のように**行われる。

2ページ後に拡大図



- ページングの場合は、
  具体的には、DATの変換は次のように行われる。

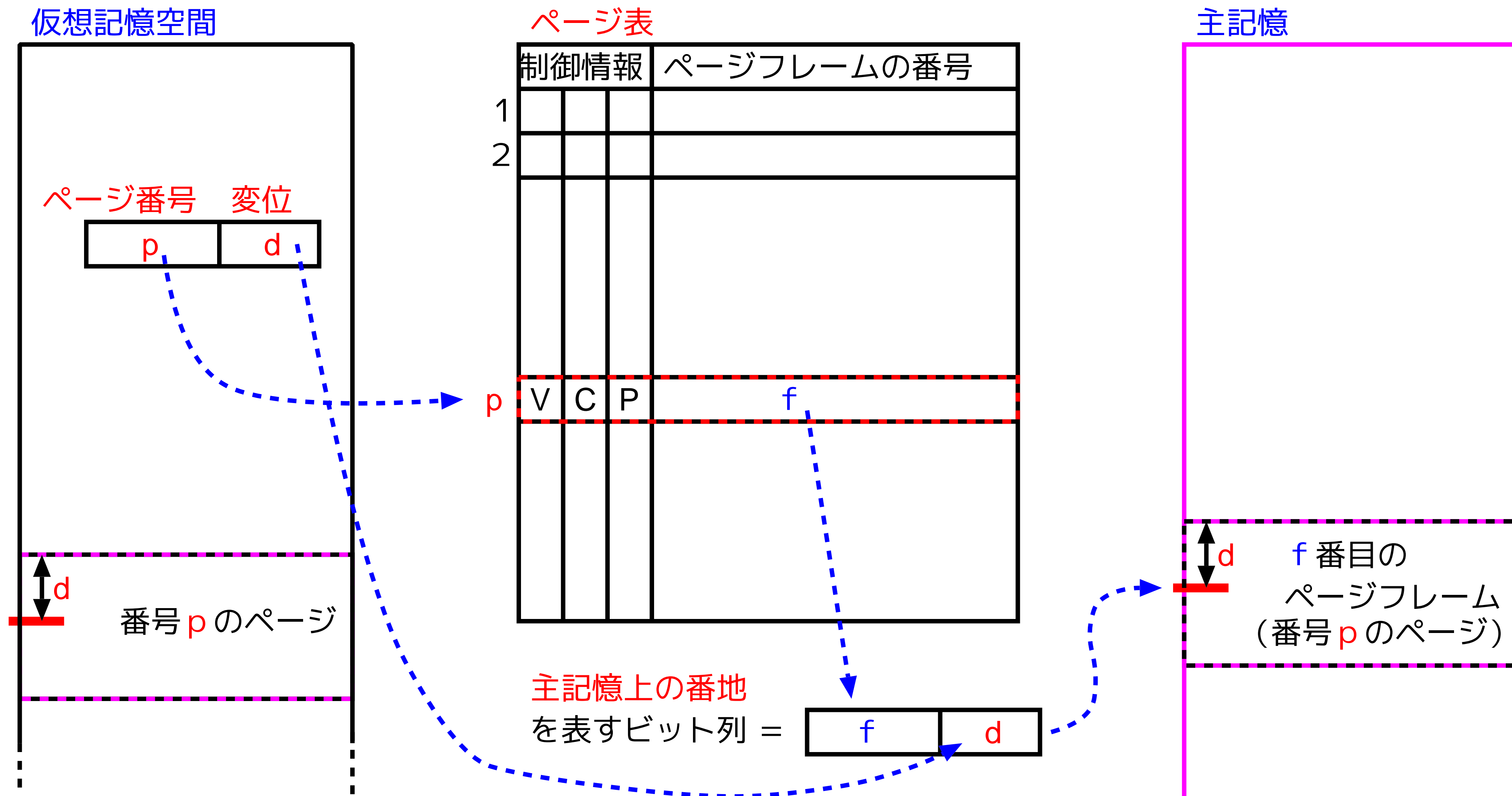
  ◇ 仮想アドレスを上位ビット列と下位ビット列の2つに分け、

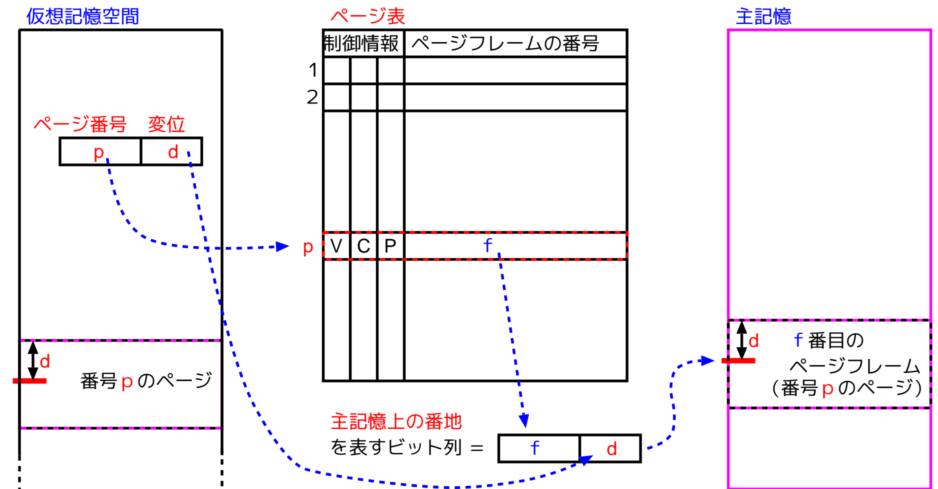
  $\begin{cases} \text{上位ビット列がページ番号を、} \\ \text{下位ビット列がページの先頭からの変位を} \end{cases}$

  表しているものと考える。
  例えば、下位ビット列が 12 bit なら1ページが$2^{12}$byte $\approx$ 4KB ということになる。

  ◇ アドレス変換表には、どの番号のページにはどの番号のページフレームが割り当てられているかの対応情報を記録する。

  ◇ アドレス変換はアドレス変換表を基に、仮想アドレスの上位ビット列を対応するページフレームの番号に置き換えるだけである。


仮想記憶空間
ページ表
主記憶
制御情報
ページフレームの番号
1
2
ページ番号
変位
p
d
p
V
C
P
f
d
番号 p のページ
f 番目の
ページフレーム
（番号 p のページ）
d
主記憶上の番地
を表すビット列 =
f
d




$$V = \begin{cases} 1 & \text{if ページが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}$$

$$C = \begin{cases} 1 & \text{if ページが修正された} \\ 0 & \text{otherwise} \end{cases}$$

$$P = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ 010 & \text{if write only} \\ 011 & \text{if read/write} \\ 100 & \text{if execute only} \\ 101 & \text{if read/execute} \\ 110 & \text{if write/execute} \\ 111 & \text{if read/write/execute} \end{cases}$$

## ページングにおける利点：

**9.6** 節の「仮想記憶の利点」**(p.101)** の所で列挙したものの他に、次の利点が生じる。

- 外部断片化が起きない。
- 主記憶容量よりも大きなプロセスを実行できる。

## ページングにおける課題：

9.6節の「仮想記憶の課題」(p.101)の所で列挙したものの中で、ページングにおけるブロック分割の仕方によって(課題1)が改善され、全体として次の様な課題が残る。

(課題1) 仮想アドレス空間内のページを実メモリに入れる仕方によっては、内部断片化が起こる可能性が残る。 (⟹ 9.8節)

(課題2) 実メモリ ⟺ 補助記憶　の間でブロックの移動が頻繁に起こり、システム全体の性能の低下につながる可能性がある。
(スラッシングと言う。⟹ 9.9節)

(課題3) 仮想アドレス ⟹ 実アドレス　のアドレス変換を常時行なわなければならないので、処理が遅くなる。 (⟹ 9.10節)

> 補足：　アドレス変換表のエントリをCPU内の高速記憶にキャッシュする、**TLB** と呼ばれるハードウェア機構も導入され、処理の遅れはかなり改善されている。

(課題4) アドレス変換表が巨大になる可能性がある。 (⟹ 9.11節)

# 9–8 デマンドページング

一口に「ページング」と言っても、次の点に関していくつかの選択肢がある。

- (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

この内フェッチの方式に関しては、通常、

プロセス起動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む、

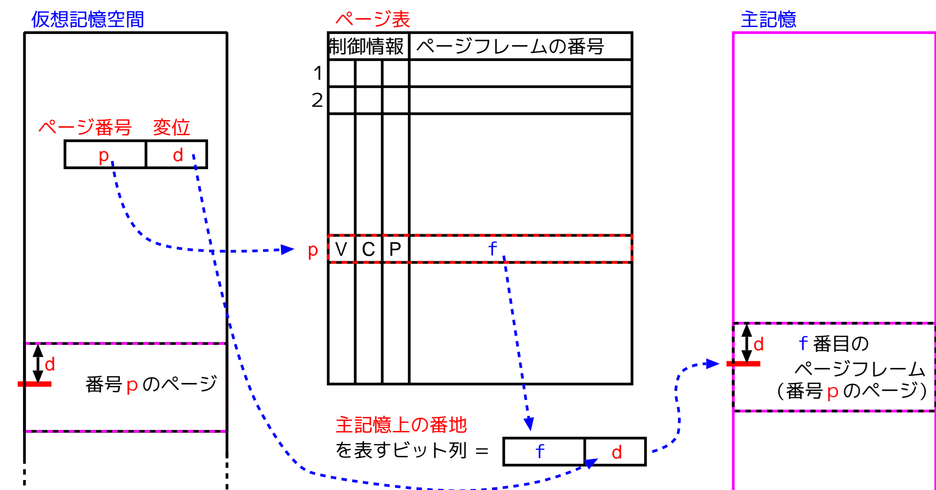
という方法がとられている。この種のページングを特にデマンドページング, オンデマンドページング, または要求時ページングと言う。

期待できる効果：
ジョブ空間内の不使用領域(e.g.予備,異常処理ルーチンの領域)のために実際に主記憶を割り付けることがなくなり、実メモリの有効利用が可能になる。

