Operating Systems

メモリ(主記憶)管理

2020-10

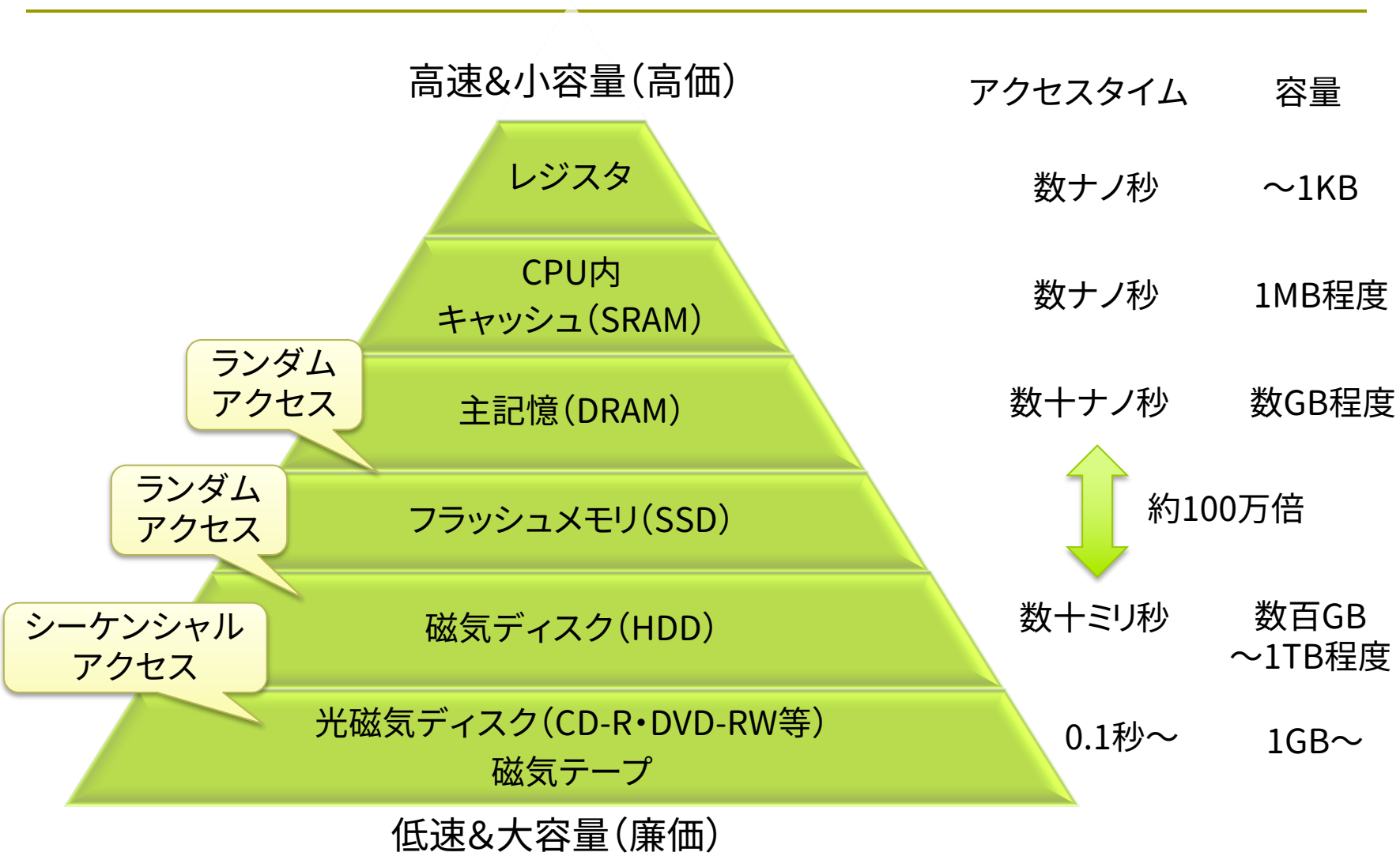
OSの代表的機能（復習）

- プロセス管理
  - 実行中のプログラムの管理
  - コンピュータの実行状態の制御
- メモリ管理
  - 主記憶の管理
  - プログラム実行のためのメモリの管理
- ファイル管理
  - 補助記憶（ストレージ）の管理
  - HDDやSSDに保存されたデータの管理

