

メモリ管理での課題についてです。教科書では第9章です。

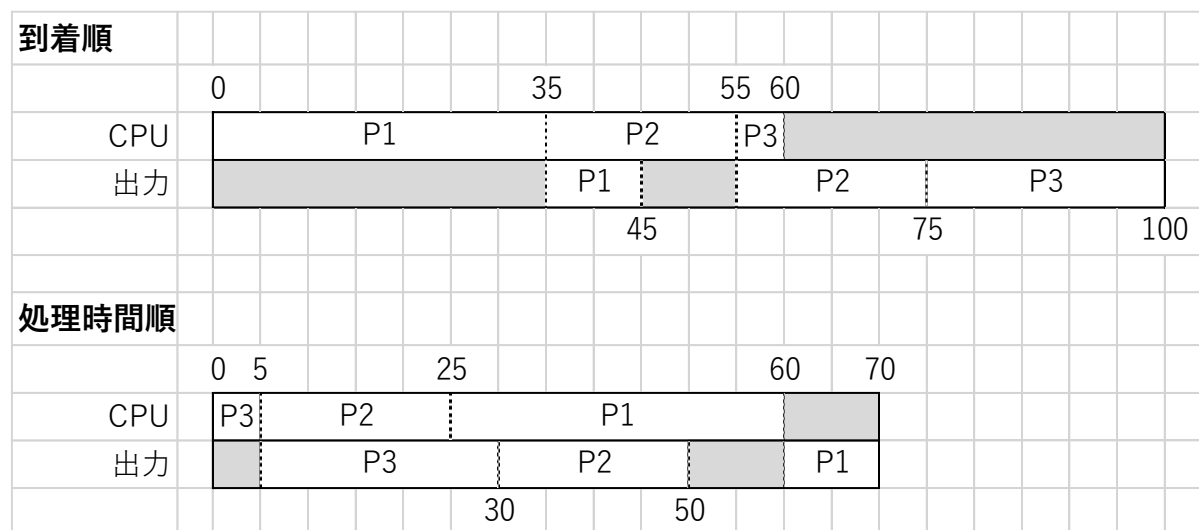
オペレーティングシステム

(2024年 第5回)

メモリ管理について(1)

前回の課題について

下図のようなタイムチャートを作成するとわかりやすいのではないかと思います。(詳細は「第4回の課題について.pdf」をみてください)



到着順の場合、まずP1のCPU処理が実行され35秒間かかります。それが終わった後にこのプロセスの出力処理が10秒間実行されますが、これと同時に2番目に到着したP2のCPU処理が実行でき20秒要します。これからP1の実行完了には45秒かかることになります。

P2の出力はP2のCPU処理が終了した後に時刻55秒から開始され（45秒から55秒までは出力は動作していない状態となります）、これと同時にP3のCPU処理が実行され5秒要して時刻60秒で終了します。P2の出力処理は20秒要しますので、時刻75秒に終了します。P2の実行には（システム開始から）75秒かかることになります。P3のCPU処理は終わっていますがP2の出力処理が続いていたのでP3の出力処理が行えず、時刻75秒からP3の出力処理が開始して25秒で終了します。P3はシステム開始から100秒で実行が完了します。

P1: 45秒、 P2: 75秒、 P3: 100秒（なお、平均の実行時間は73.3秒）

CPUの利用率は、全体の実行時間100秒の中で実際にCPUが動作してプロセスを処理していた時間60秒から、60%となります。

出力の利用率は、 $55/100$ で55% となります。



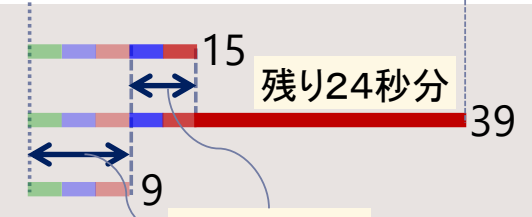
同様に、処理時間順では、 P1: 70秒、 P2: 50秒、 P3: 30秒（なお、平均の実行時間は50秒）

CPUの利用率は、全体の実行時間70秒の中で実際にCPUがプロセスを処理していた時間60秒から、85.7% 出力の利用率は、55/70で78.6%

それぞれの方式のメリットとデメリットは、教科書や前回スライドの記述を確認してください。

利用率については、教科書第15章も読んでおいてください。

▶ 他のスケジューリングの例(教科書 p.96 のもの、P1 と P2 が入れ替わっていることに注意)

アルゴリズム	プロセス (実行 所要時間)	実行のタイムチャート	実行完了 時間の平均
到着順	P1 (6秒) P2 (30秒) P3 (3秒)		27秒
処理時間順	P1 (6秒) P2 (30秒) P3 (3秒)		17秒
ラウンドロビン	P1 (6秒) P2 (30秒) P3 (3秒)		21秒

ラウンドロビンでのタイムチャートを作るには、いろいろなやり方がある。タイムスライスを仮定してもよいが、3つが同じだけ実行されるということで、P3が完了するまで3つが同じだけ実行され、それぞれ3秒分が3倍(すなわち9秒)で実行され则认为られる

P3が9秒で完了した後、P1、P2の2つが同じだけ実行される。P1はすでに3秒分実行されているので残りは3秒分であり、これが2倍(すなわち6秒)で実行されるので、P1は9+6=15秒で実行完了となる

その後はP2だけが実行され、6秒分実行されているので残りは24秒あり、それが実行されて到着からは15+24=39秒となる

スケジューリングについての補足

教科書 p.97「Linuxではキューの代りに木構造でプロセスを管理してプロセス間の公平性を担保する公平共有 (Fair-share) が採用されている」に関連して

Linuxカーネル2.4でのスケジューリング

- カーネル2.4のスケジューラではRUNキューは単純なリスト構造を採用
 - 構造がシンプル = オーバヘッドが小さい
 - カーネル内ではプリエンプトしない
 - $O(n)$ スケジューラ
プロセススイッチのために優先度の最も高いプロセスをRUNキューから1つ選ぶには、キューを線形探索しなければならない
→ プロセス数が増えると非常に時間がかかる