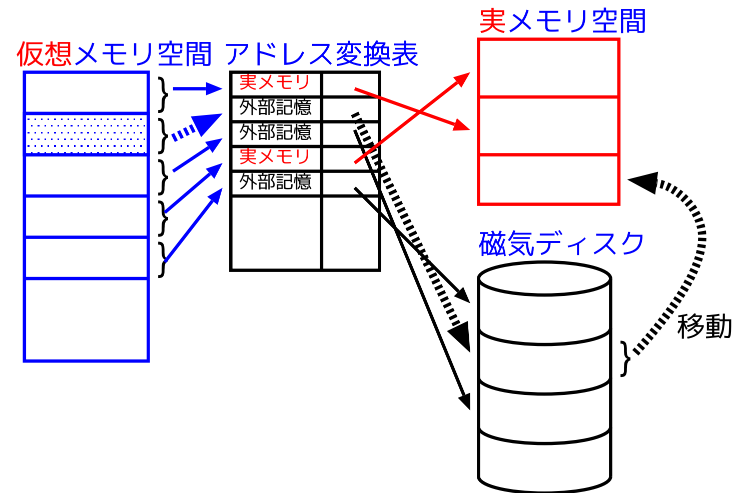
# デマンドページングの詳細：

- アドレス変換表に
  - ◇ ページに対して実メモリが割り当てられているかどうか,
  - ◇ ページが修正されたかどうか,
  - ◇ ページが読込み可能かどうか,
  - ◇ ページが書込み可能かどうか,
  - ◇ ページが実行可能かどうか

  を記録するフラッグの欄を設ける。




- アクセスされたページに実メモリが割り当てられていなかった場合は、DAT がページ・フォールト割り込みを発生させるハードウェアを用意する。

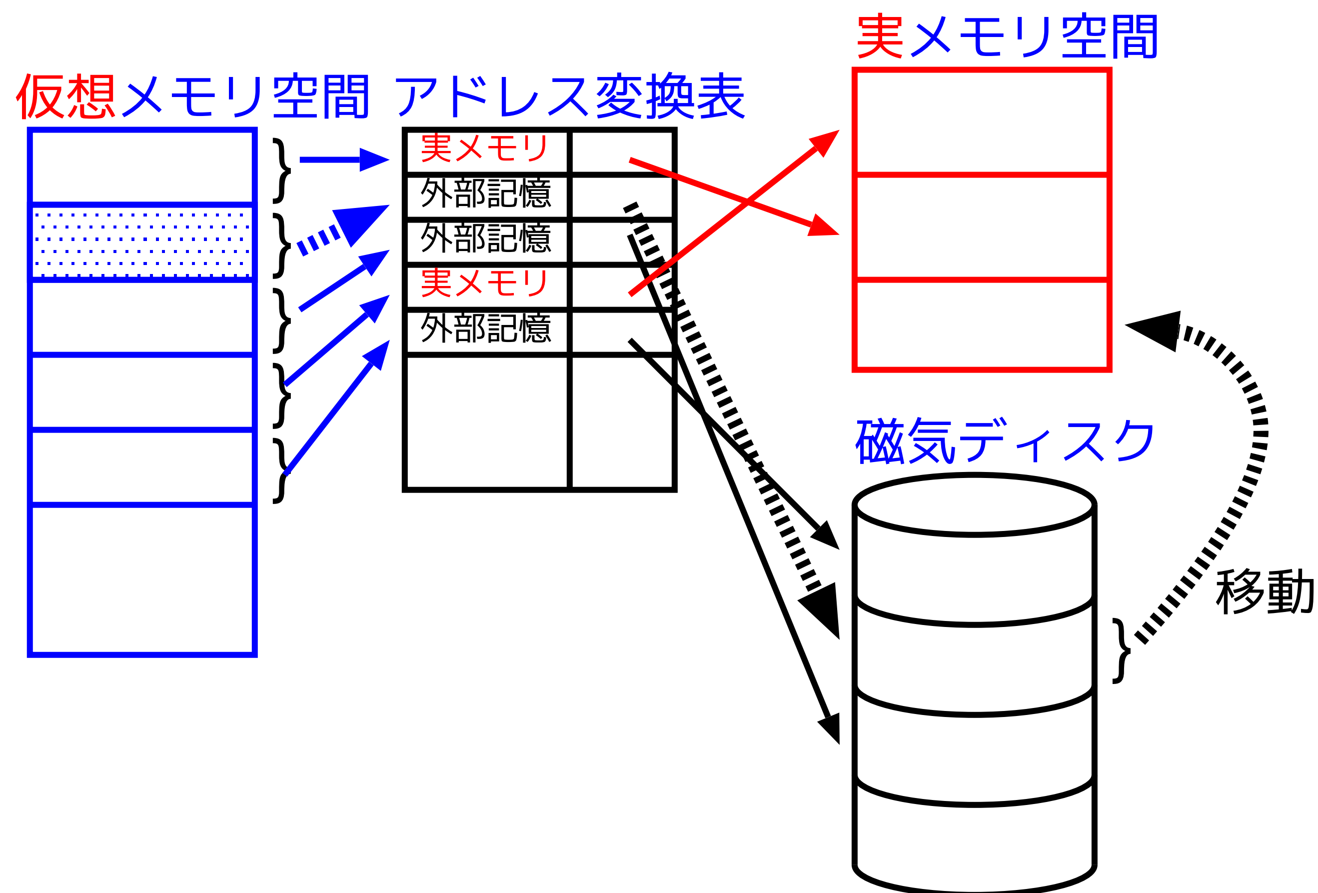



- プロセス発生時は、アドレス変換換表だけを用意してページに対して実メモリを割り当てることはしない。

**プロセス生成時に実際にOSが行うこと：**

(1) 必要なページ数を計算する。
(2) 空き状態にあるページフレーム数を確認する。
(3) このジョブ用にアドレス変換の表を用意する。
　(エントリ数＝必要なページ数)
(4) アドレス変換の表の全てのページに関して
　◇ ページに対して実メモリが割り当てられているかどうか＝**No**,
　◇ ページが修正されたかどうか＝**No**,
　◇ ページが読込み可能かどうか＝...,
　◇ ページが書込み可能かどうか＝...,
　◇ ページが実行可能かどうか＝...
　と記録する。(実メモリは割り当てない。)

- ページ・フォールト割り込みが発生すると、
  OSはページ・フォールトの起こったページに対して実メモリを割り当てる。この操作をページインと言う。



## ページ・フォールト割り込み発生時に実際にOSが行うこと：

(1) 不使用のページフレームを探し出して、その番号をアドレス変換表に書き込む。

(2) (主記憶に入れる)ページがテキスト部の場合、 ←プログラム等

ロードモジュールが格納された補助記憶からページ内容をロードする。

ページがデータ部の場合、

最初のページイン時にはロードモジュールが格納された補助記憶から、
2度目以降のページイン時には「ページアウト」された補助記憶から
ページ内容を読み込む。

ページがスタック部の場合、

最初のページイン時にはページ内容の読み込みは行わず、
2度目以降のページイン時には「ページアウト」された補助記憶から
ページ内容を読み込む。

(3) アドレス変換表において、

ページに対して実メモリが割り当てられているかどうか=**Yes**

とする。

- ページインの操作時に
  不使用のページフレームが無い ことが判明したら、
  近い将来参照がないと思われる実ページ を主記憶全体から探し出し、
  (場合によっては) 一旦補助記憶にその実ページの内容を退避した上で、
  解放する。 この操作をページアウトと言う。

ページアウトのために実際にOSが行うこと：

**(1)** 追い出すページが修正されたかどうかのフラッグが立っていなければ、
ページフレームの退避は行わない。
立っていれば、
**(1.1)** 補助記憶上にページアウトのための領域がなければ確保する。
**(1.2)** 補助記憶上にページフレームの内容を退避する。
(この退避によって出来たファイルをページングファイルと呼ぶ。)
**(1.3)** 補助記憶上のページアウト先のアドレスをアドレス変換表に書き込む。
**(1.4)** アドレス変換表において、
ページが修正されたかどうか=**No** とする。
**(2)** ページフレームを解放する。(空きページフレームとして登録する?)
**(3)** アドレス変換表において、
ページに対して実メモリが割り当てられているかどうか=**No**
とする。

## デマンドページング方式の利点：

- 参照されたページに対してだけ実メモリが割り付けられるので、必要最低限の実メモリ消費となり、メモリ節約の問題をほぼ完全に解決できる。

実際、

◇ ジョブの使う空間外だけのページに対しては、主記憶も補助記憶も割り当てられない。

◇ ジョブ空間内の予備データ領域,異常処理ルーチン領域, ...だけから成るページは本当に使われる時点まで実メモリが割り当てられない。

ただ、

各々のページの中には色々な領域が混在しているので、実メモリが割り当てられたページの中に不使用な領域が含まれることもある。それゆえ、内部断片化は多少起こる。

# 9–9 スラッシングとその対策

スラッシング： . . . (p.101「仮想記憶の課題」の所で触れた課題2)
仮想記憶においては 実メモリ $\Longleftrightarrow$ 補助記憶 の間のブロックの移動をOSが断続的に行なうことになるので、その分だけOSの処理によるオーバヘッドは大きくなる。

overhead n. [商][米]間接費

そして、
必要な仮想記憶の全容量が(実際の主記憶の容量より) 大きくなり過ぎる場合には、

実メモリ $\Longleftrightarrow$ 補助記憶

の間でブロックの移動が頻繁に起こり、システム全体が極端な性能低下に陥ってしまう。 この現象を一般にスラッシングと呼ぶ。
例えば、ページングの場合 には次の様な現象を指す。

- ページフォールト割り込みが多発し、
- これらの割り込み処理(ブロックの移動)にほとんどのCPU資源,入出力資源を消費してしまう。

## スラッシングに対する対策：

スラッシングを防ぐための対策としては、

| 実メモリ$\Longleftrightarrow$補助記憶　の間のブロック移動の頻度を監視し、
| その結果を基に多重プログラミングの多重度を調節する、

という簡単な方法がとられることが多い。

特にページングの場合には、多重度の調節は次の様に行なわれる。

- ページフォールトの起きる頻度を常に監視する。
- ページフォールトが頻繁に起こり過ぎる場合、
  優先度の低いプロセス(に割り当てられた実メモリ)をに完全に主記憶から補助記憶に退避する。この退避操作をスワップアウトと言う。
- ページフォールトの頻度があるレベルまで落ちた場合、
  スワップアウトされたプロセスがあれば、その中の1つを実記憶に戻し実行を再開する。この回復操作をスワップインと言う。

**p.83からの引用：**

**(swapper)** ···

**BSD**系のシステムにおけるデーモンで、**PID = 0** である。

.....................(中略).....................