記憶管理

- その他
  - 通信・ネットワーク，セキュリティ，ユーザ・課金管理など…

記憶階層

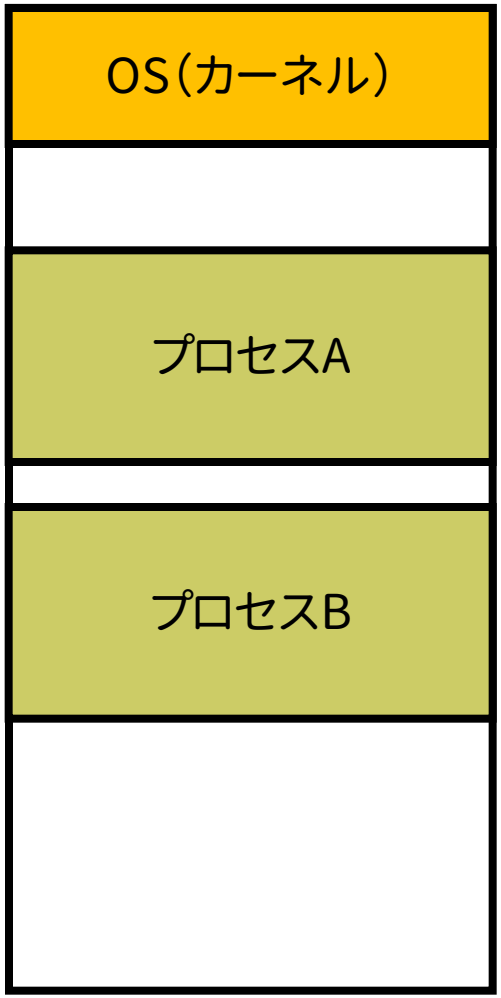

高速&小容量(高価)
レジスタ
CPU内
キャッシュ(SRAM)
主記憶(DRAM)
フラッシュメモリ(SSD)
磁気ディスク(HDD)
光磁気ディスク(CD-R・DVD-RW等)
磁気テープ
低速&大容量(廉価)
ランダム
アクセス
ランダム
アクセス
シーケンシャル
アクセス
アクセスタイム
容量
数ナノ秒
~1KB
数ナノ秒
1MB程度
数十ナノ秒
数GB程度
約100万倍
数十ミリ秒
数百GB
~1TB程度
0.1秒~
1GB~

記憶装置の役割

- レジスタ
  - CPUが演算処理をするための一時的な記憶領域
  - データだけでなくプログラムの実行位置（アドレス）も保持している
- CPU内キャッシュ
  - 主記憶のコピーを一時保持しておく非常に高速なメモリ
  - CPUと主記憶との間のデータ転送を高速化する
- 主記憶（メインメモリ）
  - アドレス（番地）を指定して記憶場所に直接アクセスできる記憶装置
  - 実行中のプログラムやデータを保持する
- 補助記憶（2次記憶，ストレージ）
  - データやプログラムを長期間保存しておくための記憶媒体
  - 一般的には，名前をつけた「ファイル」として情報を管理する