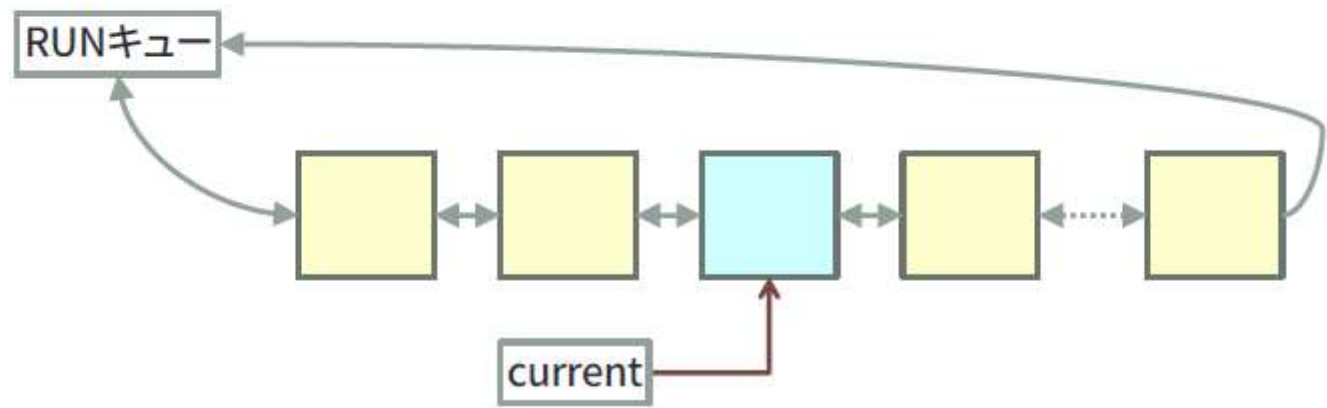
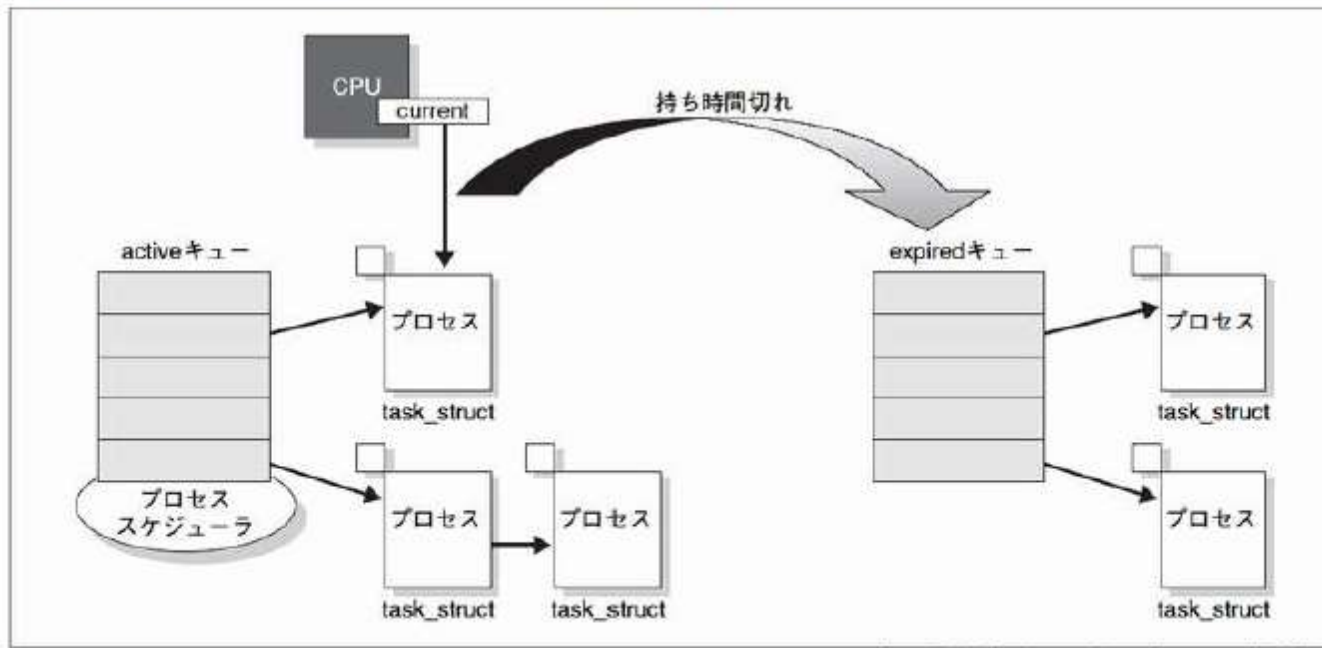


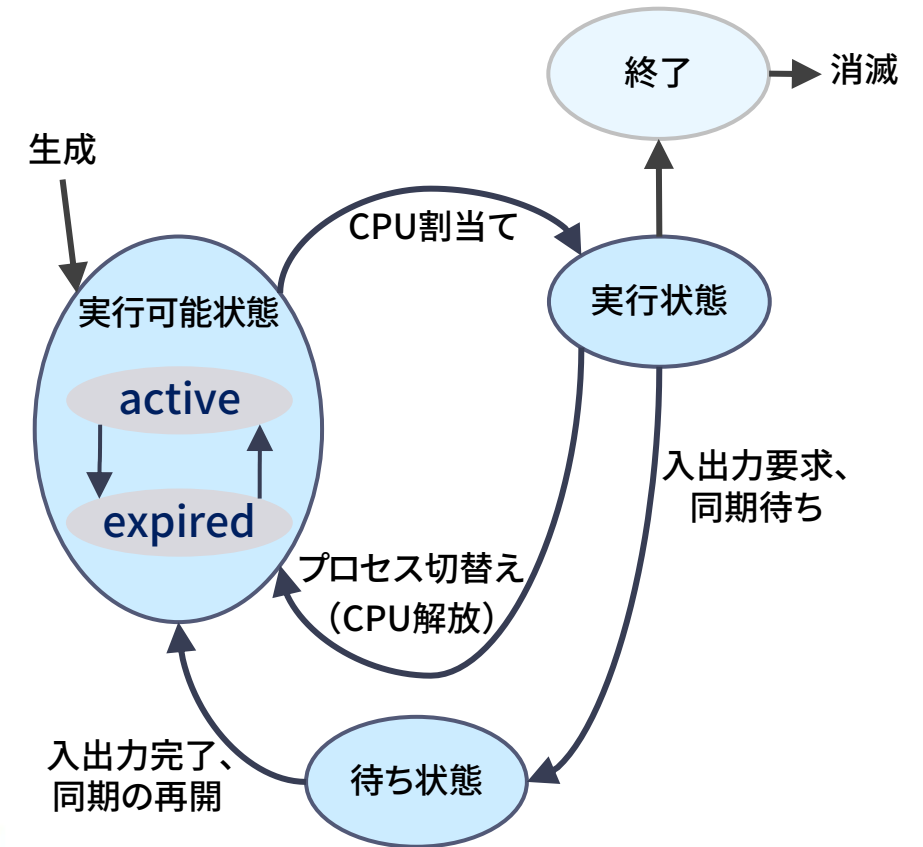
表 記	意 味	例
$O(1)$	定数	配列を添字アクセスする場合
$O(\log n)$	対数	二分探索
$O(n)$	1次	線形探索
$O(n \log n)$	$n \log n$	クイックソート
$O(n^2)$	2次	2重ループを伴う配列全体の走査。バブルソートなど

Linuxカーネル2.6(2.6.22まで)のスケジューリング

- カーネル2.6では優先度別リストを導入(多重レベルフィードバックキュー)
最も高い優先度のスロットから、先頭に登録されているプロセスを選択
 - 実行可能なプロセスがいくつ存在していても、検索量は常に一定
 $O(1)$ スケジューラ
 - 優先度ごとに activeキューと expiredキュー をもつ