**swapper**はページフォールトの起きる頻度を常に監視し、ページフォールトが頻繁に起こり過ぎる場合、スワップ領域に追い出してしまうプロセスを選び出す。

これで選び出されたプロセスは完全に主記憶から追い出され(スワップアウトと言う)、比較的長い時間実行されない。

そして、ページフォールトの頻度があるレベルまで落ちた段階で、スワップアウトされたプロセスが実記憶の中に戻され(スワップインと言う)、その実行が再開される。

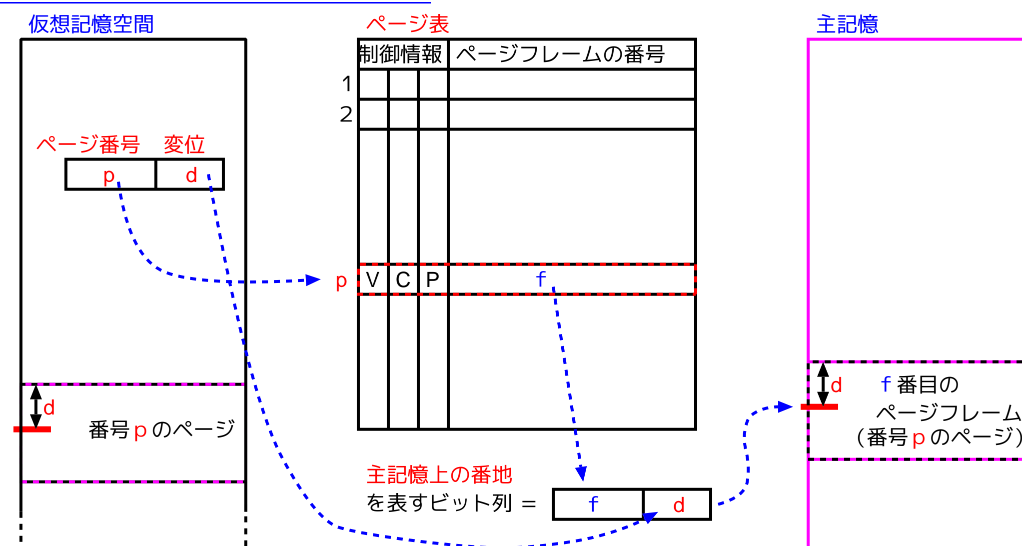
# 9–10 仮想記憶を支えるハードウェア

実用的な仮想記憶を実現するために、次の様な機構がハードウエアの一部として備わっていることが多い。

- **アドレス変換機構 (DAT)**
  ··· アドレス変換表を使って論理アドレスを実アドレスに変換するCPU内の機構。 論理アドレス空間にアクセスする際は必ず呼び出される。

**p.103からの引用：**

p.103からの引用：




$$V = \begin{cases} 1 & \text{if ページが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}$$

$$C = \begin{cases} 1 & \text{if ページが修正された} \\ 0 & \text{otherwise} \end{cases}$$

$$P = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ & \cdots\cdots \end{cases}$$

- **TLB**(アドレス変換バッファとも言う)

  ··· アドレス変換表のエントリを**CPU**内の高速記憶にキャッシュして利用する機構。

  仮想アドレスから実アドレスへの変換要求があると、アドレス変換表による変換処理と**TLB**による変換処理が並行して進み、**TLB**側で成功すればアドレス変換表による変換は中止される。

- ページの使用状況を記録する機構

  ··· ページ参照フラグ，ページ更新フラグ，等をページ表の中に用意し、ハードウェア的に更新していく。

# 9–11 ページ表が大きくなり過ぎることをどう克服するか

論理アドレス空間が大きい場合、単純に1個のアドレス変換表(ページ表)を用意するのではページ表が大きくなり過ぎてしまう。

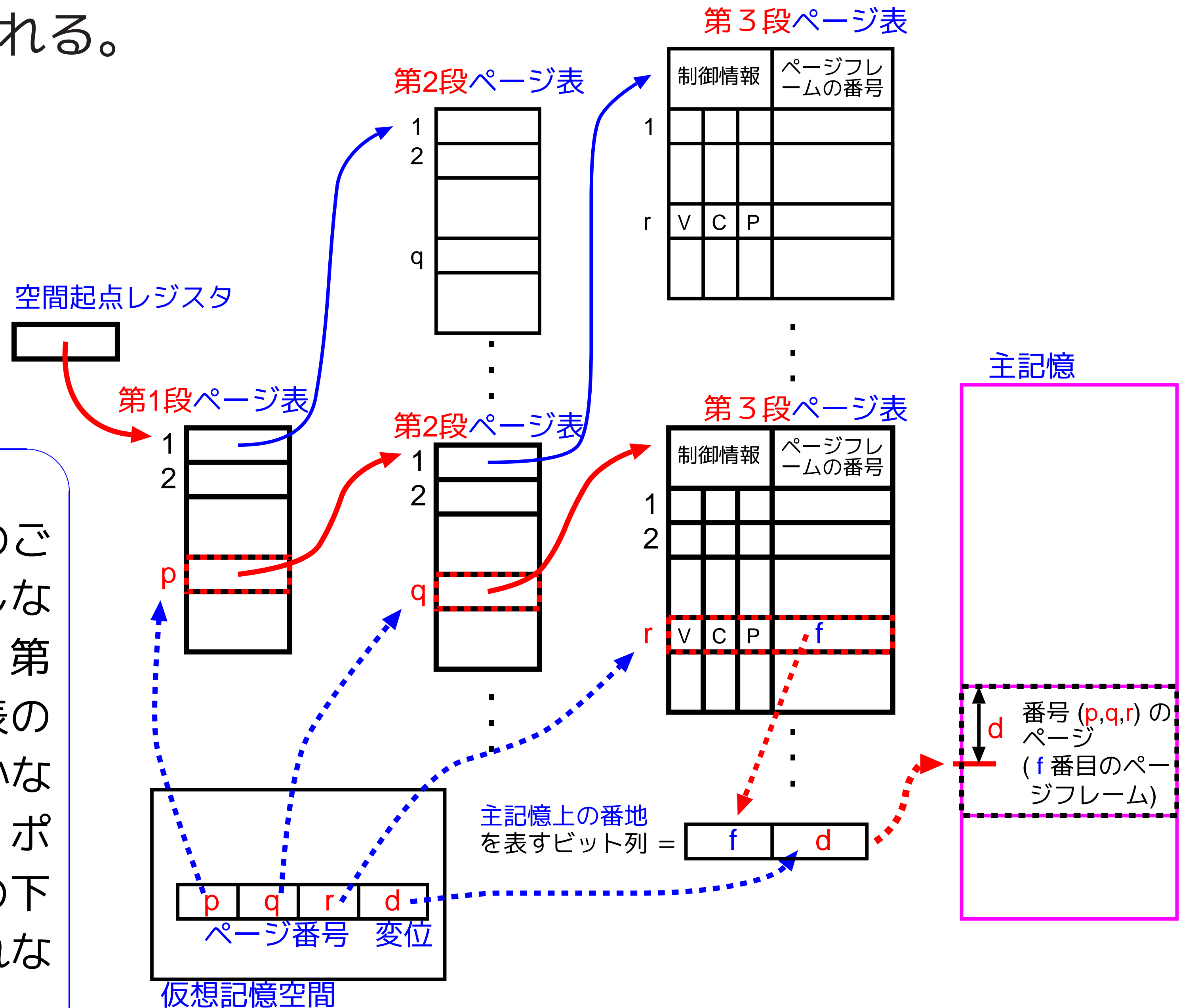
例えば、
32 bit のアドレス空間でページサイズが $2^{12} = 4$ K byte の場合、ページ表を1個用意する方法ではページ表のエントリ数は $2^{20} \approx 10^6$ になってしまう。

⇨ ほとんどのプロセスは論理アドレス空間のごく一部分しか使用しないので、メモリを節約するために通常は次の様な方法がとられている。

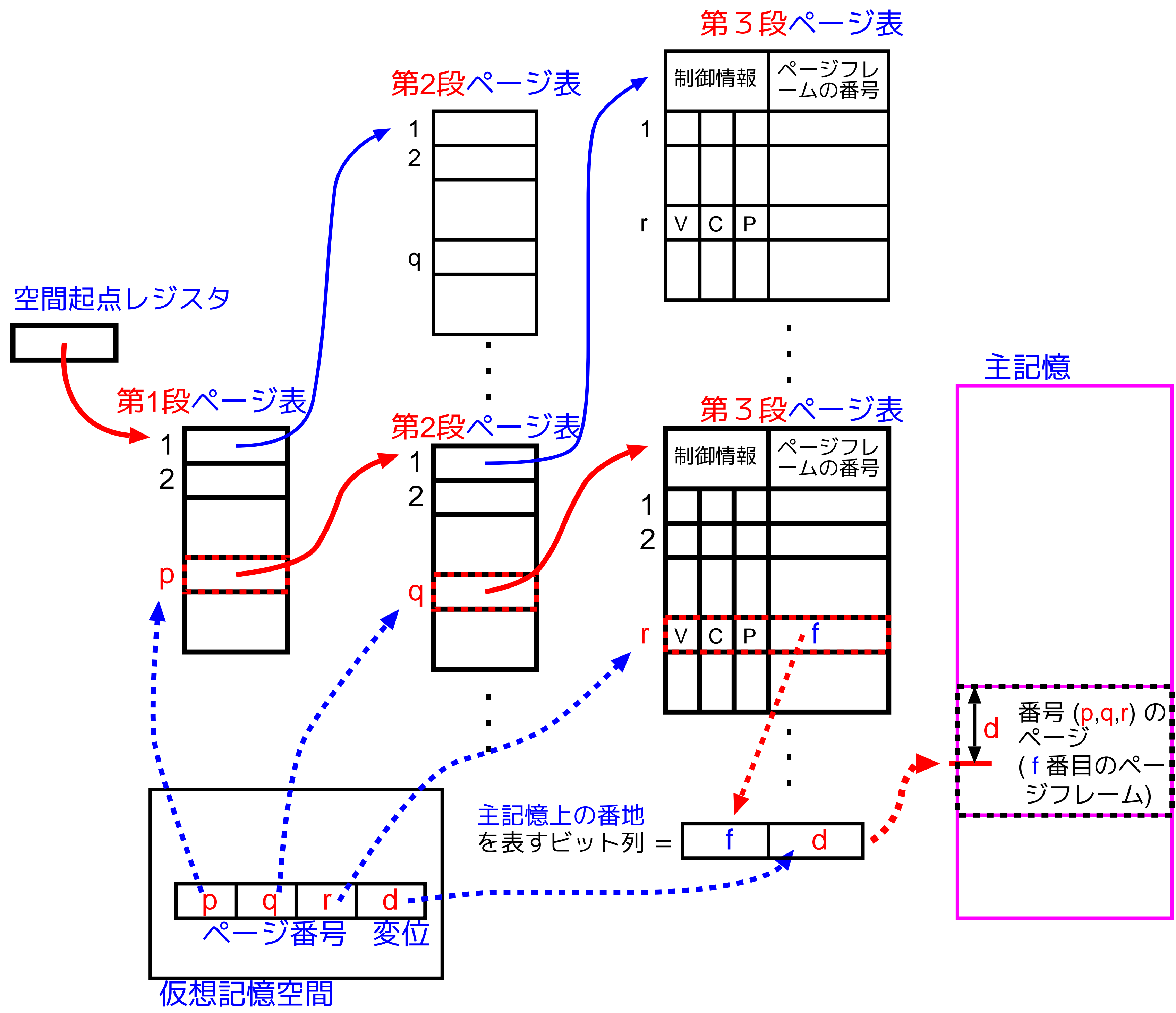
- (方法1) ページ表を多段に構成
- (方法2) 逆ページ表を構成しハッシングで表検索