メモリ管理

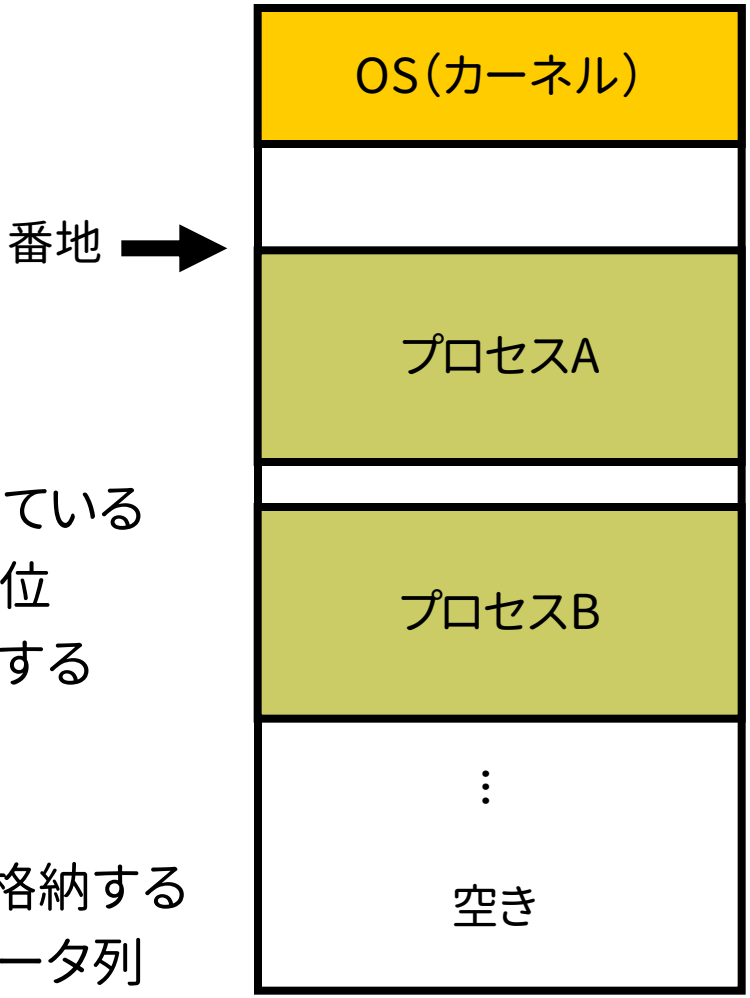
- メインメモリ(主記憶)
  - 半導体メモリ(LSI)で構成される
  - RAM: 読み書き可能なメモリ
  - ROM: 読み出し専用メモリ
- メモリ管理とは?
  - OSによる主記憶の管理
  - なるべく無駄なくメモリを使うのが目的
- メモリ管理の基本
  - プロセス起動時に空き領域を割り当てる
  - プロセス終了時にその領域を回収する
  - 適切なアクセス管理をしてメモリを保護する



