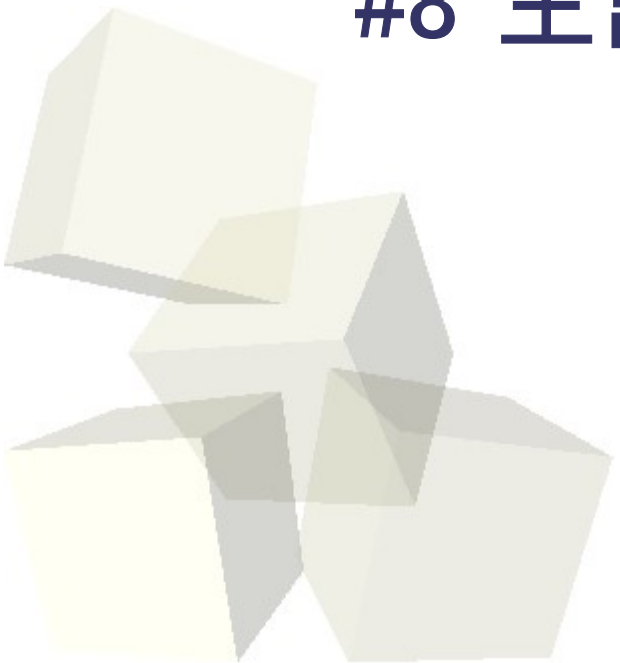
オペレーティングシステム

#8 主記憶管理：主記憶割り当て



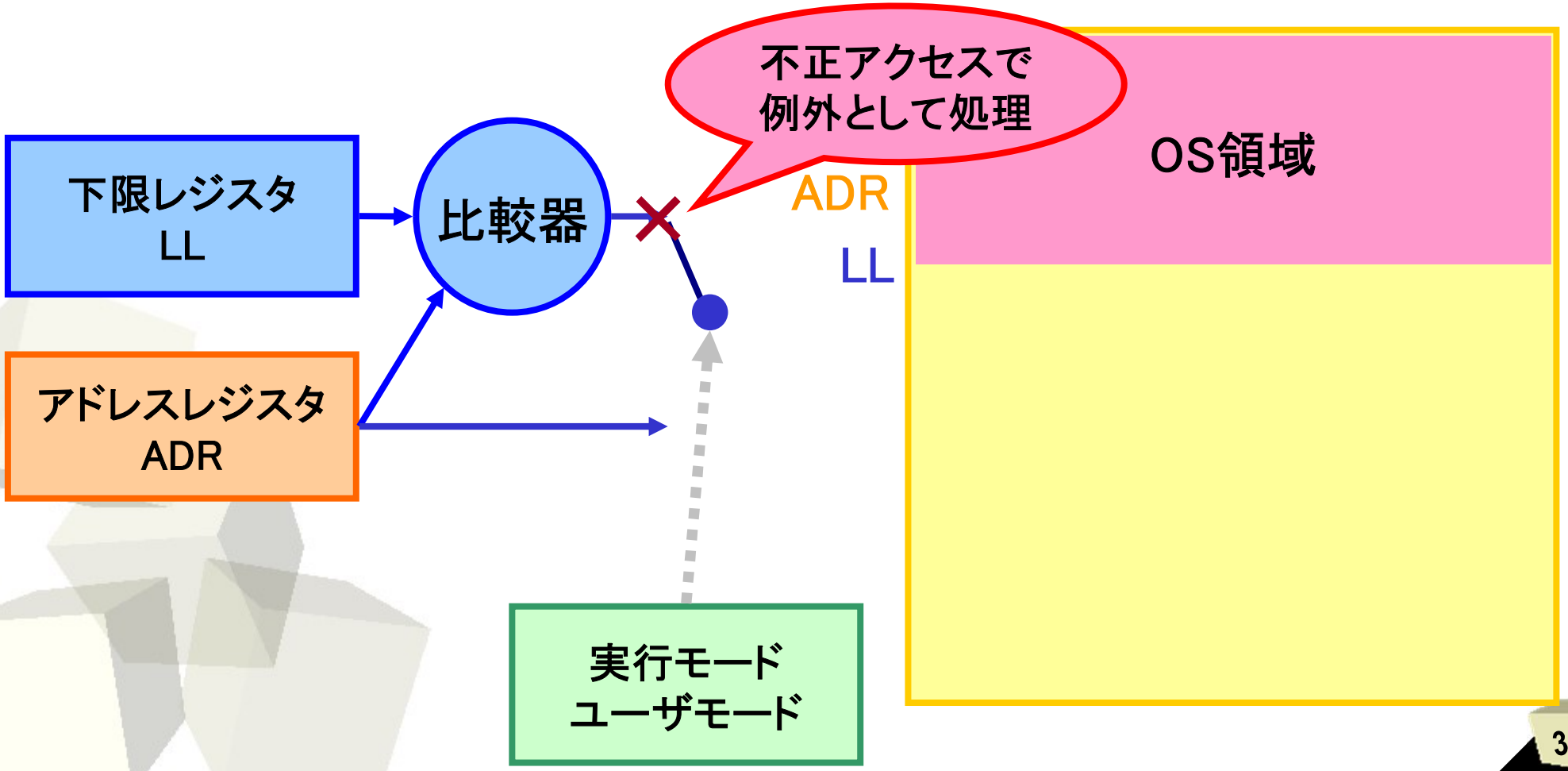
復習

主記憶管理

- ユーザに独立した論理アドレス空間を提供
- 論理アドレス空間の要件
  - 無限大
  - プロセス事に固有
  - プロセス間で共有可能
  - 複数の1次元アドレス（プログラム部，データ部等の分離）