UNIX USER 2004年6月号「Linuxカーネル2.6解説室」より



- 単純に優先度の高いプロセスから実行していくとプロセスの実行頻度が大きく偏るという構造的な問題があり、これを避けるためにプロセスの優先度を調整したりしなければならず、実装が複雑になる

Complete Fair Scheduler

- カーネル2.6.23以降では、CFS (Complete Fair Scheduler)と呼ばれるスケジューラを用いている
 - 特定のプロセスに割り当てられた時間を「仮想実行時間」に保持する
プロセスの仮想実行時間が短ければ短いほど、そのプロセスがCPUを必要とする度合いは高くなる
 - プロセスをRUNキューに入れて保持するのではなく、時間で順序付けされた赤黒木で保持する
 - 赤黒木に保存されるプロセスのうち、CPUを最も必要としているもの(仮想実行時間が最も短いプロセス)は赤黒木の左端に保管され、CPUの必要性が最も低いもの(仮想実行時間が最も長いプロセス)は右端に保管される
 - 次のスケジューリング対象として赤黒木の最左端に位置するノードを選択する
プロセスはその実行時間を仮想実行時間に加算してCPU使用時間を計上する

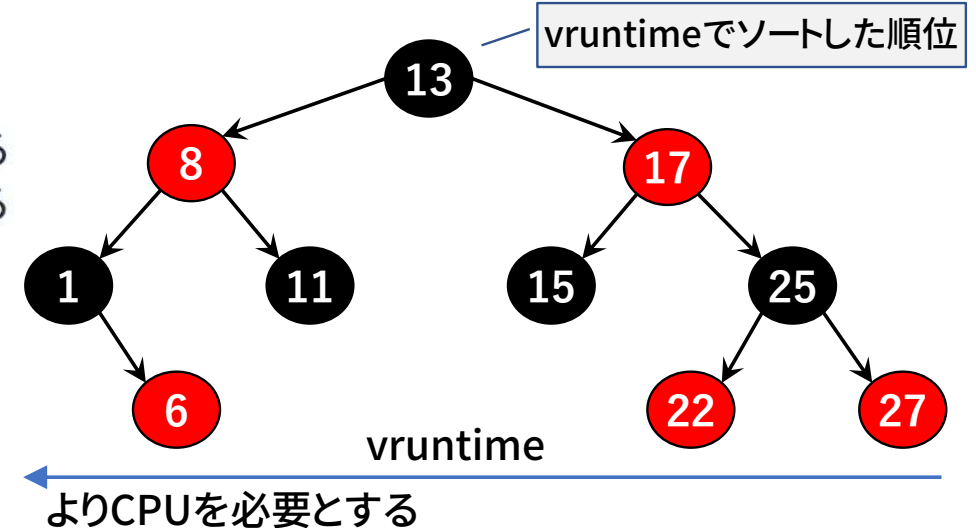
赤黒木: 平衡二分木の一種で、最悪ケースにおける計算量が、データの挿入・削除・検索のいずれにおいても最善。このため、リアルタイムコンピューティングのような時間計算量に敏感なアプリケーションにおいて有益

探索コストの面では赤黒木の操作は $O(\log n)$ なので $O(1)$ に比べて劣るように見えるが、プロセス数が多くなれば、その差は非常に小さくなる

vruntime (プロセスが今までに使用したCPU時間の合計に対して優先度による重み付けを行った値)

1つのプロセスを実行し続けるとvruntimeが増加する。他のプロセスのvruntimeの方が最も小さい値になった時点でプロセス切り替えが行われる

RUNキューはvruntimeの値でソートされ、スケジューラはvruntimeの値が最も小さいプロセスを実行しようとする



メモリ

- ▶ **メモリ**(主記憶装置、メインメモリ、一次記憶装置)
CPUなどから直接読み書きすることができる記憶装置
 - ▶ メモリアドレス
 - … CPUやその他のハードウェアがデータを書き込み、また読み出すメモリ上の位置の一意な識別子(番地)
典型的には整数として表現される
 - ▶ アドレスはメモリ中の1バイトを識別する
 - ▶ アドレス空間
アドレス指定(番地づけ)によってアクセス可能であるメモリの範囲
- ▶ メモリの用途
 - ▶ プログラム(命令語とデータ)を格納する(アプリケーションもOSも)

まず、メモリとか主記憶、メインメモリと呼ばれるものは、CPUなどから直接読み書きできる記憶装置のことです。
メモリを一次記憶、それに対してディスク装置などを二次記憶とよぶこともあります。

メモリは配列のようなものと考えられ、整数値のアドレス(番地)で位置を指定します。アドレスによって位置を指定でき、アクセス可能であるメモリの範囲をアドレス空間と呼びます。「アクセス」という用語はすでに出てきています)

メモリ管理

▶ メモリ管理

プログラムを実行するには、そのプログラムをHDDなどからメモリにロードしなければならない

- ▶ プログラム(プロセス)の要求に応じてメモリの一部を割り当てる
 - ▶ メモリに十分な大きさの領域があることが前提
 - ▶ 連続した領域が必要
 - ▶ 複数のプログラム(プロセス)に、それぞれ領域を割り当てる
 - ▶ オペレーティングシステムにも領域を割り当てる
- ▶ そのメモリが不要となったときに再利用のために解放する
- ▶ 領域の容量が足りない場合の対処
 - ▶ (1つで)メモリに入りきらないプログラム
 - ▶ 多数のプログラムを同時に実行したい(合計が容量を超える)

メモリ管理で行われることです。

プロセスのところでも説明しましたが、プログラムを実行するのにメモリが必要です。プログラム内蔵方式では、プログラムを実行するには命令とデータがメモリ(主記憶)上に格納されていなければなりません。

通常、実行形式のプログラムはハードディスクなど二次記憶装置上にあります。実行に先立ってそれをメモリに置くことが必要になり、ローダ(と呼ばれるプログラム)がそれを行います。(第2回の講義を思い出してください)

このとき、実行するプログラムに応じて、十分な大きさで、連続した領域のメモリ(部分)を割り当てます。

プログラムが終了してその領域が不要となった場合には、解放します。

ここで、主として考えることは、十分な大きさがなかった時にどうするかです。

メモリ管理の方式

- ▶ **マルチプログラミング**(見かけ上、同時に複数のプログラムを実行すること)が前提
 - ▶ メモリに複数のプログラム(プロセス)を置く

メモリ管理に対する個々の(仮想記憶方式以前の)技術

- ▶ **メモリの割り当て方式**
 - ▶ **固定区画方式(固定長領域管理)**
メモリを固定した区画に分け、そこに収まるプログラムを選んでロードする
 - ▶ **可変区画方式(可変長領域管理)**
固定した区画を設けず、プログラムをロードするとき十分な大きさの連続した空き領域を探し、そこに割り付ける