# (方法1) ページ表を多段に構成する。

例えば、Sun Microsystems社のSPARCにおいては、3段のページ表が構成される。




**補足：**
論理アドレス空間のごく一部分しか使用しないプロセスの場合、第1段, 第2段ページ表のエントリの中にはかなりの個数の NULL ポインタが並び、その下のページ表は作られない。

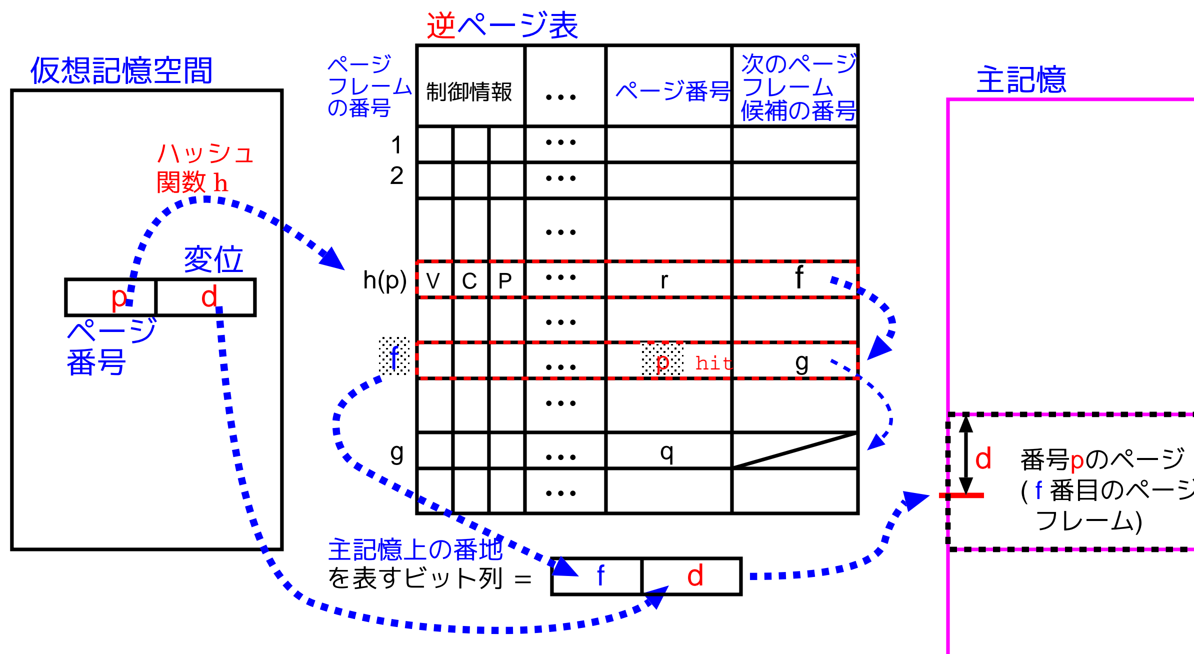


第３段ページ表
制御情報
ページフレームの番号
第2段ページ表
1
2
q
r
V
C
P
空間起点レジスタ
第1段ページ表
1
2
p
第2段ページ表
1
2
q
第３段ページ表
制御情報
ページフレームの番号
1
2
r
V
C
P
f
主記憶
d
番号 (p,q,r) のページ
( f 番目のページフレーム)
主記憶上の番地
を表すビット列 ＝
f
d
p
q
r
d
ページ番号
変位
仮想記憶空間

## (方法2) 逆ページ表を構成しハッシング手法で表検索。

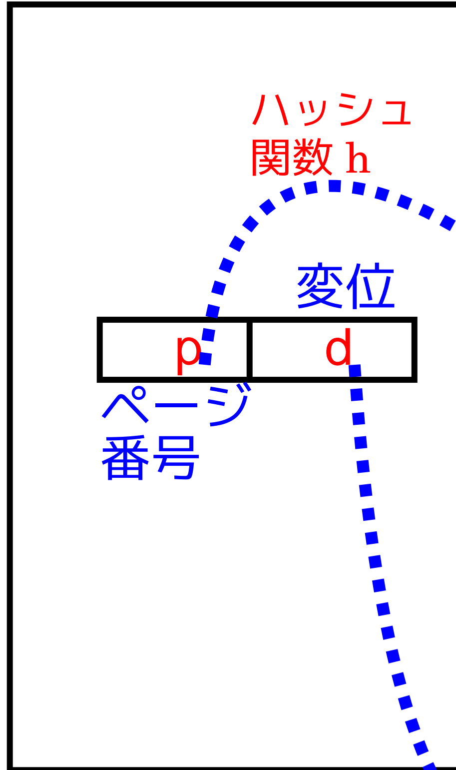
仮想空間内のページ数より主記憶内のページフレーム数の方がかなり小さい。

⇨ 主記憶内の各々のページフレームについて、占有ページの情報を記録した表(逆ページ表と言う;全プロセスに共通)を構成し、ハッシュ関数を使ってこの逆ページ表を検索する。




**補足：**
指定されたページが主記憶上にない場合もあるので、ページ表はやはりプロセス毎に(多分ディスク内に)用意しなければならない。

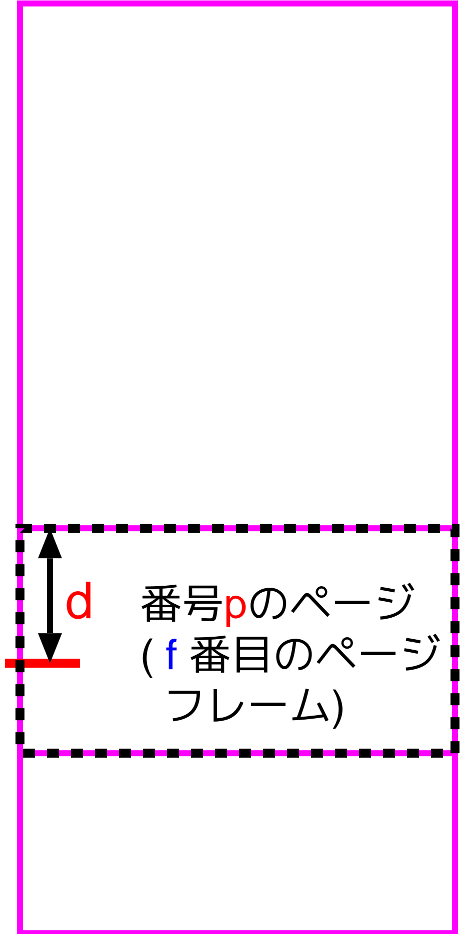
⇨ ページフォールトがあるとページインの前にページ表を読み込む必要がある。




逆ページ表

| ページフレームの番号 | 制御情報 | | | … | ページ番号 | 次のページフレーム候補の番号 |
|---|---|---|---|---|---|---|
| 1 | | | | … | | |
| 2 | | | | … | | |
| | | | | … | | |
| h(p) | V | C | P | … | r | f |
| | | | | … | | |
| f | | | | … | p hit | g |
| | | | | … | | |
| g | | | | … | q | |
| | | | | … | | |

主記憶上の番地を表すビット列 = | f | d |



# 9–12 ページングvs.セグメンテーション
## —仮想記憶を実現する2つの方法—

仮想記憶を実現する方法として基本的には次の2つがある。

- ページング
- セグメンテーション

これらは仮想記憶空間の分割の仕方に主な違いがある。

## ページング(9.7~ 9.11節のまとめ)：

仮想記憶空間を固定長のブロック(ページと呼ぶ) に分割して、これらの単位毎に実メモリとの対応付けを取る。

- 仮想記憶空間のアドレスを上位ビット列と下位ビット列の2 つに分け、
  - 上位ビット列がページ番号を表し、
  - 下位ビット列がページの先頭からの変位を表し

  ているものと考える。

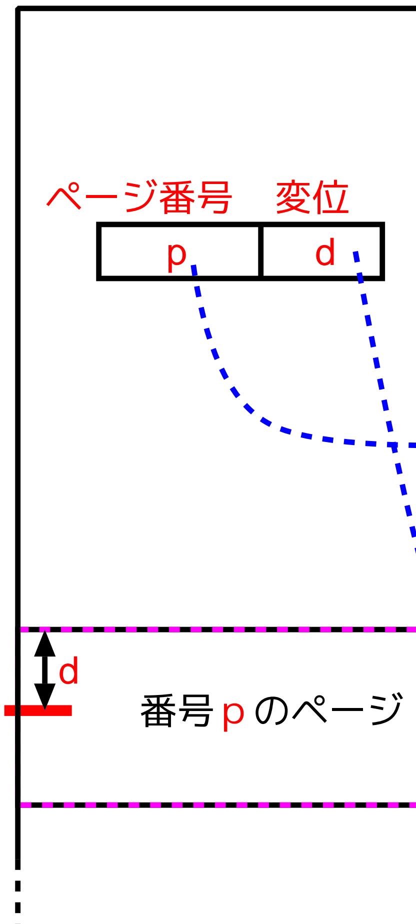
- 主記憶の実空間は1ページに相当する大きさの小領域(ページフレームと言う)に均等に分割し、仮想記憶空間上のページと対応付ける。