復習

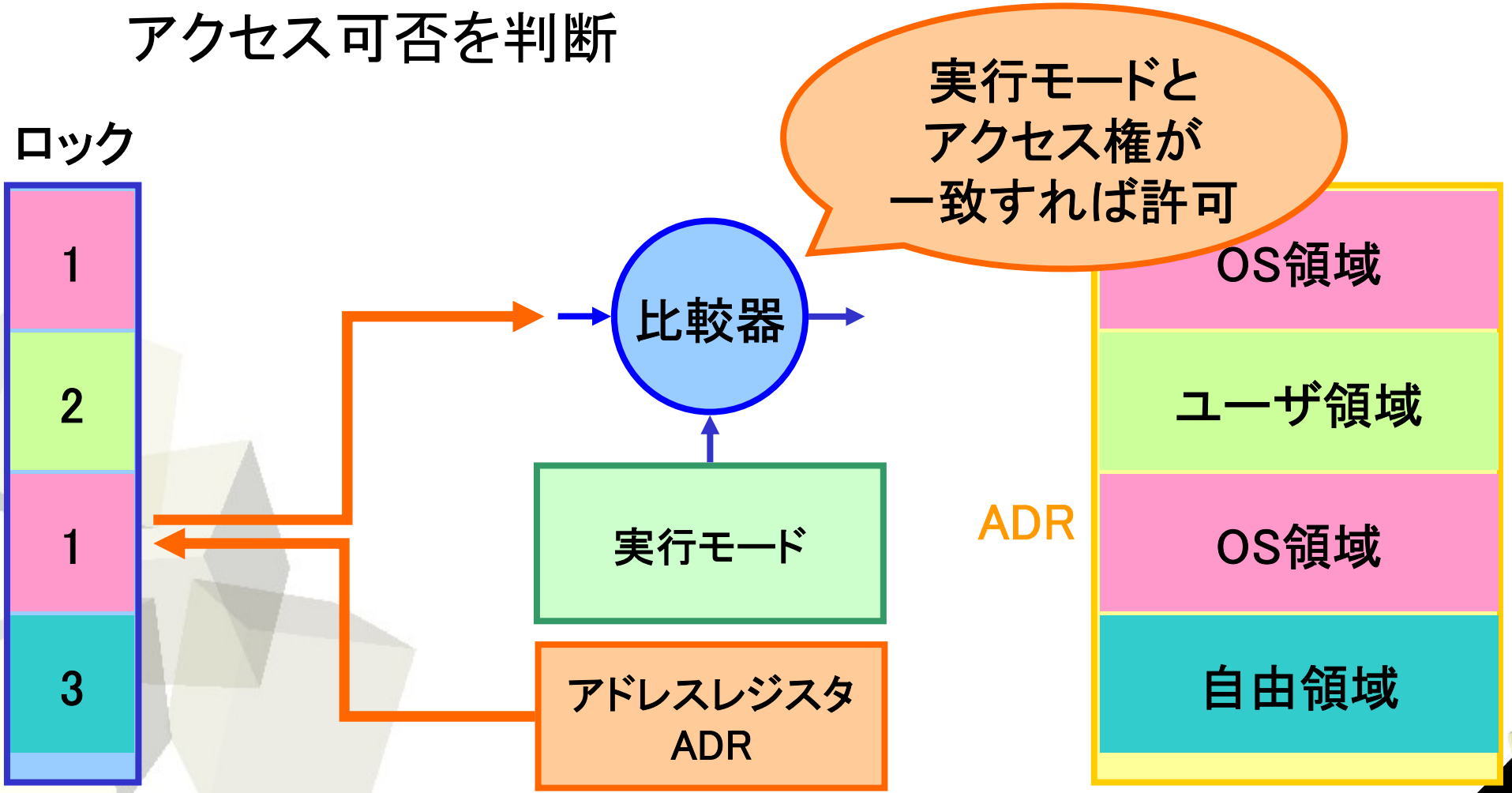
- 下限レジスタ機構
  - ユーザ領域の下限を設定，OS領域とユーザ領域を分離
  - 実行モードに応じてアクセス可否を判断



復習

ロック／キー機構

- アドレスの上位数桁で分割，各領域ごとに権限を設定
- アドレスに応じた権限（ロック値）と実行モードから
  アクセス可否を判断

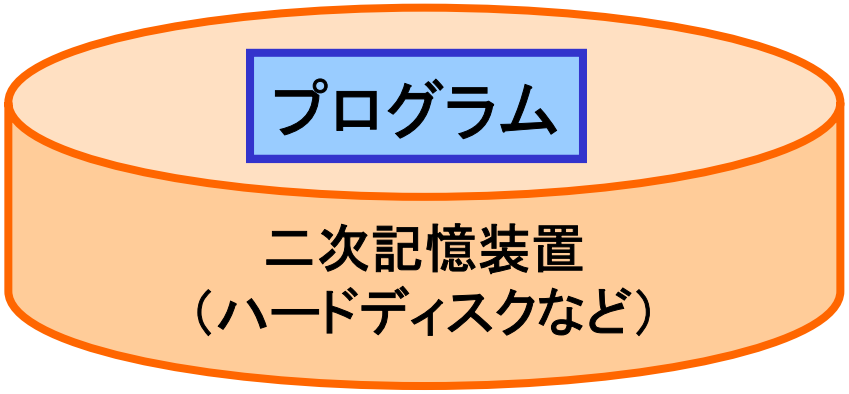


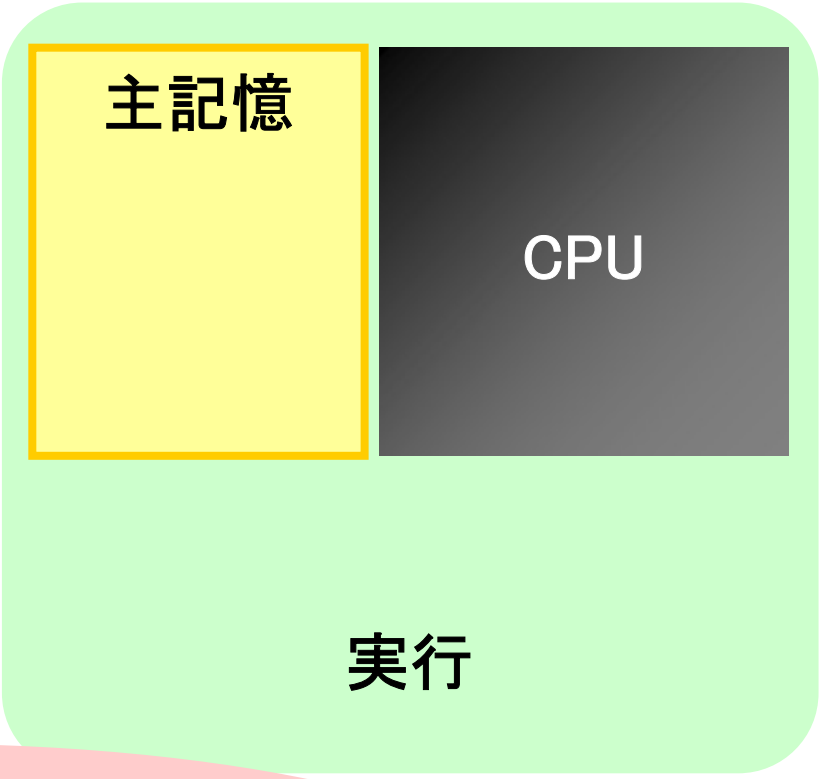
8.1
プログラムの主記憶領域確保

プログラムの実行

実行時

- 二次記憶装置から主記憶へプログラムを転送





インストール

主記憶上に
プログラムを格納できる領域を
確保する必要

主記憶領域の確保

- 主記憶確保は，他プロセスとの協調
  - 主記憶上には他のプロセスも領域を持っている
- 領域要求のタイミング
  - 静的(Static)要求
    - → プログラム実行開始時に必要領域を要求
  - 動的(Dynamic)要求
    - → プログラム実行開始時は最低限の領域を要求
    - → 実行につれて更に必要となった場合はその都度要求