前提として複数のプログラム(プロセス)が同時に動くことを前提としますので、メモリには複数のプロセスの空間(領域)が置かれます。

1つのプログラムだけがメモリに置かれて(OS用のメモリの領域は別にあるものとします)、プログラムが切りかわる度に、そのプログラムを置き換えることは効率的ではありません。

どのような方式で管理するかについて今回は、次回に説明する仮想記憶方式以前の技術を説明します。

メモリ全体を分割して、それぞれにプロセスの空間を与えることになりますが、その分割が固定長であるか、可変長であるか、で方式が分かれます。

固定長といってもすべてが同じサイズというわけではなく、100KBとか200KB、・・・とあらかじめ固定になっているということです。

可変区画方式は、固定サイズを設けずプログラムを格納するのに十分な連続する領域を割り当てるものです。

メモリ領域の割り当て手法

▶ 空き領域の管理

- ▶ リスト方式 空き領域の先頭アドレスとサイズの対をアドレス順や大きさ順にポインタでつないで空き領域を保持する
- ▶ ビットマップ方式 固定サイズの領域をビットに対応させ、そこが空きか使用中であるかを 0/1 で表す

▶ 割り当てアルゴリズム

- ▶ first-fit (先頭適合) 必要サイズを満たす最初に見つかった空き領域を割り当てる
探索が速い、実現が簡単、領域の利用効率はそれほどよくない
- ▶ best-fit (最良適合) 必要サイズを満たす最小の空き領域を割り当てる
領域の利用効率が最も良い (小さな領域の確保に大きな領域が使われにくい、確保したい大きさと一致する空き領域が得られやすい)
使われることのない小さな空き領域が多数生じる可能性をもつ
- ▶ worst-fit (最悪適合) 最大の空き領域を割り当てる
領域の利用効率は最も悪い、他の方法よりも大きな空き領域を作り出す

また、空いている領域(これから使える領域)をどのようにして管理するかということもあります。

固定区画方式ですと、たとえば、100KBの区画が2つ、200KBの区間が3つといったように空き領域があります。可変区画方式では、メモリ上にあったプログラムが終了したとき、それが使用していた領域が解放されて空き領域となります。

そのような空き領域をリストでつないで保持したり、たとえば25KBといった固定サイズに対応させたビットの配列要素で空きかどうかを表したりするのが主な方式です。

メモリを確保するとき、スケジューリングでやったように複数の候補から1つを選ぶことが必要になります。この割り当てのアルゴリズムには次のようなものがあり、それぞれメリット、デメリットがあります。

空き領域の管理

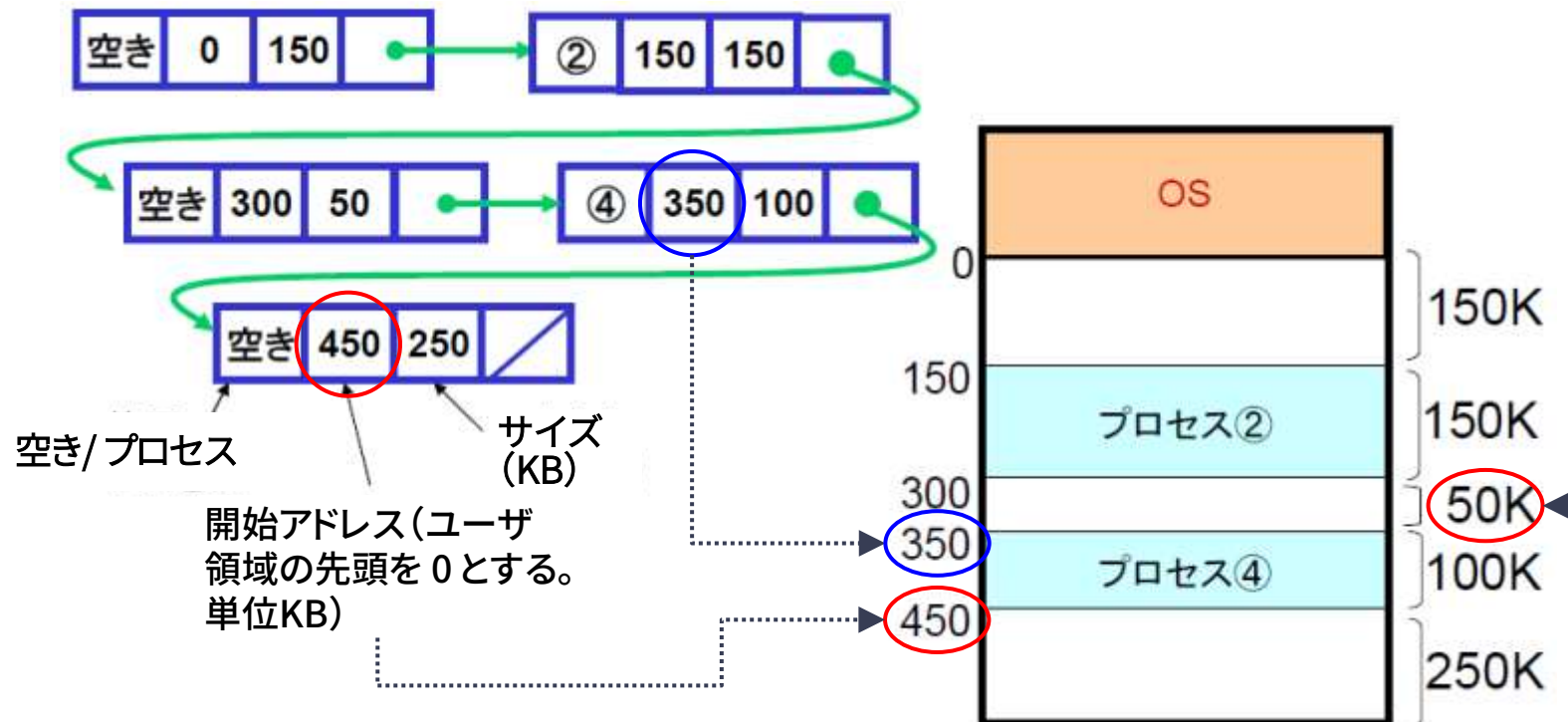
▶ ビットマップ方式

領域の使用/未使用を1/0
に対応させる(25KBを1ビット
に対応させた場合)

0	0	0	0	0	0	1	1
1	1	1	1	0	0	1	1
1	1	0	0	0	0	0	0
0	0	0	0				

前ページの内容を図で表現したものです。
ここで空き領域は0番地から150KBのもの、
300番地から50KBのもの、450番
地から250KBのものとなっています。