- 与えられた番号のページが

  ◇ 主記憶上に存在しているかどうか、  
  ◇ (主記憶上にある場合) 何番目のページフレームに入っているか、  
  ◇ 書き込み可能かどうか、修正されたかどうか、

  などの情報を保持した表(ページ表と言う)が用意され、この表を用いて仮想アドレスから実アドレスへのマッピングがハードウェア的に行われる。

仮想記憶空間

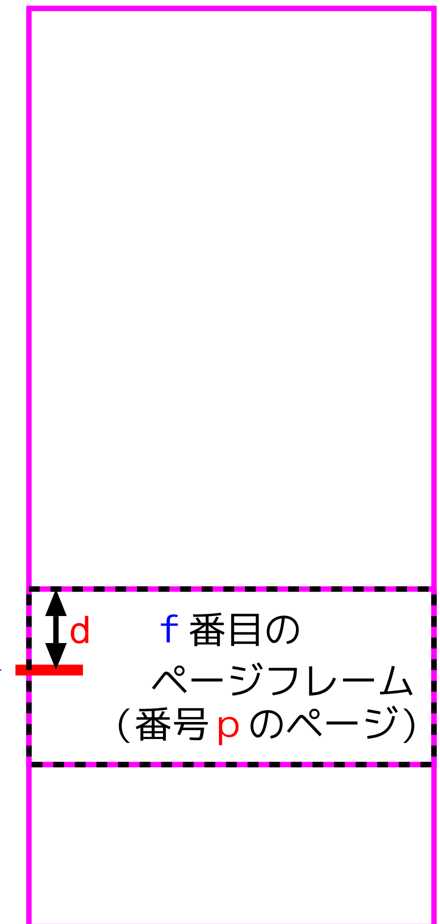



ページ表

| 制御情報 | | | ページフレームの番号 |
| --- | --- | --- | --- |
| 1 | | | |
| 2 | | | |
| p V | C | P | f |

主記憶上の番地
を表すビット列 ＝

主記憶

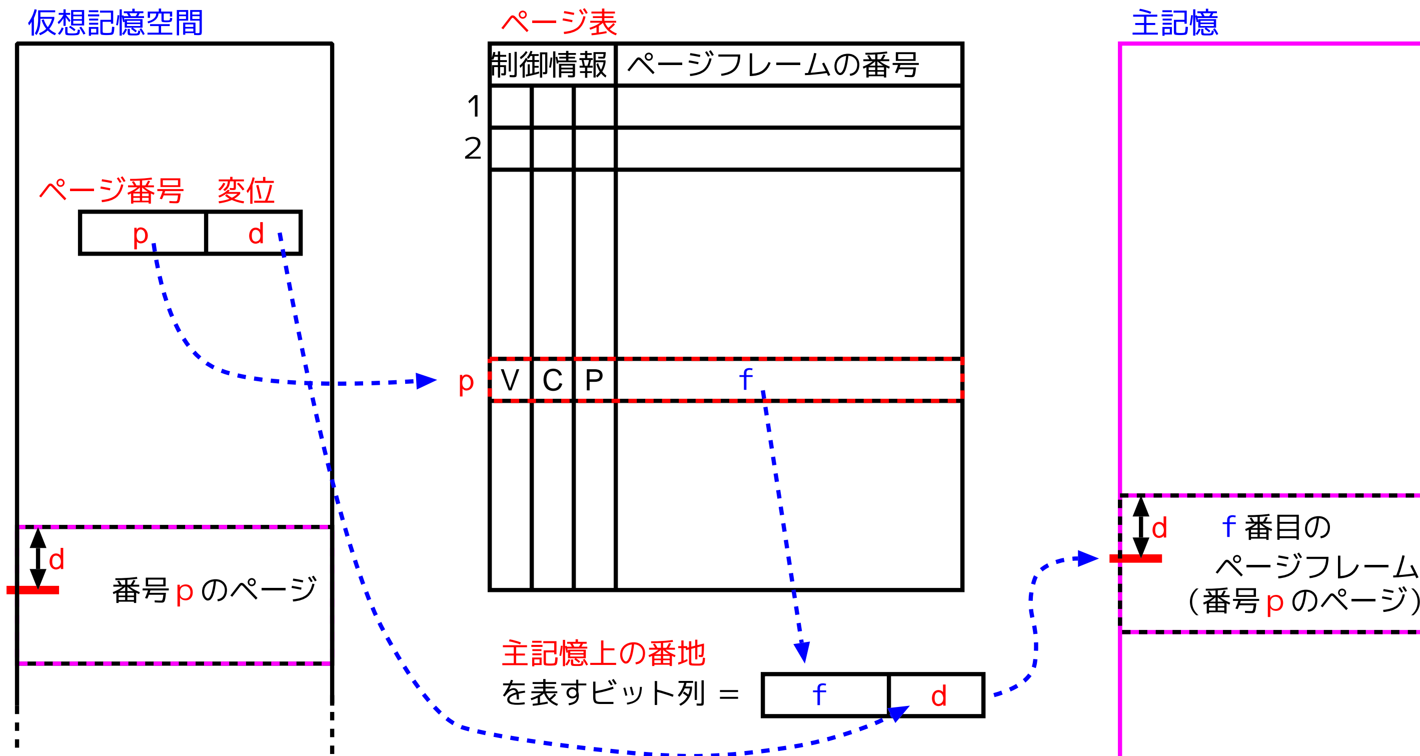



$$
\mathrm{V} = \begin{cases} 1 & \text{if ページが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}
$$

$$
\mathrm{C} = \begin{cases} 1 & \text{if ページが修正された} \\ 0 & \text{otherwise} \end{cases}
$$

$$
\mathrm{P} = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ 010 & \text{if write only} \\ 011 & \text{if read/write} \\ 100 & \text{if execute only} \\ 101 & \text{if read/execute} \\ 110 & \text{if write/execute} \\ 111 & \text{if read/write/execute} \end{cases}
$$

- ページ表内の「修正されたかどうか」の項目は、ページを主記憶から取り除く際に実際に二次記憶へ掃き出す必要があるかどうかの情報を与えている。




$$V = \begin{cases} 1 & \text{if ページが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}$$

$$C = \begin{cases} 1 & \text{if ページが修正された} \\ 0 & \text{otherwise} \end{cases}$$

$$P = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ 010 & \text{if write only} \\ 011 & \text{if read/write} \\ 100 & \text{if execute only} \\ 101 & \text{if read/execute} \\ 110 & \text{if write/execute} \\ 111 & \text{if read/write/execute} \end{cases}$$

## ページングの長所・短所：

長所 (1) 仮想記憶を実現できる。

(2) ページフレームとページフレームとの間には無駄な隙間は出来ない (i.e. メモリの外部断片化は起きない) ので、メモリコンパクションの操作は必要ではない。

(3) 実際に必要なページだけが主記憶内にあれば良いので、多重プログラミングの多重度を上げることが出来る。

短所 (1) ハードウェアの支援が必要である。

(2) 制御の処理 (e.g. 必要なページが主記憶内に無かった時の処置など) が複雑で、OS によるオーバヘッドが比較的大きい。
更には、スラッシング (ページフォールトが多発し…… ) の危険性もあるので、これを避ける手段が必要になる。

(3) ページ表が大きくなる。

(4) ページフレーム内に使用されない領域が存在する。(メモリの内部断片化と呼んでいる。)

セグメンテーション：

プログラムを手続き,データ領域,...などの論理的な単位(セグメントと呼ぶ)に分割し、これらの単位毎に実メモリとの対応付けを取る。

- 機械語命令上では、番地は

  セグメント番号と
  セグメントの先頭からの変位

  の組で表される。 [アーキテクチャにも影響を与えている。]

これにより、

◇ 仮想アドレス空間は2の冪乗の固定長の大きさに区分けされ、

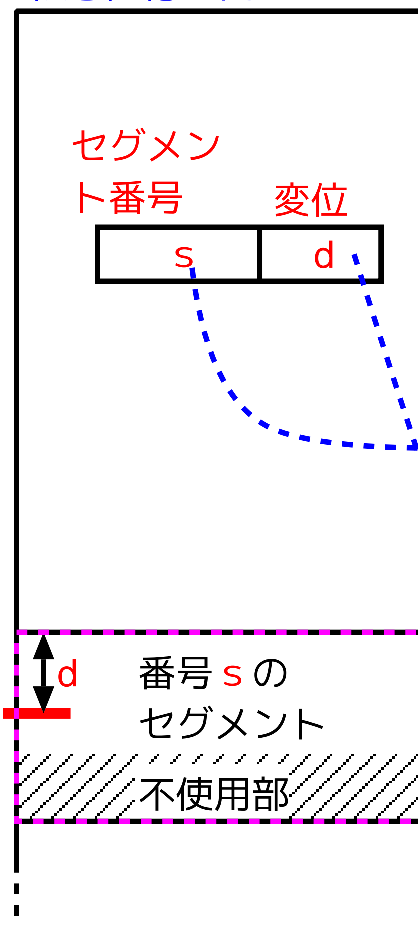
◇ 各々のセグメントは1個ずつどれかのパーティションに前に詰めて格納される

ことになる。

- 与えられた番号のセグメントが
  - ◇ 主記憶上に存在しているかどうか、
  - ◇ 主記憶上の何番地から始まっているか(基底アドレスと言う) 、
  - ◇ どの位の大きさか、
  - ◇ 書き込み可能かどうか、修正されたかどうか、

  などの情報を保持した表(セグメント表と言う) が用意され、この表を用いて仮想アドレス(**2** 次元アドレス) から実アドレスへのマッピングがハードウェア的に行われる。

---

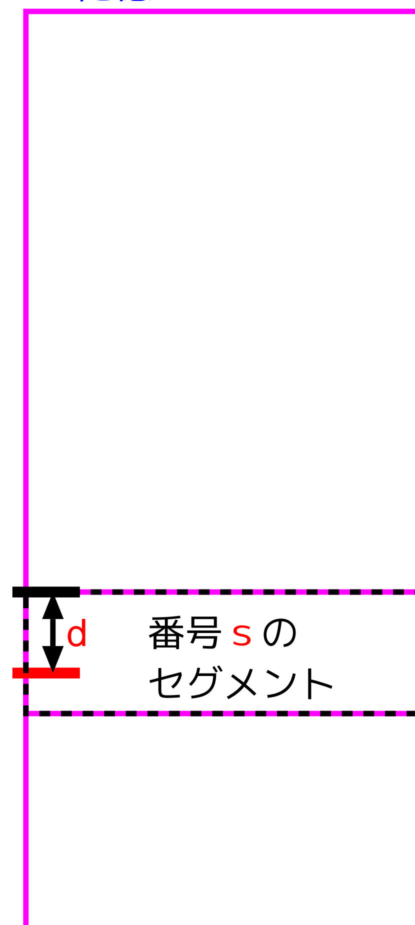



セグメント表

| | 制御情報 | | 基底アドレス | 大きさ |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| s | V | C | P | B | L |