静的主記憶領域確保

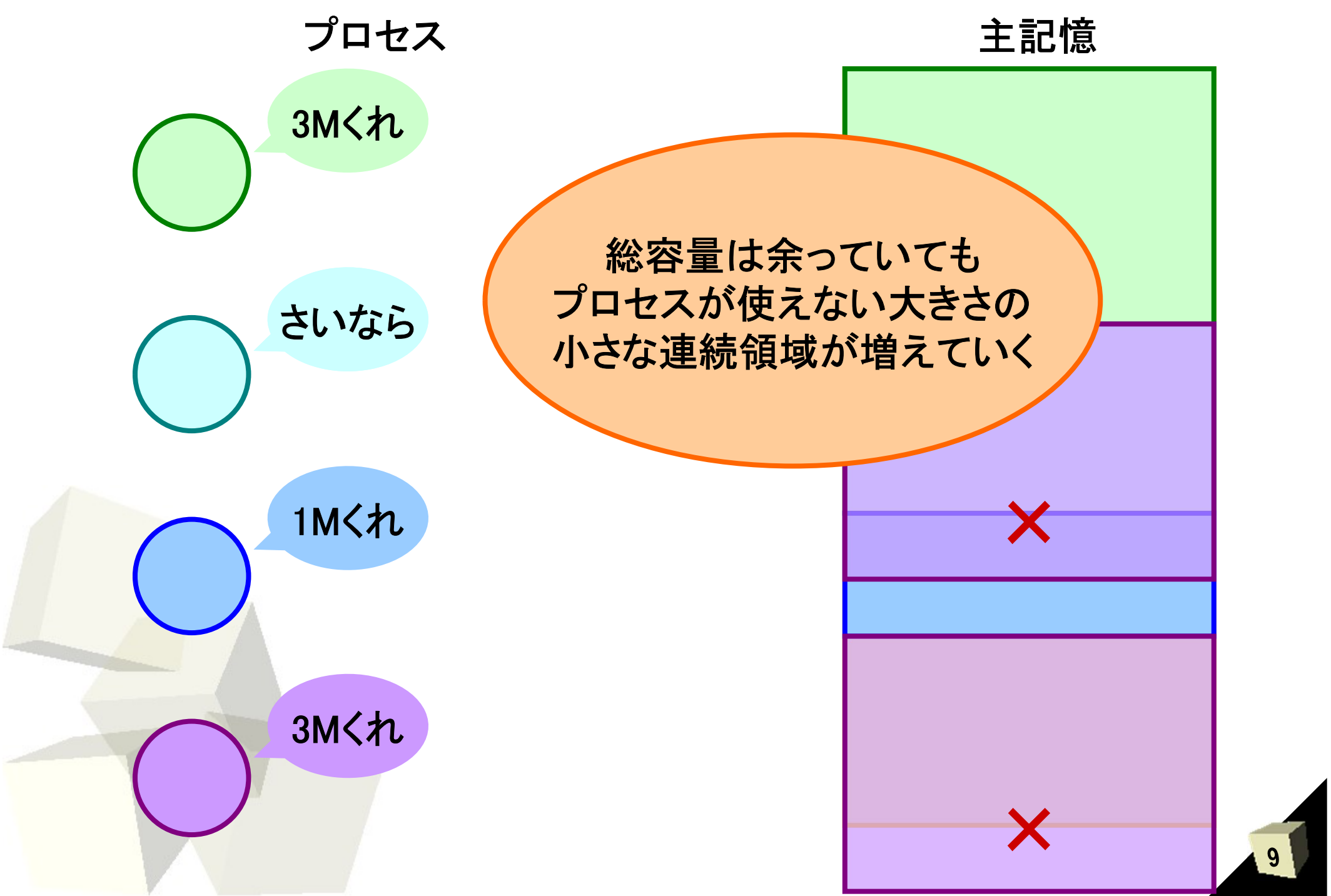
固定区画方式

- プロセスに割り当てる領域の大きさをあらかじめ決めておく(固定区画)
- プロセスから要求があった際,その決められた大きさの領域を割り当てる

可変区画方式

- プロセスは，必要な分だけ領域を要求
- プロセスごとに要求サイズは異なる
- 要求があった分だけ割り当てる

可変区画方式


プロセス
主記憶
3Mくれ
さいなら
1Mくれ
3Mくれ
総容量は余っていても
プロセスが使えない大きさの
小さな連続領域が増えていく

可変区画方式の問題点

空き領域の検索コスト

- 処理が進行するに従い，さまざまな大きさの空き領域が発生
- 新しいプロセスの要求に合う大きさの空き領域を探すコストが増大

空き領域の断片化

- 処理の進行に伴い，小さな空き領域が増加
  空き領域が断片化（フラグメンテーション）
- 主記憶全体の空き容量が十分でも，新しいプロセスに連続した領域を割り当てられない

フラグメンテーション

空き領域の断片化

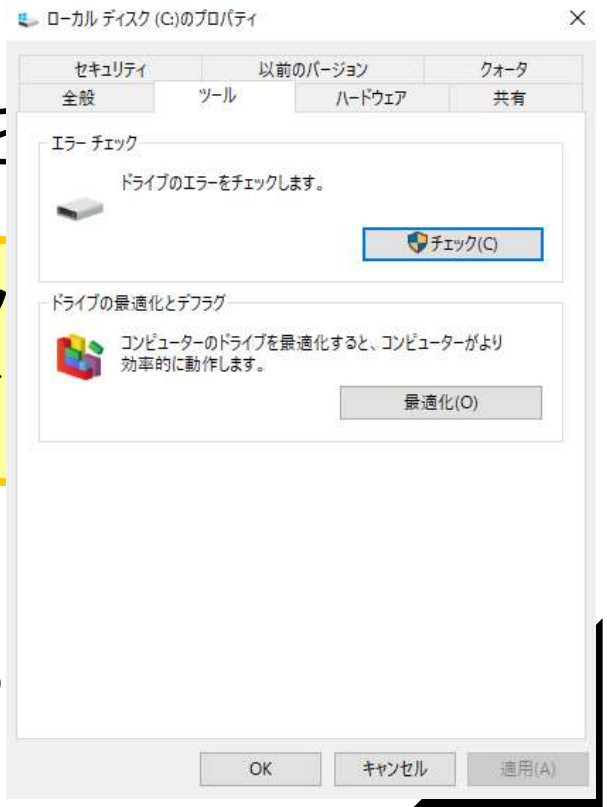
- 処理の進行とともに，小さい使用できない領域が増加
- 統計的には，全領域数の約 1/3(使用中の領域数の半数)が無駄に(注:全容量の1/3 ではない)

解決法:メモリコンパクション

- プロセス実行を停止し，断片化した領域を
  ひとつの連続した領
  まとめる

参考

Windowsの「デフラグ」は，
フラグメンテーションに対し
を行うプログラム



- 動作中のほぼ全てのプロセスを停止する
  必要があり，現実的には困難

1／2ルール

図 8.4 に，主記憶上にロードされた領域の大きさとその位置の例を示す．図に示す通り，主記憶上の領域は，その領域と両隣の領域との関係において 3 種類の関係が存在する．

領域の種別	両隣の状態 状態	当該領域の解放 による空き領域数の増減
a	両隣とも空き領域	-1
b	片隣が空き領域	0
c	両隣とも他のプロセスが使用中	+1

さらに N を使用中のブロック総数 N_a, N_b, N_c をそれぞれ各領域種別の総数，M を空きブロック総数とした場合以下の式が成立する．

$$N = N_A + N_B + N_C \tag{8.1}$$

$$M = \frac{2N_A + N_B + \epsilon}{2} \quad (\text{ただし} \epsilon = 0, 1, 2) \tag{8.2}$$

主記憶

B
空き
B
C
B
空き
A
空き
B

式 (8.2) の M と N_A,N_B,N_C の関係式は，それぞれ N_a, N_b の両端に領域する空き領域を足し，さらに重複分である 2 で割ることにより算出した．また ϵ は主記憶の最上位，最下位のブロックの種別による境界条件であるが，以後 $\epsilon = 0$ と考える．