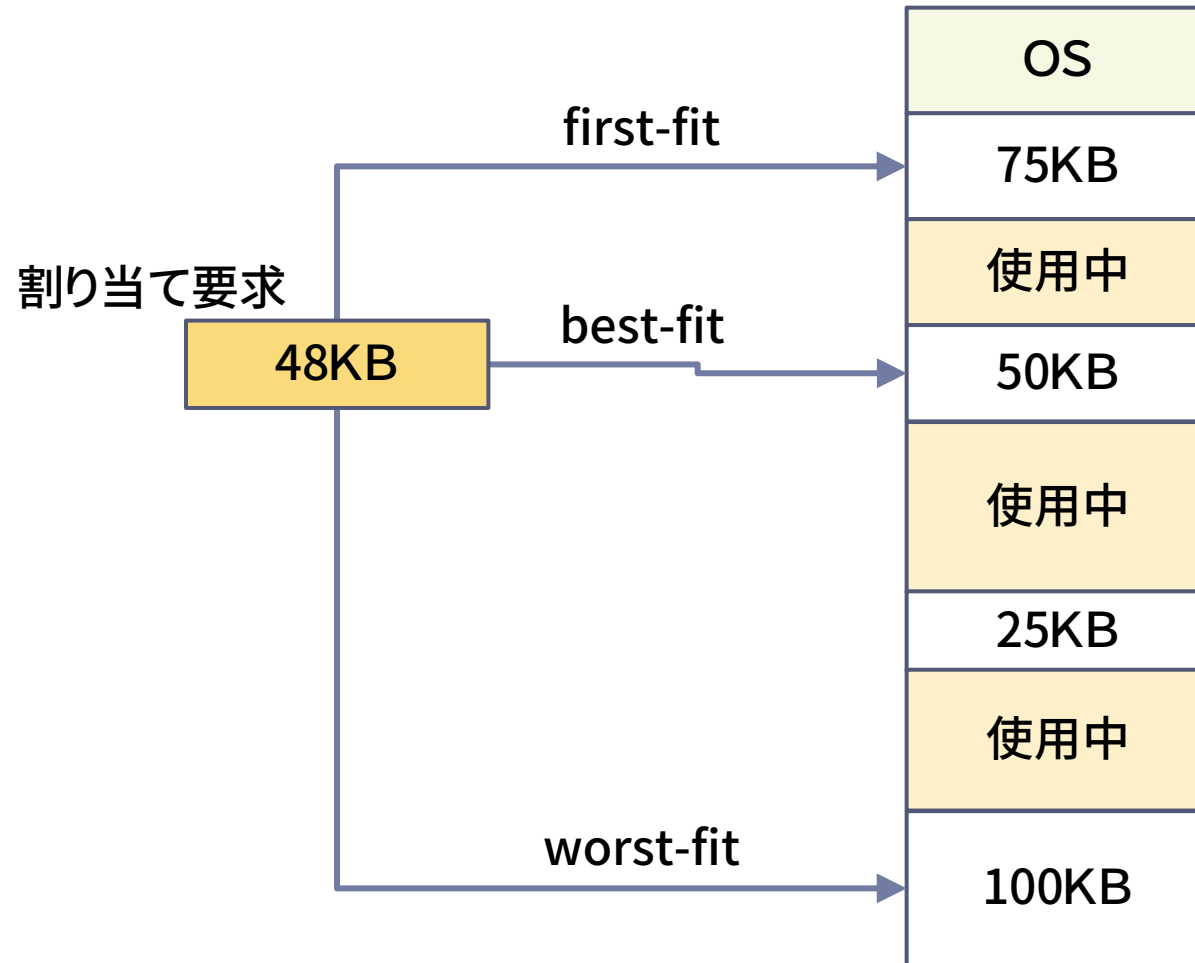
▶ リスト方式



リスト方式では、リストの「空き」と指定されているものをたどることによって得られます。

ビットマップ方式では、25KB単位に対応したビット配列(ビットマップ)の要素の「0」が連続するところで空き領域が見つかります。

割り当てアルゴリズムの例



割り当てアルゴリズムを図で比べると、このようになります。空き領域が、75KB、50KB、25KB、100KBで、この順で管理されているときに、48KBのサイズの要求があるとしてみます。First-fit では 75KB、best-fit では (48KBに最も近い) 50KB、worst-fit では 100KBのものを、それぞれ選択することになります。

(リスト方式では、「空き」のものをたどってサイズを見て、ビットマップ方式では連続する「0」の個数を見て、それぞれ条件を満たすものを選択します)

先のページで説明したメリットやデメリットがあるかどうか、確認してください。

メモリ管理の課題 (1)

▶ プログラムのロードとメモリ確保

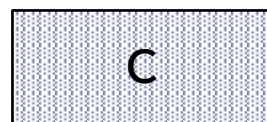
二次記憶装置上にある命令語とデータをメモリに転送(ロード)する

▶ 必要な大きさの連続したメモリ領域の確保

メモリ領域の大きさがプログラムのサイズよりも十分に大きければ、どこかにとれる
そうでない場合は？

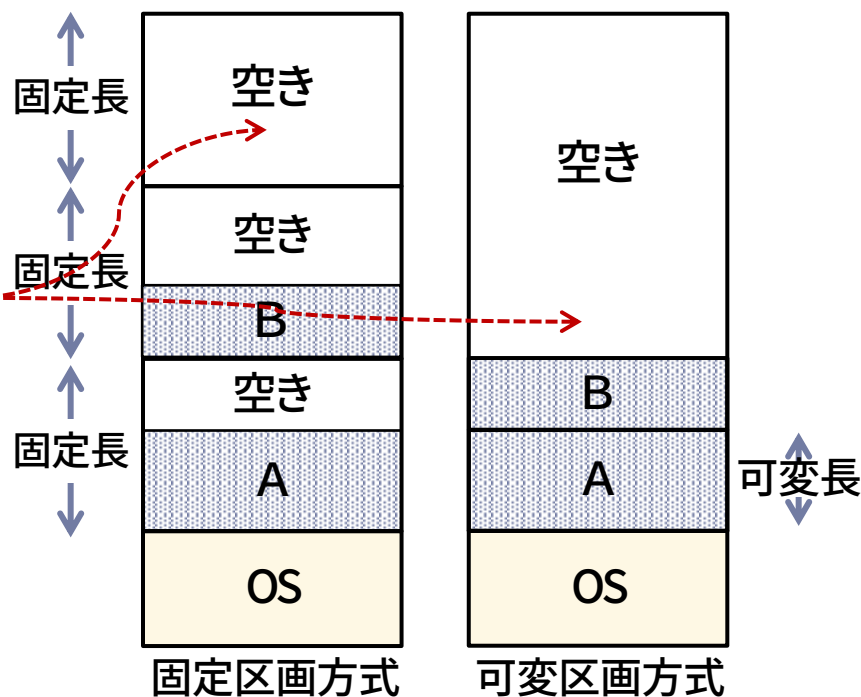
▶ 複数の領域の確保

空き領域の大きさが不足する
場合がある



▶ 断片化(フラグメンテーション)