主記憶上の番地 = B + d

（d>L なら記憶保護違反）




$$V = \begin{cases} 1 & \text{if セグメントが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}$$

$$C = \begin{cases} 1 & \text{if セグメントが修正された} \\ 0 & \text{otherwise} \end{cases}$$

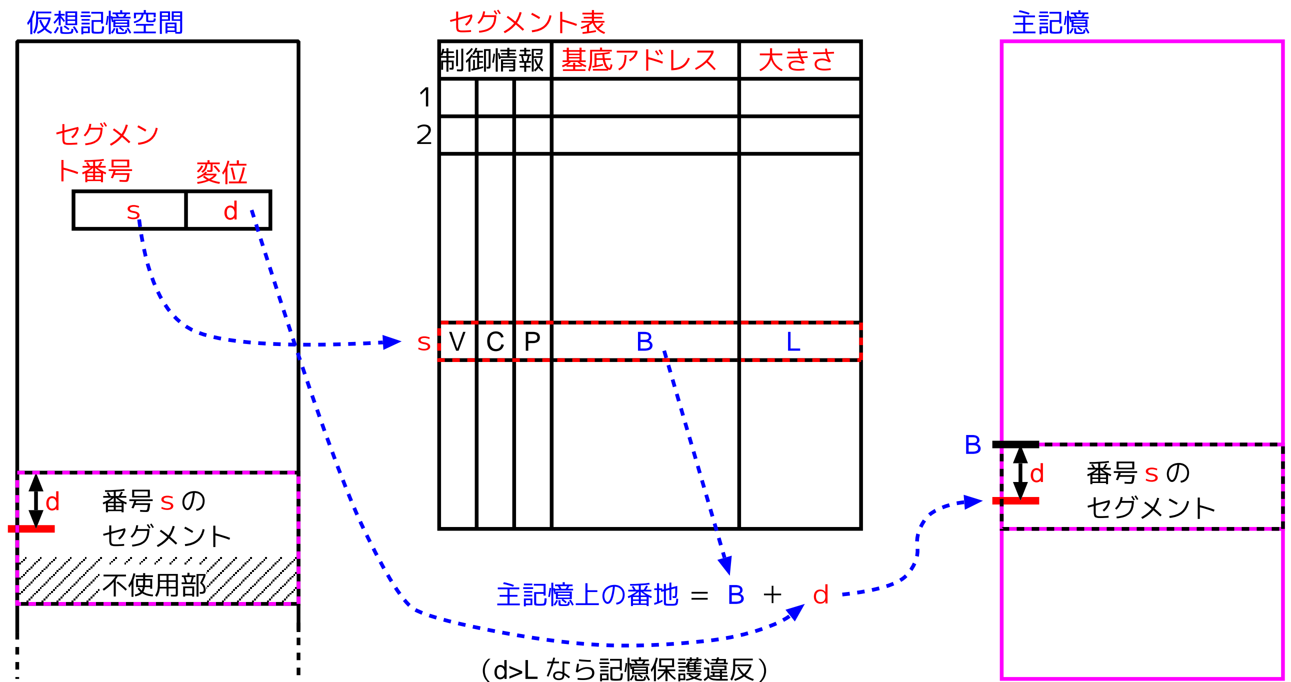
$$P = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ 010 & \text{if write only} \\ 011 & \text{if read/write} \\ 100 & \text{if execute only} \\ 101 & \text{if read/execute} \\ 110 & \text{if write/execute} \\ 111 & \text{if read/write/execute} \end{cases}$$

- セグメント表内の「セグメントの大きさ」の情報は、記憶保護違反のチェックに使われる。
- セグメント表内の「修正されたかどうか」の項目は、セグメントを主記憶から取り除く際に実際に二次記憶へ掃き出す必要があるかどうかの情報を与えている。
- 9.2節で述べたオーバレイと異なり、セグメントの主記憶へのロード、二次記憶への掃き出しは事前の指定無しでシステムが自動的に行う。

$$\mathrm{V} = \begin{cases} 1 & \text{if セグメントが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{C} = \begin{cases} 1 & \text{if セグメントが修正された} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{P} = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ 010 & \text{if write only} \\ 011 & \text{if read/write} \\ 100 & \text{if execute only} \\ 101 & \text{if read/execute} \\ 110 & \text{if write/execute} \\ 111 & \text{if read/write/execute} \end{cases}$$



## セグメンテーションの長所・短所：

長所 (1) 仮想記憶を実現できる。

(2) 複数のプロセス間でセグメントの共用が可能。

(3) セグメントの大きさを動的に増減することが出来る。

短所 (1) セグメントの大きさは実メモリより小さくないといけない。

(2) セグメントの大きさが一定でないため、主記憶上のセグメント間に隙間が出来る。(外部断片化)

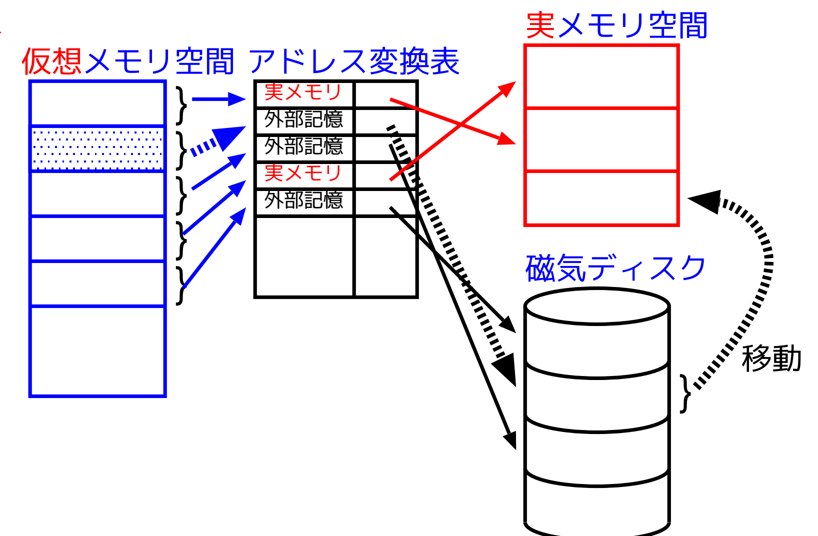
そのため、主記憶上のセグメントを隙間なく再配置して小さな空き領域を無くす、メモリコンパクションの操作が必要になる。

(3) セグメントの大きさが一定でないため、主記憶や補助記憶の管理が (ページングの場合に比べて) 面倒になる。

# 9–13 単一仮想記憶 vs. 多重仮想記憶

- 単一仮想記憶 ··· 全体に対して1つの仮想空間。
- 多重仮想記憶 ··· プロセス毎に別々の仮想空間。

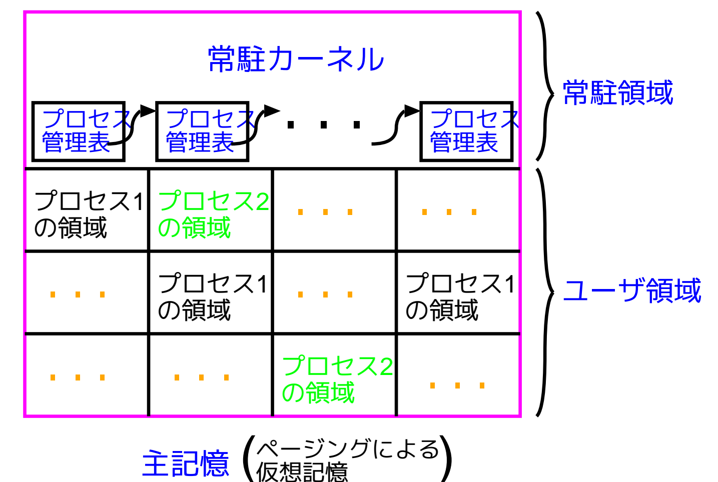
p.100 からの引用：　単一を想定 →




p.88 からの引用：

プロセスの管理／seamless な切替えのために OS が必要とする情報は全てプロセス管理表に保持され、主記憶内の常駐領域に置かれる。プロセス管理表には次の様な情報が入っている。

- 名前 ··· 人間がプロセスを識別するために付けた名前
- ············
- 空間情報 ··· プロセスが利用している実メモリ空間や仮想メモリ空間に関する情報



## 単一仮想記憶 ：

コンピュータシステム全体に対して１つの仮想空間を設ける。

- 仮想アドレス空間を複数のパーティションで区切り、多重プログラミングを実現できる。
- フラグメンテーションは発生するが、空き領域に実ページを割り当てることはないので実害は少ない。
- 仮想記憶が本格的に利用され始めた1970年代の始めは大半がこの方式。

補足：
仮想記憶なしのOSのかなりの部分をほぼそのまま利用できたので、多くのシステムではまず単一方式の仮想記憶が採用された。

しかし、

- 各々のプログラムは仮想アドレス空間の一部しか利用できないので、大きさの制限も強い。
- プロセス間の記憶保護について考えないといけない。
- プロセスごとに先頭アドレスを変える必要がある。(⟶動的リンク)