ここで，オペレーティングシステムが定常状態になったと仮定する．定常状態では，空き領域 M と使用中の領域 N の比率は固定と考えることができる（もし固定でなければ，最終的には空き領域が無限大，もしくは 0 となってしまう）．従って以下の式が成立する．なお P_s,P_e は，プロセスの生成，終了確率を，また P は，新しく発生したプロセスが要求する領域の大きさに，ちょうど合致する空き領域が見つからない確率とする．

$$M \text{ の増加} = N_c P_e \tag{8.3}$$

$$M \text{ の減少} = N_a P_e + N P_s (1 - P) \tag{8.4}$$

M の増加は，種別 c の領域を確保したプロセスの終了によってのみ起こり，M の減少は，種別 a の領域を確保したプロセスの終了と，新しく発生したプロセスの要求する領域が，空き領域と丁度同じになったときに発生する．平衡状態では式 (8.3), (8.4) は等しくなければならないし，またプロセスの生成確率，終了確率も同じでなければならない．従って式 (8.3)=式 (8.4) が成立し，

$$N_c = N_a + N(1 - P) \tag{8.5}$$

が導出できる．式 (8.5) と式 (8.1) の左辺，右辺同士を加算することにより，以下の関係を導き出すことができる．

$$PN = 2N_a + N_b \tag{8.6}$$

さらに，式 (8.6) を式 (8.2) に代入することにより，

$$M = \frac{PN}{2} \quad (\text{ただし} \epsilon \approx 0) \tag{8.7}$$

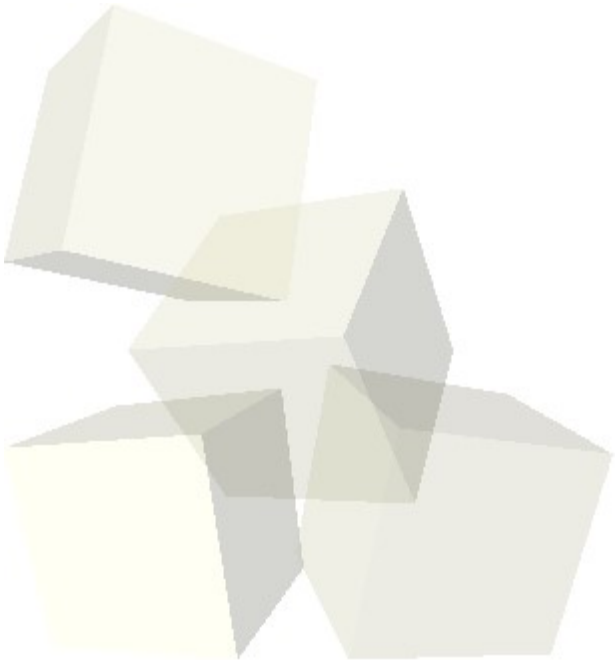
新しく発生したプロセスが要求する領域と同じ空き領域がない確率は，ほぼ 1 と近似することができる．従って最終的に，式 (8.7) は

$$M = \frac{N}{2} \tag{8.8}$$

と変形できる．つまり，統計的にシステムの平衡状態では，空き領域の数は，使用中の領域数の５０％も存在することが導出できる．この空き領域に関する統計的事実は，1/2 ルールと呼ばれる．この主な原因は，P が 1 に近似できる事に起因することに注意して欲しい．

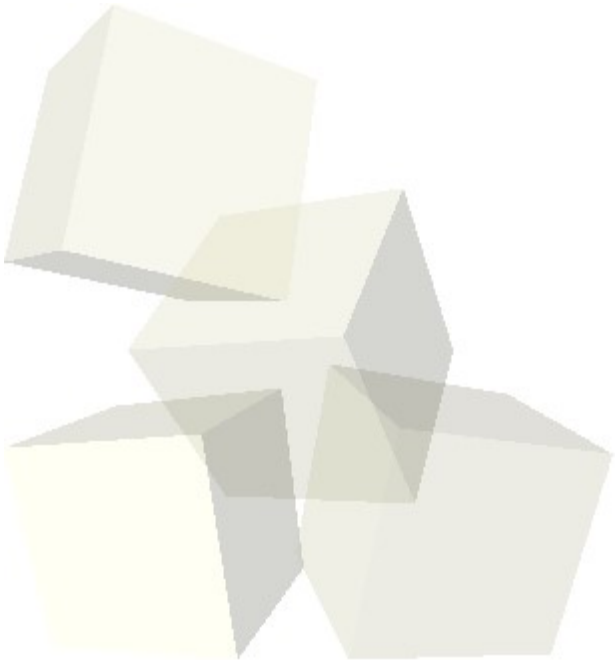
可変区画方式における管理

- 空き領域割り当て
  - ベストフィット
  - ファーストフィット
  - ワーストフィット
- 領域管理
  - リスト方式
  - ビットマップ方式



可変区画方式における管理

- 空き領域割り当て
  - ベストフィット
  - ファーストフィット
  - ワーストフィット
- 領域管理
  - リスト方式
  - ビットマップ方式



割当方式：ベストフィット

- ベストフィット方式
  - 割り当てた残り領域が最も少なくなる空き領域に割当
  - 効率的に見えるが…
  - ×空き容量の探索コストが大きくなる場合がある
  - ×残った領域が小さすぎて他のプロセスが使用できない確率が高い

必要量

割当方式：ワーストフィット

- ワーストフィット方式
  - 割り当てた残り領域が最も大きくなる空き領域に割当
  - 残った領域は，ベストフィット方式より比較的大きくなる
  - ×処理が進むにつれ空き領域の大きさが均一化し，大きい要求に応えられない

必要量

割当方式：ファーストフィット

- ファーストフィット方式
  - 要求された量を確保できる最初に見つかった領域を割り当てる
  - ○探索コスト小
    主記憶領域の全てを調べる必要がない
  - ○アドレス上位に大きい領域が残りやすくなり，大きい要求にも対応しやすい

必要量

空き領域の割り当て方式

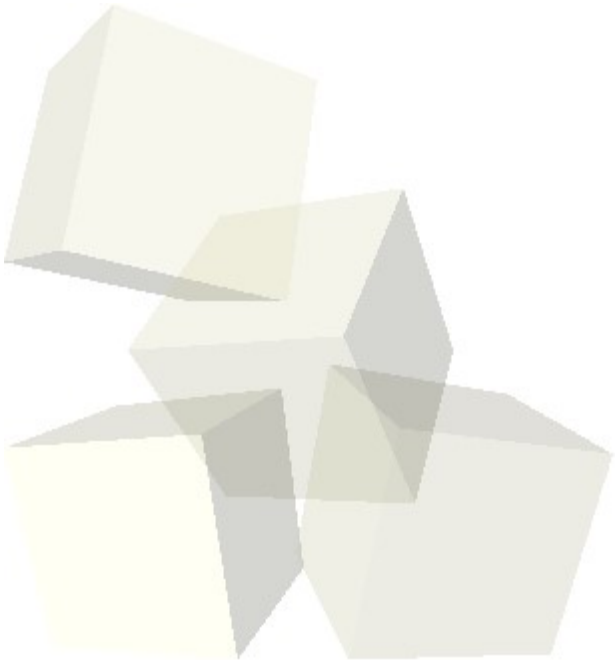
- 割り当て方式
  - ベストフィット
  - ワーストフィット
  - ファーストフィット

- 注意すべき点
  - プロセスは要求と解放を繰り返す
    - → 空き領域の数は常に増減
    - → これを高速管理する必要
  - 空き領域の高速な検索も必要