要求の大きさを満たさない空き
領域が散在する状態



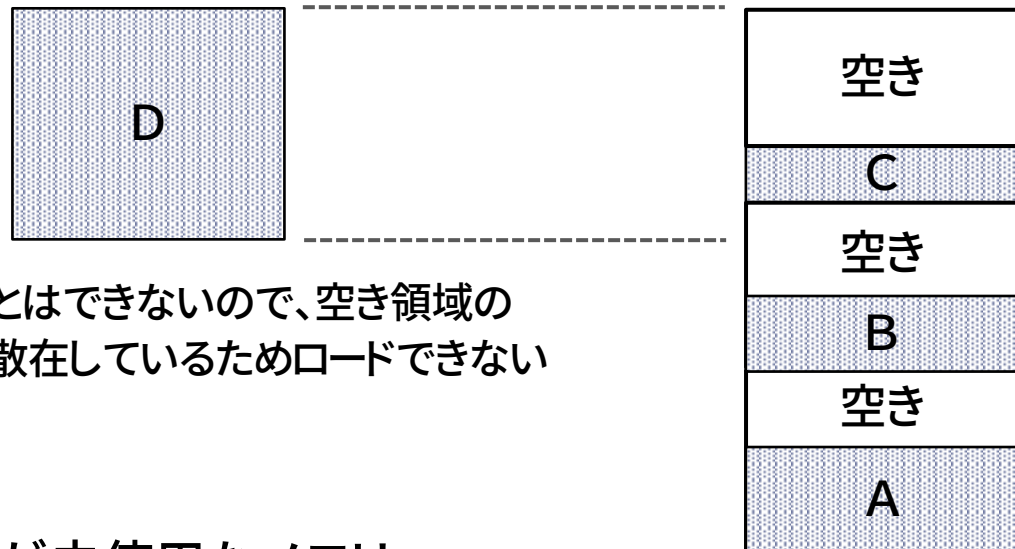
それでは、メモリ管理においてどのようなことを解決しなければならないかについてです。

先にも言いましたが、メモリはプログラムを実行するためにロードするところです。そのために必要な大きさの連続した領域が確保できることが必要で、多くのプログラムがあったり大きなプログラムであったりすると不足する場合があります。この図は空きがある場合ですが、さらに要求があると不足します。この図は、AやBが使用中の領域であり、新たにCの領域を確保する場合です。

固定区画方式の図でみると、空き領域と空き領域の間に使用領域があって、これら空き領域を足すと十分な大きさがあるのに、連続ではないため要求に応えられない場合があります。これは「断片化」と呼ばれるものです。

メモリ管理の課題 (2)

- ▶ 断片化(フラグメンテーション)の発生
空き領域が散在するため、総計は十分であるのにロードできない



- ▶ 割り当てられているが未使用なメモリ

```
#define MAX 10000  
double array[MAX][MAX];  
...
```

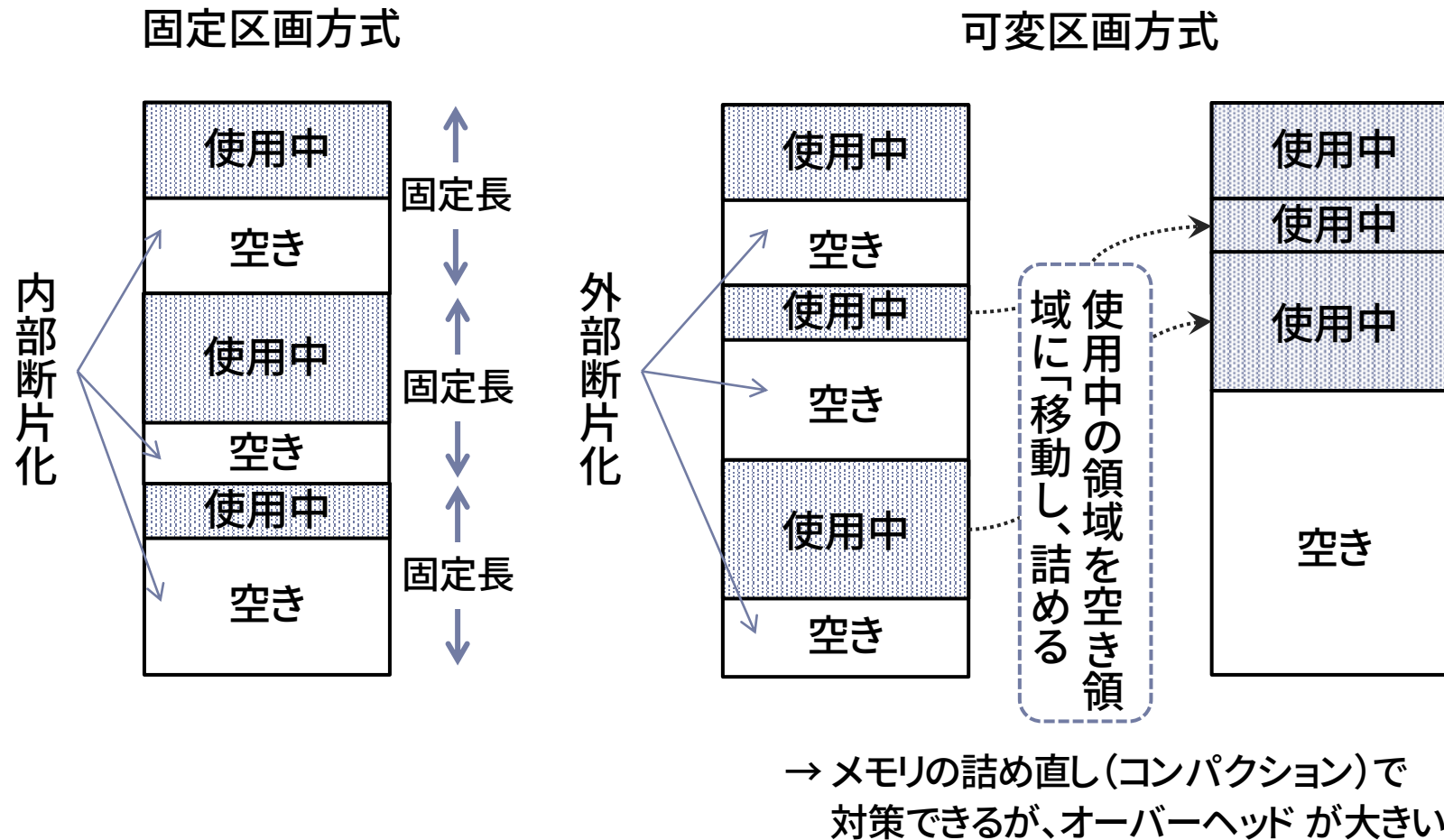
この配列arrayには800MBのメモリ領域を必要とし、実際にはわずかの要素しか使用しなくても 800MB分の確保が必要

断片化は、空きの領域が散らばって存在するため、合計した総量は十分であるのに、連続でないためにロードできないということです。

また、巨大な配列などで全体量は大きいものでも、たとえば、対角要素しかないなど実際に計算に使用する領域の量は小さくなくても、全体の領域を確保しなければならないという問題もあります。