## 多重仮想記憶 ：

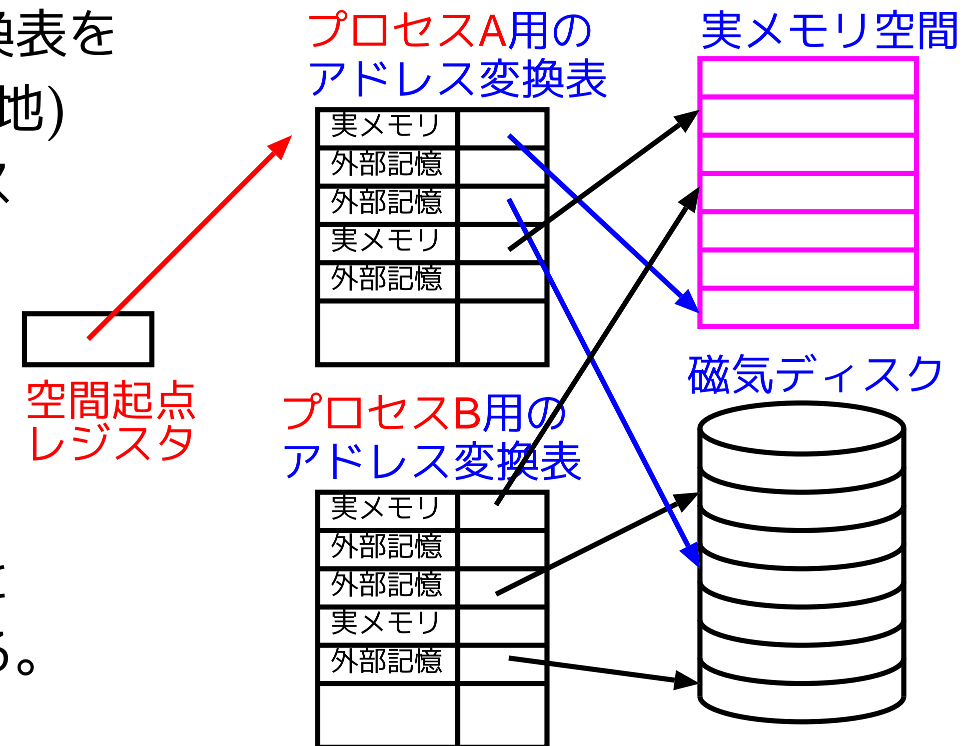
各プロセスに対して別々の仮想アドレス空間を設ける。

- アドレス変換表はプロセス毎に用意し、プロセスの切り換えと伴にアドレス変換表も切り替える。

これは次の様に行う。

◇ プロセス生成時にアドレス変換表を作成し、その表の所在位置 (番地) を生成したプロセスのプロセス管理表の中に格納しておく。

◇ プロセス切替時には、プロセス管理表の中からアドレス変換表の所在位置を読み出して、それを「空間起点レジスタ」と呼ばれるレジスタにセットする。




◇ CPU は、空間起点レジスタの指すアドレス変換表を使うことにより切替えられたプロセスのアドレス変換を行い、プロセスの処理を進める。

- アドレス空間はプロセス毎に独立なので、各プロセスの領域はハードウェア的に完全に他プロセスの不当アクセスから保護される。
- システム内の総アドレス空間の広さに原理的な制約はなくなった。

補足：

新しいプロセスが加わればその分だけアドレス空間が広くなると考えられる。

しかし、実際上は

◇ **CPU** の能力（アドレス変換を高速に行うための仕組が必要; **e.g.TBL**),
◇ 実記憶容量 (少な過ぎるとスラッシングを起こし易くなる),
◇ 仮想記憶をサポートするバッキングストア容量 (総アドレス空間はこの容量よりは広くできない)

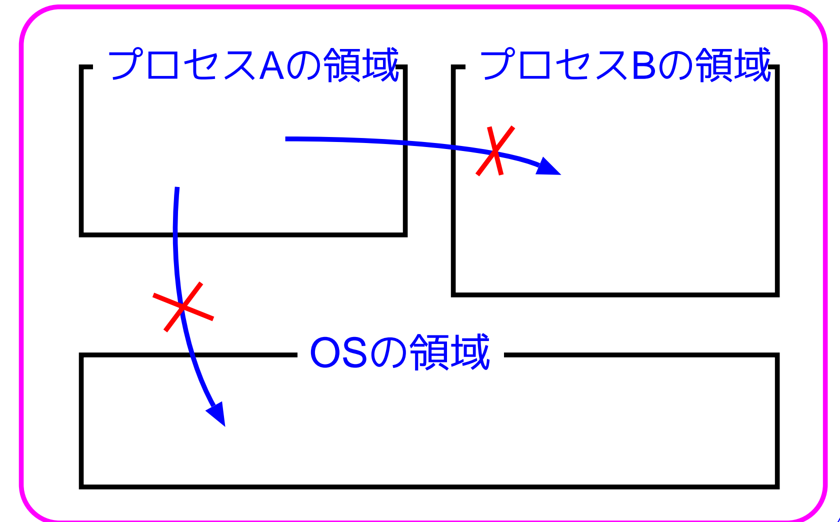
による制約もあるので、多重仮想記憶を実装／運用していくには性能評価を十分に行っておく必要がある。

- 近代の **OS** はほとんどこの多重仮想記憶を採用している。

# 9–14 記憶保護の機構

記憶保護の問題： ···(p.97からの引用)
多重プログラミング環境においては複数のジョブが同時に主記憶領域を使うことになる。 ……

⇨ 1つのジョブから別のジョブに割り当てられた主記憶領域への不当アクセスは絶対に避けなければ...。




どういった場合に記憶保護すべきか：

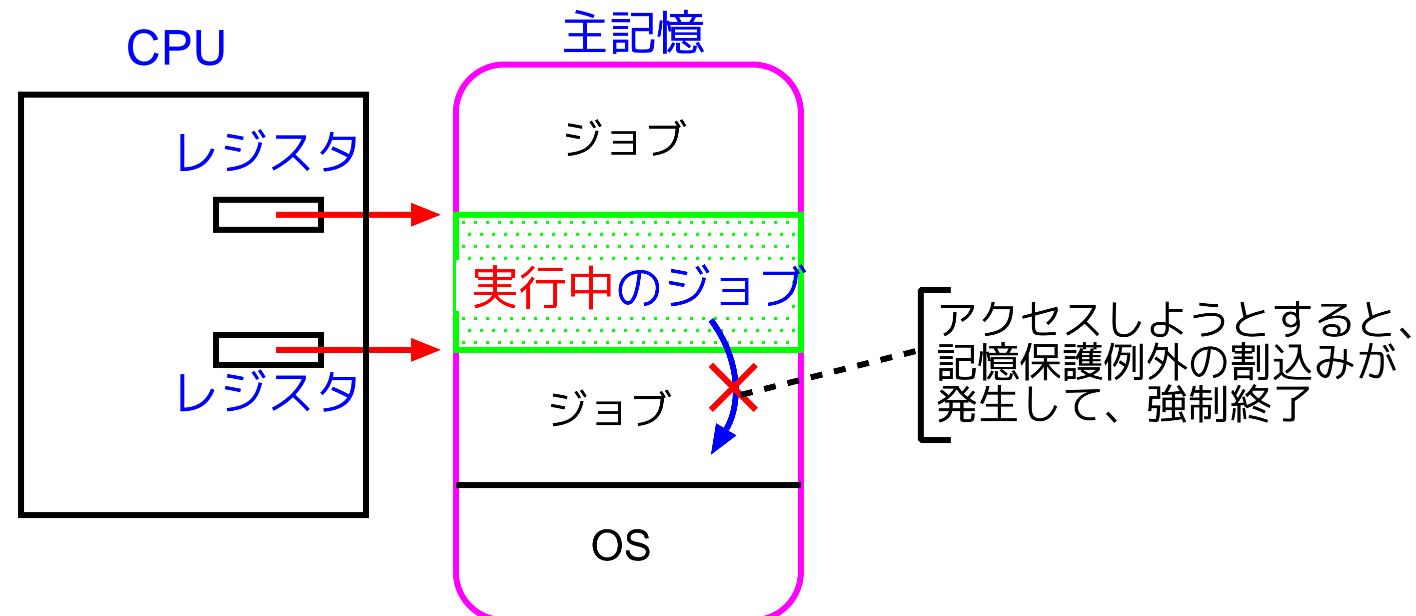
- 一般プロセスから**OS**の領域へのアクセスは許さない。
- 一般プロセスから別のプロセスの領域へのアクセスは許さない。
- 自プロセス内であっても、プログラム部分への書き込みは許さない。

## OS や他プロセスの領域へのアクセス制御：

### (方法 1) メモリ保護境界レジスタを用いる。

1 つのアドレス空間内に複数のジョブが混在する環境を想定した方法である。

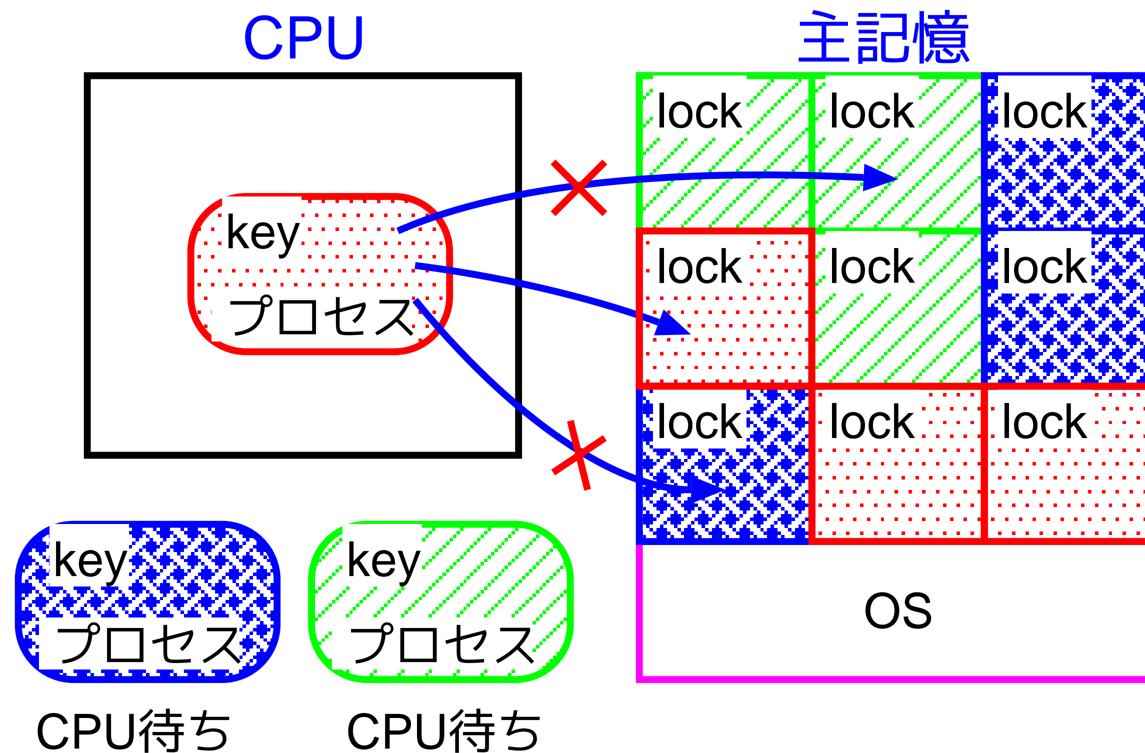
ジョブが CPU 資源を割り当てられた時点で、そのジョブの使っている領域の境界を特別なレジスタ (メモリ保護境界レジスタと呼ぶ) に記憶する。 これによって、実行中のプログラムがアクセス可能なメモリの範囲は常に分かることになるから、その外側の領域にアクセスしようとすると記憶保護例外の割込みを発生させる。