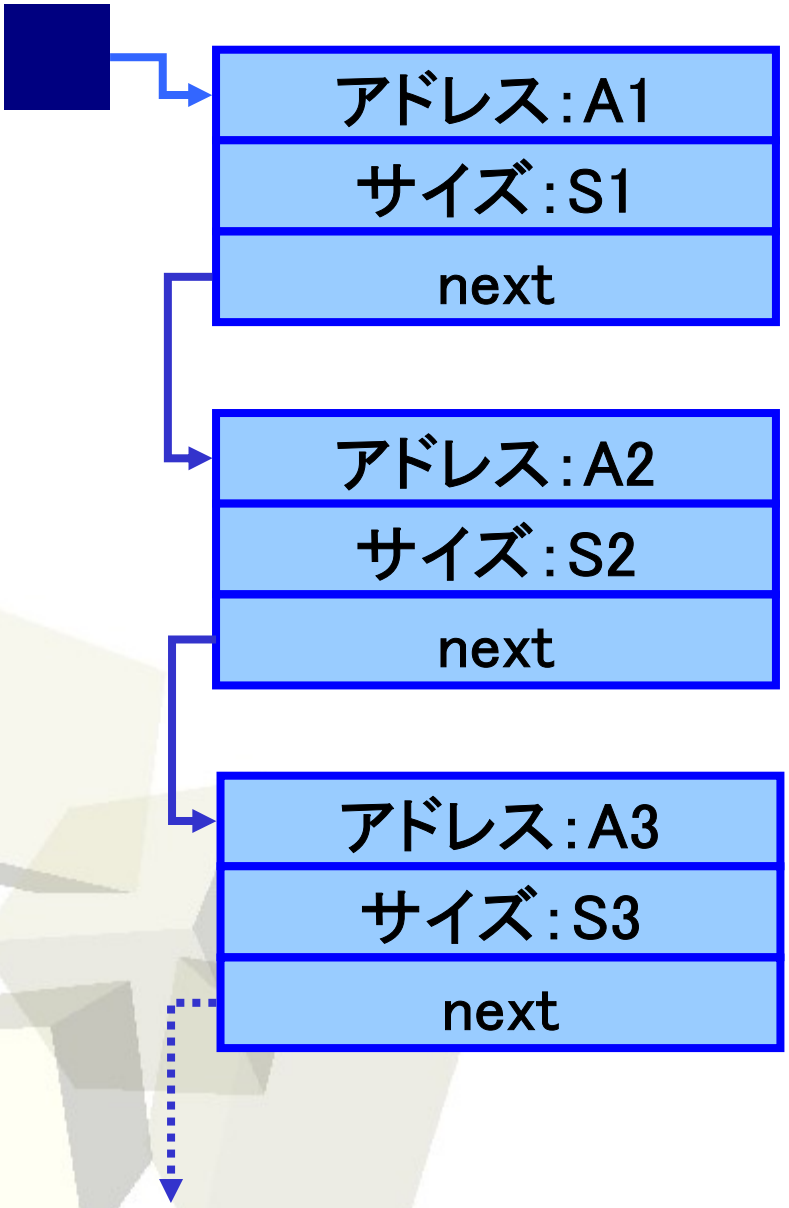
領域管理方式に
工夫が必要

可変区画方式における管理

- 空き領域割り当て
  - ベストフィット
  - ファーストフィット
  - ワーストフィット
- 領域管理
  - リスト方式
  - ビットマップ方式



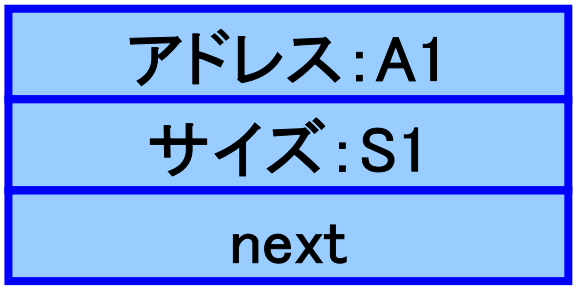
領域管理:リスト方式


アドレス:A1
サイズ:S1
next
アドレス:A2
サイズ:S2
next
アドレス:A3
サイズ:S3
next


主記憶
使用中
A1
S1
空き1
使用中
A2
S2
空き2
使用中
A3
S3
空き3
使用中

領域管理:リスト方式

- 構造体
  - 空き領域の先頭アドレス
  - 空き領域の大きさ
  - 次情報へのポインタ

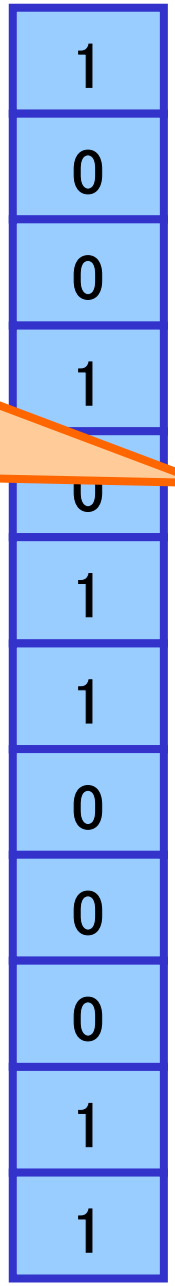


- 上記構造体のリストとして全空き領域を管理
  - アドレス順リストの場合
    - → ファーストフィット方式での検索が高速
  - 大きさ順リストの場合
    - → ベストフィット方式での検索が高速