内部断片化と外部断片化

- 断片化は固定区画方式でも可変区画方式でも発生する



断片化につきまして、固定区画方式では、区画の中に利用できない部分が存在するので「内部断片化」と呼び、可変長区画方式では、使用中の区画の外に利用できない部分が存在する(可変長の区間が連続してとられた後、とびとびに解放される場合になります)ので「外部断片化」と呼びます。

外部断片化は、使用中の領域を空き領域に詰めて行くことで、大きな領域を確保することができますが、この操作はオーバーヘッドが大きいものです。また、プログラムを移動させるには、再配置が可能になっている必要があります。

参考:再配置について

▶ プログラムの再配置(リロケーション)

再配置可能 …

プログラムがメモリのどこにロードされても問題なく動作できる性質
リロケートブルプログラム、リロケートブルオブジェクト/バイナリなどという

▶ 再配置ローダによる方法

ローダがロード時に、ロードされるアドレスを用いて、プログラム中の命令やデータのアドレスを変更(再配置)する

- ▶ 再配置の際に修正が必要な箇所をリストアップしたものを付加しておく

▶ 再配置レジスタによる方法

命令やデータのアドレスを、ベースレジスタからの変位で表現する(0番地からロードされるという想定)

- ▶ ベースレジスタの設定はOS、もしくはプログラム自身で行う
- ▶ 配置されるアドレスに依存するようなコードを含まないプログラム(位置独立コード)

▶ 仮想記憶を使用するコンピュータでは、メモリが仮想化されているため、複数のプロセスを同時に実行する際でも再配置を意識しなくてよい

先の詰め直しのように、プログラムのメモリ上の位置を動かす場合、移動されるプログラムは移動されてもちゃんと動作することが必要です。たとえば、データのアドレスとして1500番地を用いている場合、プログラムを移動すると1500番地の内容が変わってしまいます。プログラムがメモリのどこにロードされても問題なく動作できる性質を「再配置可能」と呼びます。

1500番地のデータが移動されて2500番地になったとすると、もともと1500番地を参照していた命令は2500番地を指すようにする変更が必要です。

再配置ローダによってそれを行ったり、1500番地を直接指すのではなく、1000番地を指しているレジスタ+500といった表現をし、このレジスタを2000番地にするといった再配置レジスタで行う方法などあります。このとき命令では「レジスタ+500」という形式で指していて、それを変更する必要はありません。

なお、仮想記憶を使用するコンピュータでは、メモリが仮想化されているため、複数のプロセスを同時に実行する際でも再配置を意識する必要はありません。

メモリ管理の課題 (3)

- ▶ **メモリ容量制限の回避**
そのコンピュータが有するメモリの容量を超えるプログラムを実行する
 - ▶ **オーバーレイ**
プログラムをモジュール単位に分け、ある時点で必要なプログラムとデータだけを主記憶上に持つ
 - ▶ **スワッピング**
主メモリと二次記憶装置の間でプログラムの出し入れを行うこと

もっと根本的に、実装されているメモリが4GBで、それを超えるサイズのプログラムを実行したいときにどうするかという問題もあります。

プログラムを分割できれば、必要な時点で必要なプログラムとデータだけをメモリにもつというオーバーレイ手法があります。

プログラム1つはメモリに入るが、もう1つ（これも1つだけならメモリに入る）を実行しようとするとう限度を超えるという場合にはスワッピングという手法があります。

オーバーレイ

▶ オーバーレイによるメモリ容量制限の回避

プログラムの実行には、これから実行する部分さえメモリにあればよい
他のコードが必要になったら、それまで別のコードが置かれていたメモリ上の領域に必要なコードをロードして使用

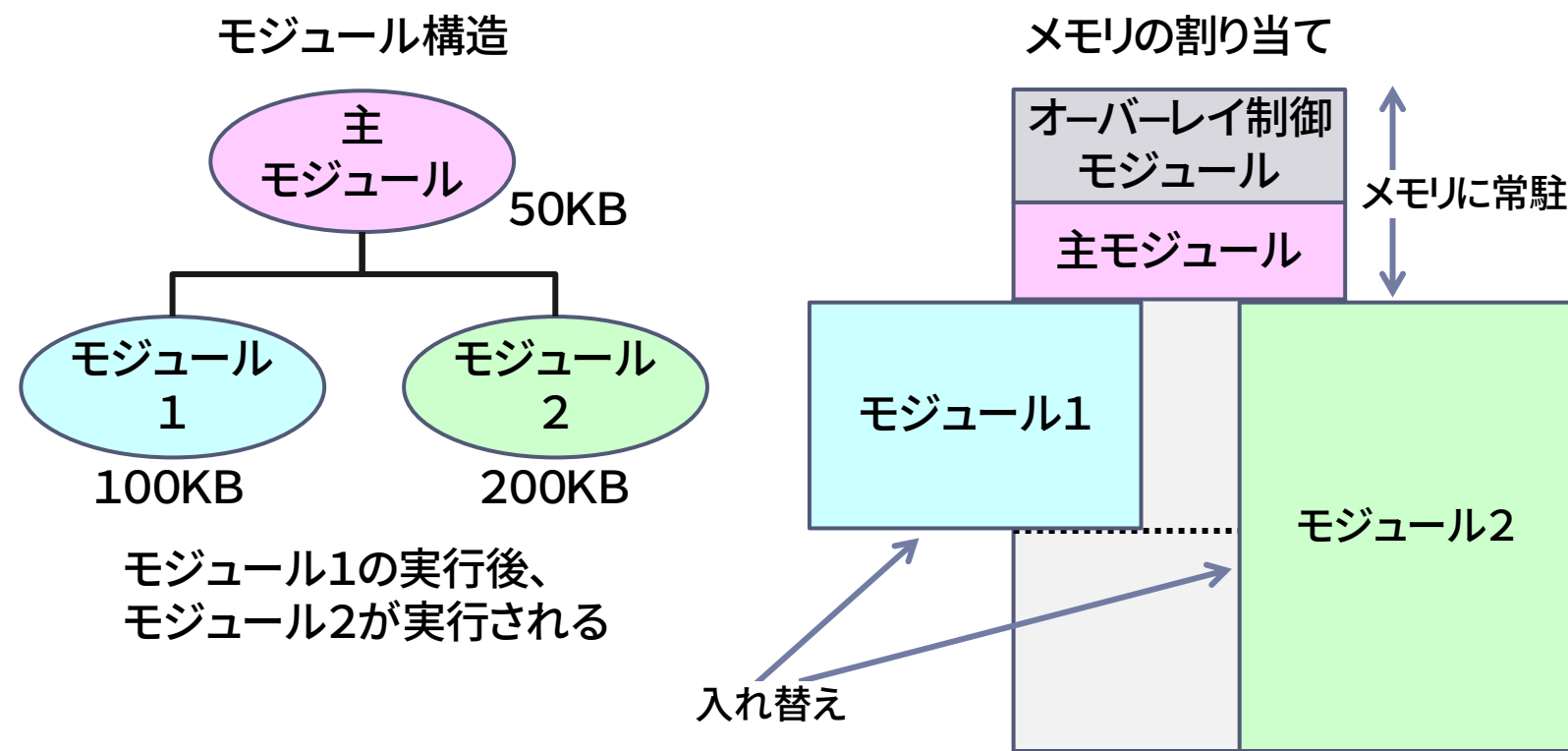
- ▶ プログラムを論理的な単位(手続きやその集まりなど):モジュールに分割して作成する
- ▶ 同時に実行されることのないモジュールを、メモリの領域を重ねるようにロードして使用する(ローディングの単位をセグメントと呼ぶ)
- ▶ 実行済みのモジュールは上書きされる