## (方法2) キー/ロック機構を用いる。

仮想記憶の様に、主記憶が複数のブロックに分割されている環境を想定した方法である。

プロセスには**キー** (key,鍵) を与え、そのプロセスの利用可能なブロックには対応する**ロック** (lock,錠前) を掛ける様にし、メモリアクセス時にキーとロックが対応するかどうかを調べることにより、不当アクセスをチェックする。

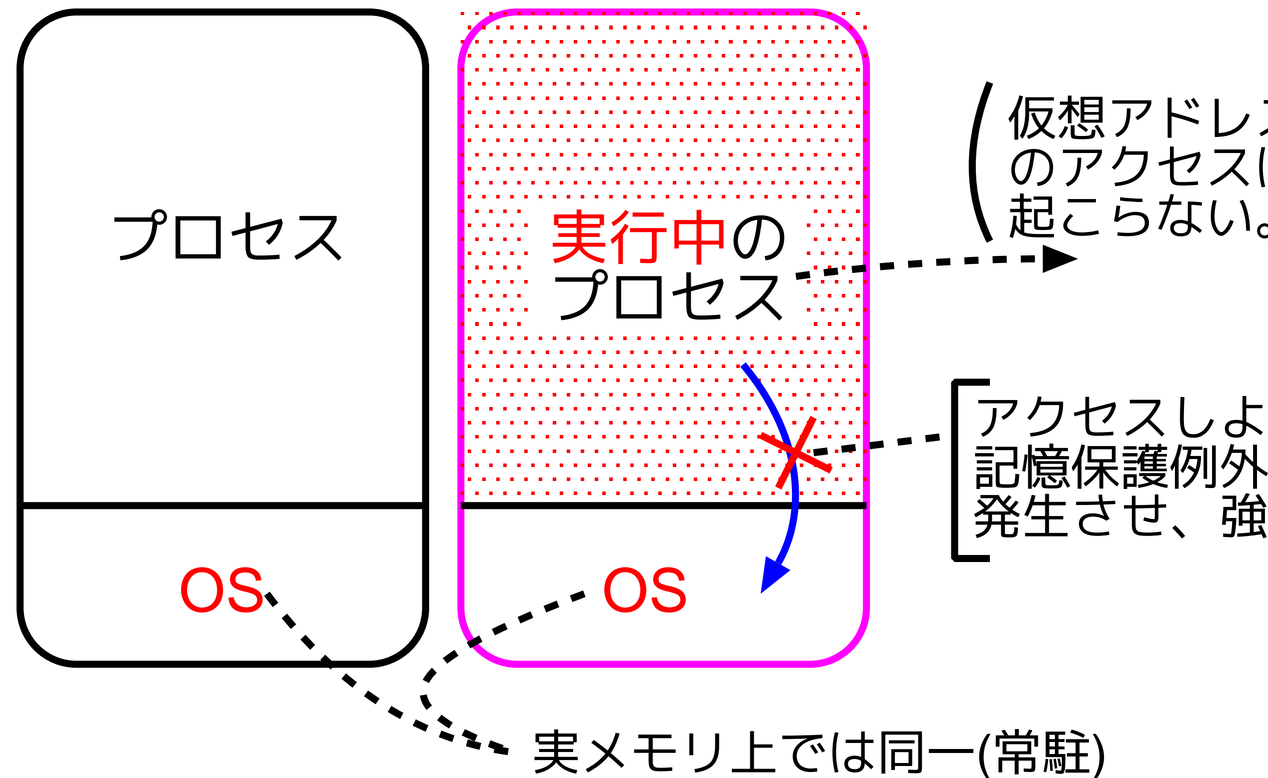



補足：
IBM System/360(1964年) で初めて採用された。その時のブロックは 2048 byte、キーは 4bit。

## (方法3) 仮想アドレス空間の壁の恩恵を受ける。

仮想アドレス空間の中では、変数参照のアドレスがどの様に設定されたとしてもその仮想空間外を参照することはあり得ない。従って、多重仮想記憶の下では、プロセス間の不当アクセスは原理的に起こらない。

多重仮想記憶の場合でも、
プロセスはOSと1つの仮想アドレス空間を共有することになるので、プロセスがOSの領域に不当アクセスするのは防がないといけない。そのためには、例えば各ブロックにOSの領域かどうかのロック(1ビット)を付け、CPUがスーパバイザモードでない場合はOS領域へのアクセスを禁止すれば良い。



多重仮想記憶の場合でも、

プロセスはOSと1つの仮想アドレス空間を共有することになるので、プロセスがOSの領域に不当アクセスするのは防がないといけない。そのためにはキー/ロック機構やメモリ保護境界レジスタの考えが有効である。例えば、各ブロックにOSの領域かどうかのロック(1ビット)を付けCPUがスーパバイザモードでない場合はOS領域へのアクセスを禁止すれば良い。

プロセスとOSが仮想アドレス空間をどう共有するか：

**(BSD)** 1ページ=512バイト, 32ビットマシンであるVAX-11のアドレス空間モデルを採用している。4GBのアドレス空間は次の4つの領域に分けられる。

- **P0(program)** 領域(1GB) ··· プロセスのテキスト(実行コード)とデータが格納される。
- **P1(control)** 領域(1GB) ··· ユーザ・スタック, u領域, カーネル・スタックが格納される。
- **S0(system)** 領域(1GB) ··· カーネルのテキストとデータが格納される。
- 最後の1GB領域(1GB) ··· システム用に予約済。詳細未定。

**(Linux)** 1ページ=$2^{12}$バイトである。4GBのアドレス空間は次の2つの領域に分けられる。

- ユーザ セグメント ··· 3GB。
- カーネル セグメント ··· 1GB。

## プロセス内の記憶保護：

通常、アドレス変換表には、仮想記憶を実現したり

実メモリ $\Longleftrightarrow$ 補助記憶

の間の間のブロックの移動の手間を出来るだけ減らしたりするために、

- ◇ ページ/セグメントに対して実メモリが割当てられているかどうか,
- ◇ ページ/セグメントが修正されたかどうか

を記録するフラッグの欄が設けられているが、

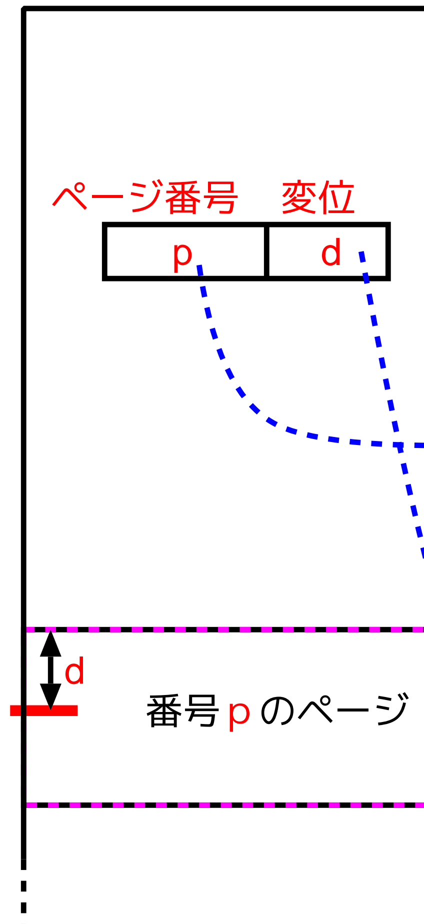
記憶保護のために、

- ◇ ページ/セグメントが読出し可能かどうか,
- ◇ ページ/セグメントが書込み可能かどうか,
- ◇ ページ/セグメントが実行可能かどうか

を記録するフラッグの欄も設けられている。これらの保護モードのフラッグを使って、次の様な誤った操作を避けることが出来る。

- 自プロセス内の、プログラム部への書き込み,
- 自プロセス内の、プログラム部の (データと見ての) 読み出し,
- 自プロセス内の、データ部の実行

# p.103からの引用：

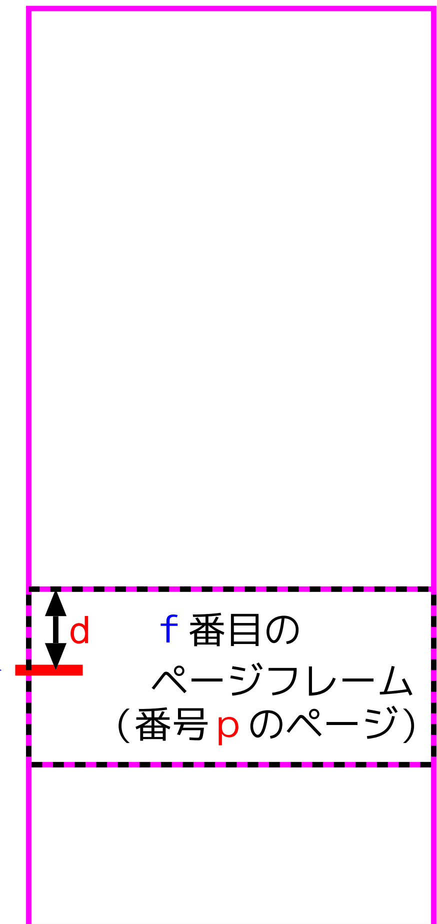



ページ表

制御情報　ページフレームの番号

1
2
p　V　C　P　f

主記憶上の番地
を表すビット列 ＝




$$V = \begin{cases} 1 & \text{if ページが主記憶内にある} \\ 0 & \text{otherwise} \end{cases}$$

$$C = \begin{cases} 1 & \text{if ページが修正された} \\ 0 & \text{otherwise} \end{cases}$$

$$P = \begin{cases} 000 & \text{if アクセスは許されない} \\ 001 & \text{if read only} \\ 010 & \text{if write only} \\ 011 & \text{if read/write} \\ 100 & \text{if execute only} \\ 101 & \text{if read/execute} \\ 110 & \text{if write/execute} \\ 111 & \text{if read/write/execute} \end{cases}$$