領域管理:ビットマップ方式

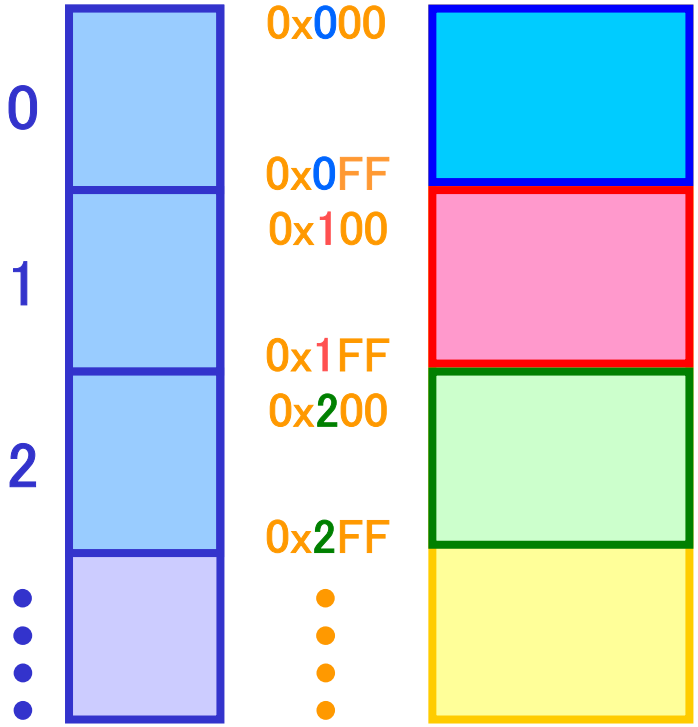
単位ブロックに分割

各ブロックに
対応づけられたフラグで
領域ブロックの状態を
管理



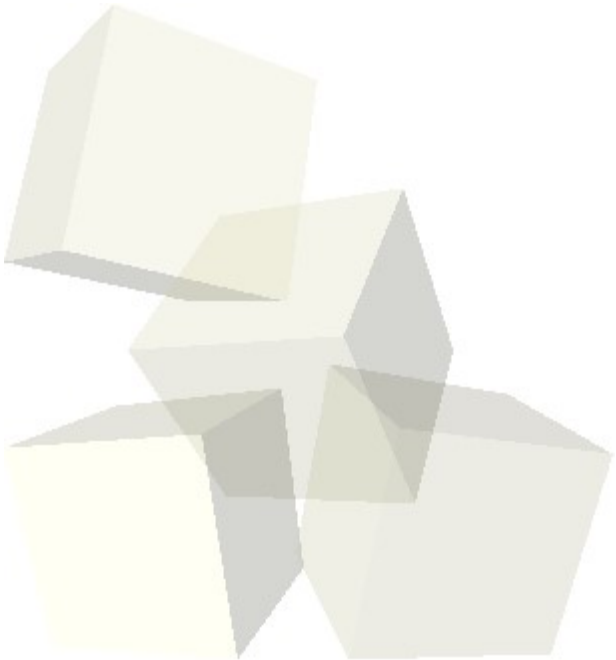
領域管理:ビットマップ方式

- ビットマップ
  - 記憶領域を単位ブロックに分割
    - → 例)アドレスの上位数ビットが
      共通の部分など
  - 各領域に対応するビット
    (フラグ)を用意



- ビット配列により，主記憶全体の空き領域を表現
  - 大きい連続した空き領域を検索する際は
    - → フラグに0が続いている部分を探す
  - 各要素へのアクセスは高速だが，
    空き領域の検索コストは大きくなりがち

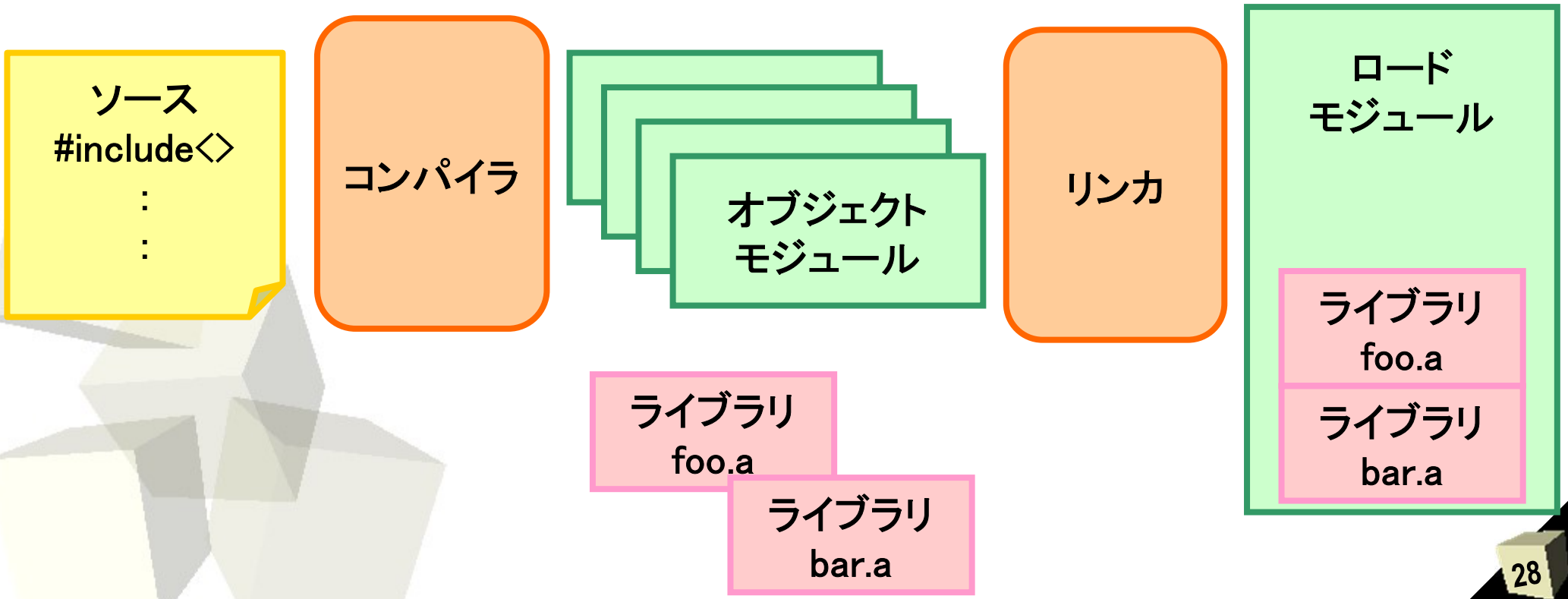
8.2
プログラムのロードと領域の再配置



ロードモジュール

実行可能形式

- 一般にソースプログラムから
  コンパイラ&リンカが生成する
- システム上で実際に実行されるプログラム



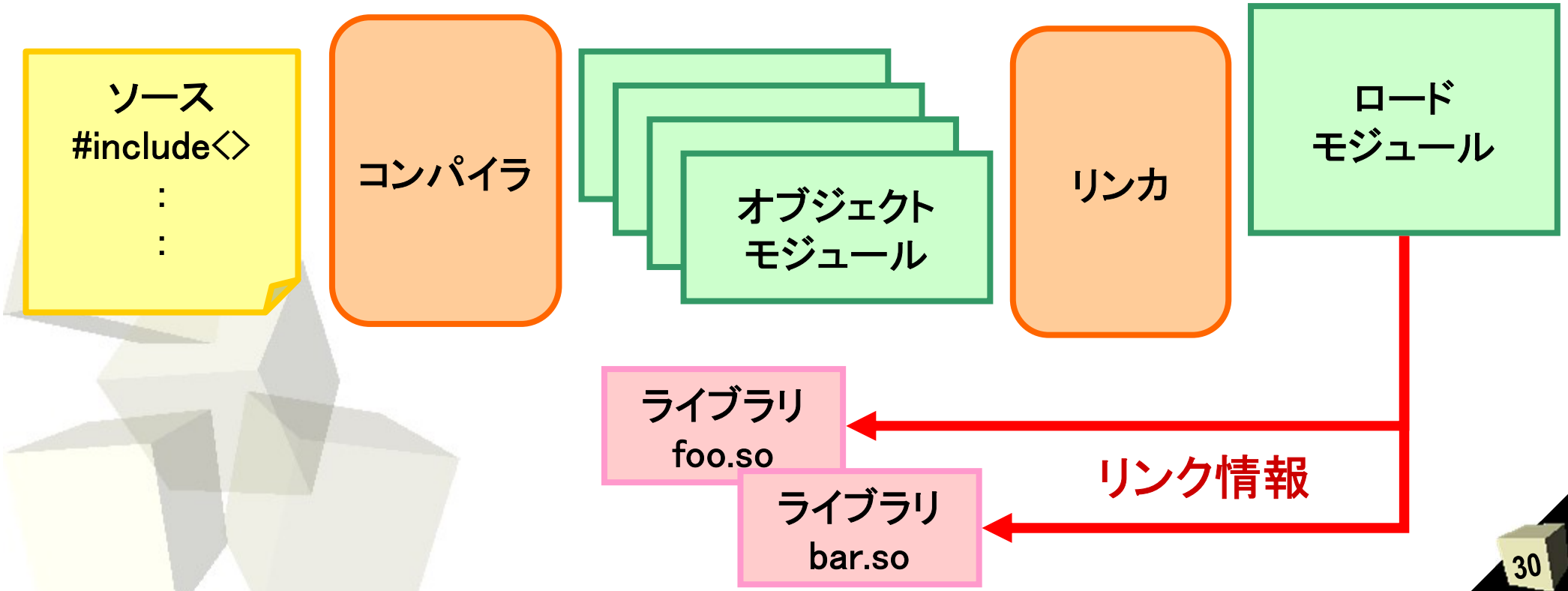
ライブラリ

- ライブラリ
  - 複数のプログラムで
    共通して使用する機能
  - (静的)リンクにより
    各ロードモジュールに
    内包される

  - 主記憶領域の浪費
  - (ディスク領域も)