オーバーレイの考え方は、プログラムの実行には、これから実行する部分さえメモリにあればよいということです。

他のコードが必要になったら、それまで別のコードが置かれていたメモリ上の領域に必要なコードをロードして使用するという事です。

そのために、プログラムをモジュールに分割して作成し、同時に実行されることのないモジュールに対して、メモリの領域を重ね合わせてロードします。すなわち、実行済みの領域に上書きするという事です。

▶ モジュールの入れ替えはプログラム自身で制御

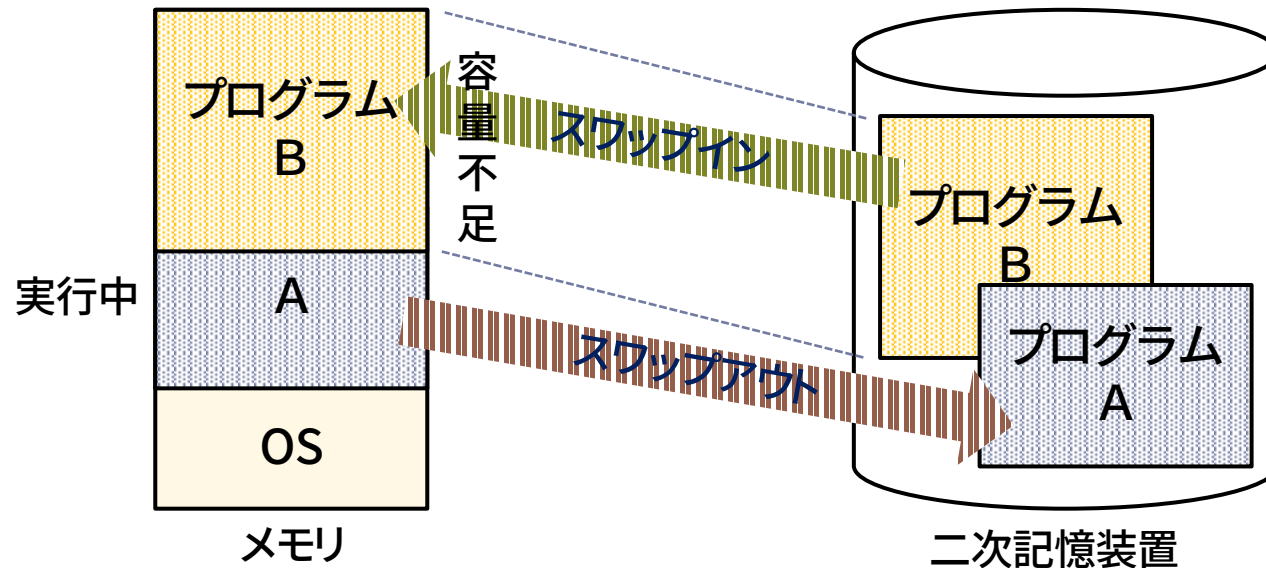


この図は、モジュール1と2がオーバーレイされているものです。主モジュールがモジュール1、モジュール2双方を呼ぶもので、主モジュールはメモリに常駐しています。また、全体を制御するモジュールも常駐することになります。

- ▶ 例えば、この例で、モジュール1とモジュール2とで共通に使用する手続きやデータは主モジュールに置く

スワッピング

- ▶ メモリ領域が不足すると実行中のプロセスの全領域を二次記憶装置に書き出(スワップアウト、swap out)して強制的に空き領域を作り、そこに新しいプロセスをロードする
- ▶ プロセスを再開するときは、二次記憶からメモリに読み込む(スワップイン、swap in)



実行したい複数のプログラムのプログラムをすべてメモリにロードしておくことは、メモリの量に制限があり不可能です。(ここで、1つ1つのプログラムはメモリにロードできるとします)

メモリ領域が不足した場合に、実行中のプロセス(ここではA)のすべての領域をディスクなどの二次記憶装置に書き出して空き領域を作ります。(入出力待ちとなったプロセスを書き出すといったこともあります)。これがスワップアウトです。

空いた領域に別のプログラム(B)をロードします。これがスワップインです。

- ▶ メモリと二次記憶との間でプログラムを転送するのに多大な時間を要する

メモリ管理の課題 (4)

▶ メモリの保護

- ▶ メモリ操作に対するバグ(配列の添字の上下限越えやポインタの誤り)などからの保護
 - ▶ 複数プロセスが動作している場合、どこにバグがあるかを判別するのは非常に困難
- ▶ カーネルの領域だけ保護するなら、実行モードによって行える
- ▶ プロセスが利用する領域の保護は問題

もう一つの課題は、操作してはいけない領域に対する操作がなされないようにするといったメモリ保護です。前回、実行モードによる保護に関して説明しました。これでは、ユーザーのプロセスが使用する領域に対する保護はできません。

まとめ

▶ メモリ管理の課題

- ▶ ロード時のメモリアドレスの問題
- ▶ フラグメンテーション
- ▶ 未使用領域へのメモリ割当ての問題
- ▶ メモリ容量の限界
- ▶ メモリ保護

▶ オペレーティングシステムにおけるメモリ管理の課題の解決策

- ▶ ここであげた課題を統一的に解決する方式としてページングによる管理が広く利用されている

メモリ管理での課題をまとめます。

これらの課題を統一的に解決する(すなわちメモリの仮想化)ものが、次回説明するページングによる管理です。

教科書との対応

- ▶ 9.3 プロセスのメモリ領域の管理、9.4 メモリ領域の確保・解放機能に関しては、第2回や3回の授業でお話しています

教科書での説明と記載箇所が違うところがありますので、読むときに注意してください。

第5回の課題

メモリの詰め直しについて、教科書 p.129 で
「しかし, 割当て領域のアドレスを要求プログラムに返している場合
はこの方式はとれない。」
とあるが、これはどのようなことか説明せよ

今回の課題です。
クラスウェブのレポートで提出してください。

提出期限は、5/18の午前中とします。

事後学習・事前学習

- ▶ 今回の講義資料に基づいて内容を振り返り、教科書などの該当箇所を読む
- ▶ 教科書第10章に目を通す

今回の講義内容の振り返りと次回の準備をお願いします。