共有ライブラリ

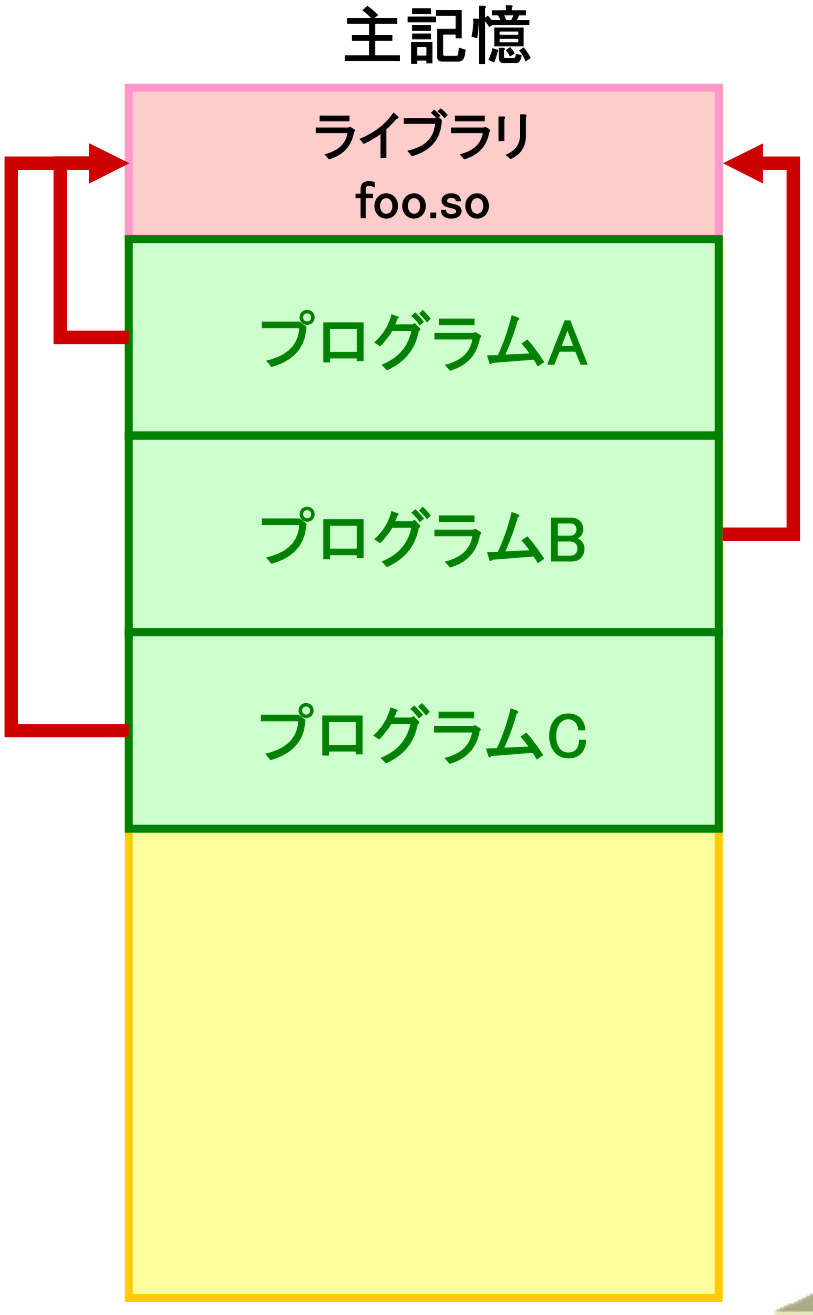
- 複数のプログラムにより共有可能なライブラリ
  - リンカはロードモジュールにライブラリの埋め込みを行わない
  - 共有ライブラリに対するリンク情報のみロードモジュールに書き込む



共有ライブラリ

- 共有(shared)ライブラリ
  - ロードモジュールには実体を内包しない
  - ライブラリへのリンク情報のみをロードモジュールに埋め込む

  - 主記憶における使用領域の削減
  - 実際のリンクは実行時に行う(動的リンク)



ライブラリまとめ

- 静的ライブラリ
  - リンク時にロードモジュールに埋込み（静的リンク）
  - 複数プログラムで使用されるライブラリがある場合，主記憶領域の無駄
- 共有ライブラリ
  - ロードモジュールはリンク情報のみを持ち，実行時にリンク（ダイナミックリンク）
  - 複数プログラムで使用される場合でも，各ライブラリは1つのイメージだけ主記憶上に存在すればよい
  - 主記憶領域（およびディスク領域）の有効活用

共有ライブラリに必要な性質

- リエントラント性
  - プログラム *A* 内の関数 *F* でプリエンプションが発生した場合，他のプログラム *B* は同じその関数 *F* を実行できなければいけない（共有してるから）
  - もちろんプログラム *A* は後に関数 *F* を再開できなければいけない
  - 各呼び出しごとに，作業領域を保存する仕組みが必要
    - → 関数内でグローバル変数をアクセスしない。関数内でstaticな変数を使わないなど、、
- 再配置可能（リロケータブル）
  - 主記憶上のどの位置にロードされた場合でも同じように実行可能であること

リエントラントでないプログラム

~~int s;~~

```
void swap(int *x, int *y)
{
        s = *x;
        *x=*y;
        *y=s;
}

void 割り込みハンドラ()
 {
        int x = 1, y=2;
        swap(&x, &y);
}
```

int s;

割り込みハンドラ実行中に割り込みが起きる可能性があるので、リエントラントである必要あり

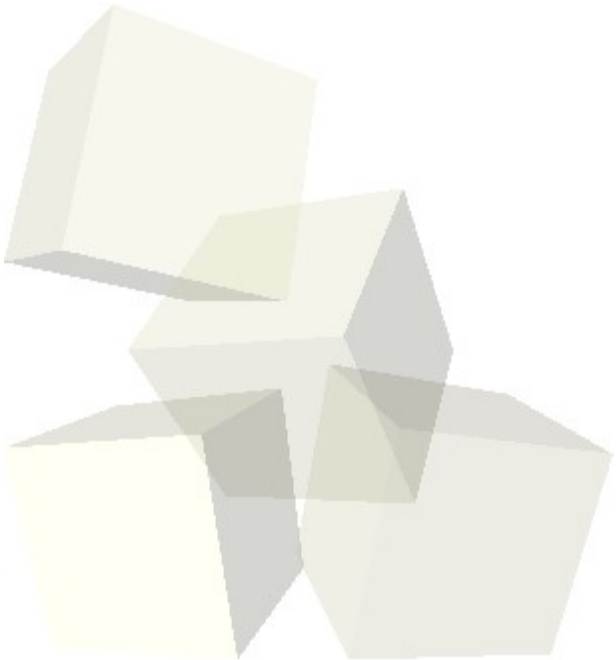
共有ライブラリの現状

- 共有ライブラリは、主記憶の効率的利用という観点からは望ましい

- プログラム（ライブラリ）全てがリエントラント性を満たすためには相当な書き換えが必要

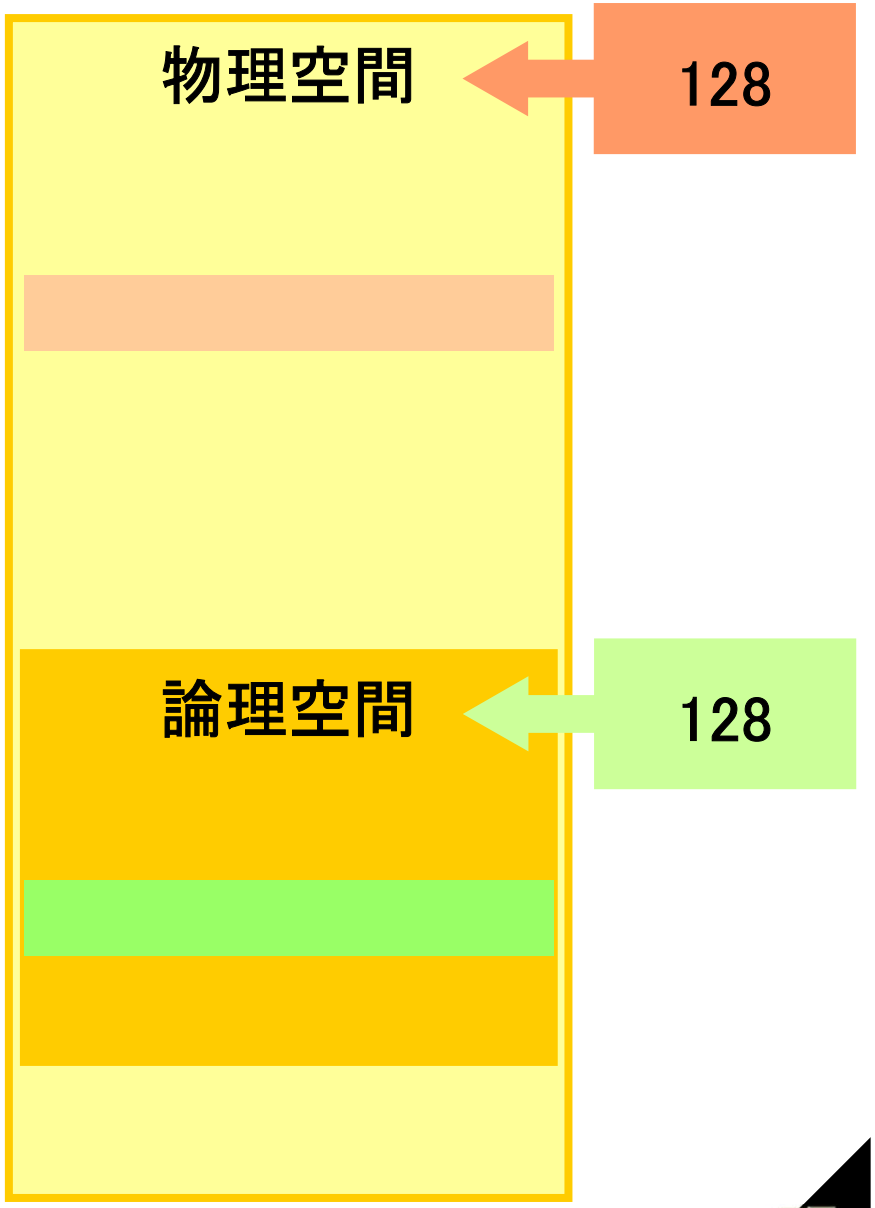
- 現状
  - ダイナミックリンク（単体）は、利用されている
    - → 2次記憶の有効利用
    - → 脆弱性対応時の容易性＝＞ライブラリだけを配布すればよい（例　Windows DLL ）

8.3
オーバレイ



復習：アドレス空間

- **物理アドレス空間**
  - 「128番地」は，メモリ上の
    まさに「128番地」を指す
- **論理アドレス空間**
  - 「128番地」は，メモリ上の
    どこかの番地（物理アドレス）を指す
    が，実際の「128番地」
    とは限らない。
    プロセスによって異なる場合も。
  - MMU（Memory Management Unit）
    が，
    論理→物理アドレスの変換を行う



復習：理想的な論理アドレス空間

- **大きさが無制限**
  - プロセスは主記憶の空き容量を考慮する必要なし
  - プログラムの簡単化，バグの可能性減少
- **プロセスごとに固有**
  - 他のプロセスからのアクセスに対し保護
- **プログラム部，データ部，スタック部など分離**
  - 用途ごとに空間を分けることで，
    自プロセス内での不正アクセスの可能性を低減
- **必要時にはプロセス間で共有も可能**
  - 並列動作するプロセス間で共有し，
    高速な通信機構として使用

大きい論理アドレス空間

```
main(){
        funcA();
}

funcA(){
        funcB();
}

funcB(){
        funcC();
}
        :
        :
```

（論理空間）

```
main:
        :
        ret;
funcA:
        :
        ret;
funcB:
        :
        ret;
funcC:
        :
        ret;

        :
        :
```

（物理空間）空き領域

大きい論理アドレス空間

- 論理アドレス空間 > 物理アドレス空間
  - そのままでは実現不可能
  - 「時間的に」共有
    - → CPUの場合，
      1つしかないリソースをどうやって無限に見せたか?
  - 論理空間のうち，まさに今必要である部分だけを物理空間にマッピング
  - 必要な箇所が変わるたびにマッピングを変更すればよい

オーバレイ

オーバレイ

- 指定した時刻に，アプリケーションのどの部分が主記憶上に存在すべきかをプログラマが指定するしくみ
- 前に実行されたコード（親関数など）のうち，さしあたり必要のなくなったコードがロードされている主記憶領域に対して，新しく必要となったコードを上書き（overlay）できる

欠点

- 非常に複雑でエラーを起こしやすい
- プログラマの負担が増大

次回以降の「仮想記憶」で解決

軽量に仮想記憶を実装できるオーバーレイはIoTなどの分野で今後も重要な位置を占める

まとめ

- 主記憶管理
  - プログラムの実行には，ディスクから主記憶へのプログラムのロードが必要
  - マルチプログラミング環境では複数プロセスが並行して主記憶を使用
  - 主記憶上でプログラムおよびデータを格納するための領域を管理する必要
- 記憶領域確保の方式
  - 固定区画方式
  - 可変区画方式

まとめ

- 可変区画方式
  - 空き領域の割り当て方式
    - → ベストフィット
    - → ワーストフィット
    - → ファーストフィット
  - 割当を適切に行わないと，プロセスが使用できない断片的な空き領域が多く発生（フラグメンテーション）
  - 空き領域の管理方式
    - → リスト方式
    - → ビットマップ方式

まとめ

- 静的ライブラリ
  - リンク時にロードモジュールに埋込み（静的リンク）
  - 複数プログラムで使用されるライブラリがある場合，主記憶領域の無駄
- 共有ライブラリ
  - ロードモジュールはリンク情報のみを持ち，実行時にリンク（動的リンク）
  - 複数プログラムで使用される場合でも，各ライブラリは1つのイメージだけ主記憶上に存在すればよい
  - 主記憶領域（およびディスク領域）の有効活用