メモリマップ

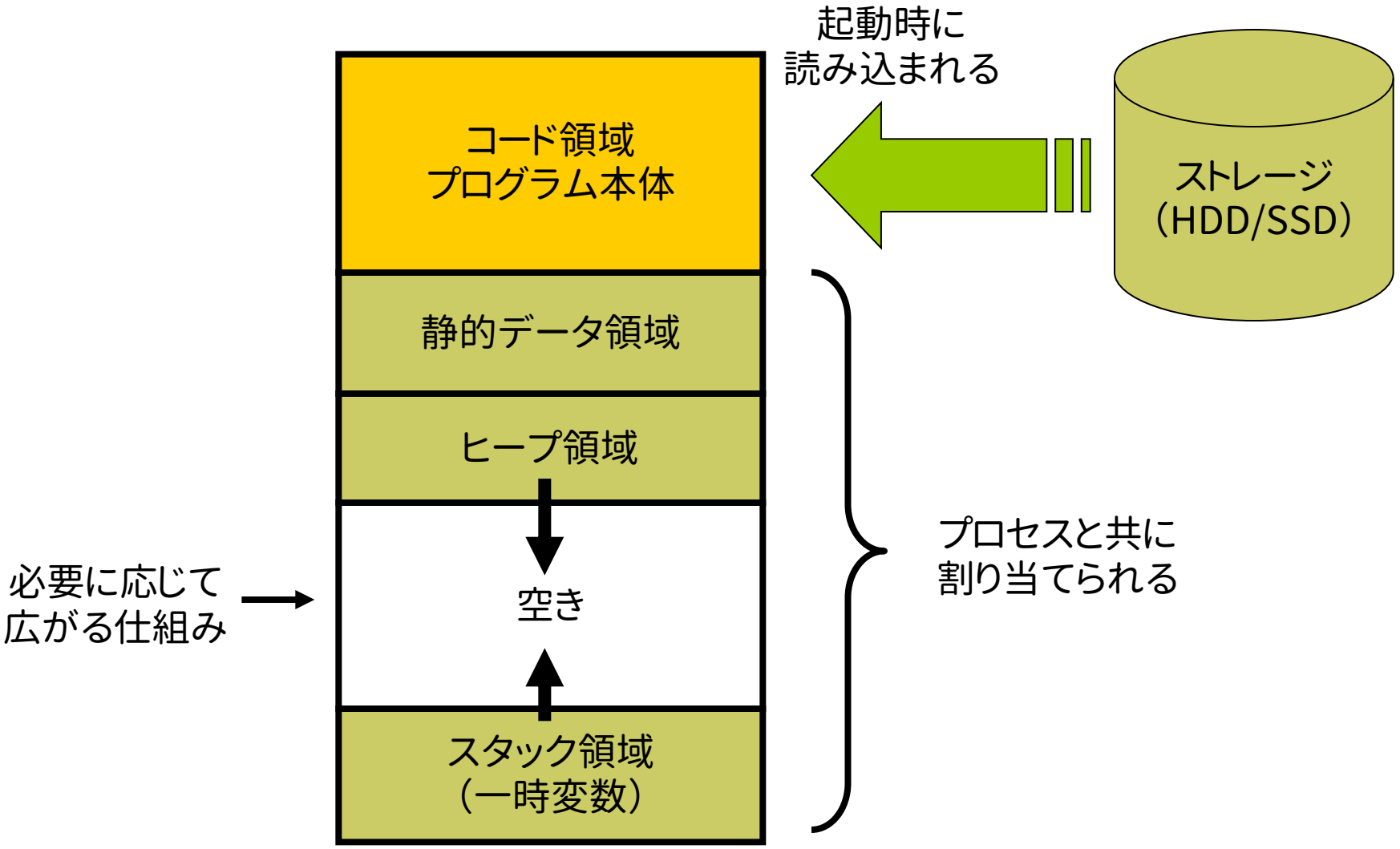
メモリ(主記憶)のしくみ

- ランダムアクセスが可能
  - 直接場所(番地)を指定して,
    どんな順番でも読み書きできる
  - ⇔ シーケンシャルアクセス
- アドレス(番地)
  - メモリの中は通し番号の番地がついている
  - 現在のPCでは,アドレスは1バイト単位
  - CPUはデータを格納アドレスで管理する
- ノイマン型アーキテクチャ
  - プログラムもデータと同じ主記憶に格納する
  - プログラムの形式は,「機械語」のデータ列
  - 機械語の例: 10110000 01100001



メモリマップ

プロセスの内部


起動時に
読み込まれる
ストレージ
（HDD/SSD）
コード領域
プログラム本体
静的データ領域
ヒープ領域
空き
スタック領域
（一時変数）
必要に応じて
広がる仕組み
プロセスと共に
割り当てられる

プロセス空間

- コード領域
  - プログラム本体（機械語の命令列）
  - 共有ライブラリは別領域が割り当てられることもある
- 静的データ領域
  - グローバル変数など，プロセス実行中にサイズの変更がないメモリ
- スタック領域
  - 関数の引数，（一時）変数，戻り先アドレスなどが保持される
  - 関数の開始時にスタック内に“枠”が確保され，戻るときに破棄される
- ヒープ領域
  - プロセスが必要に応じてデータ用の領域を動的に確保できるメモリ
  - C言語では，malloc/freeで利用できる（C++等ではnewで確保）

変数や関数のアドレス(課題10a)

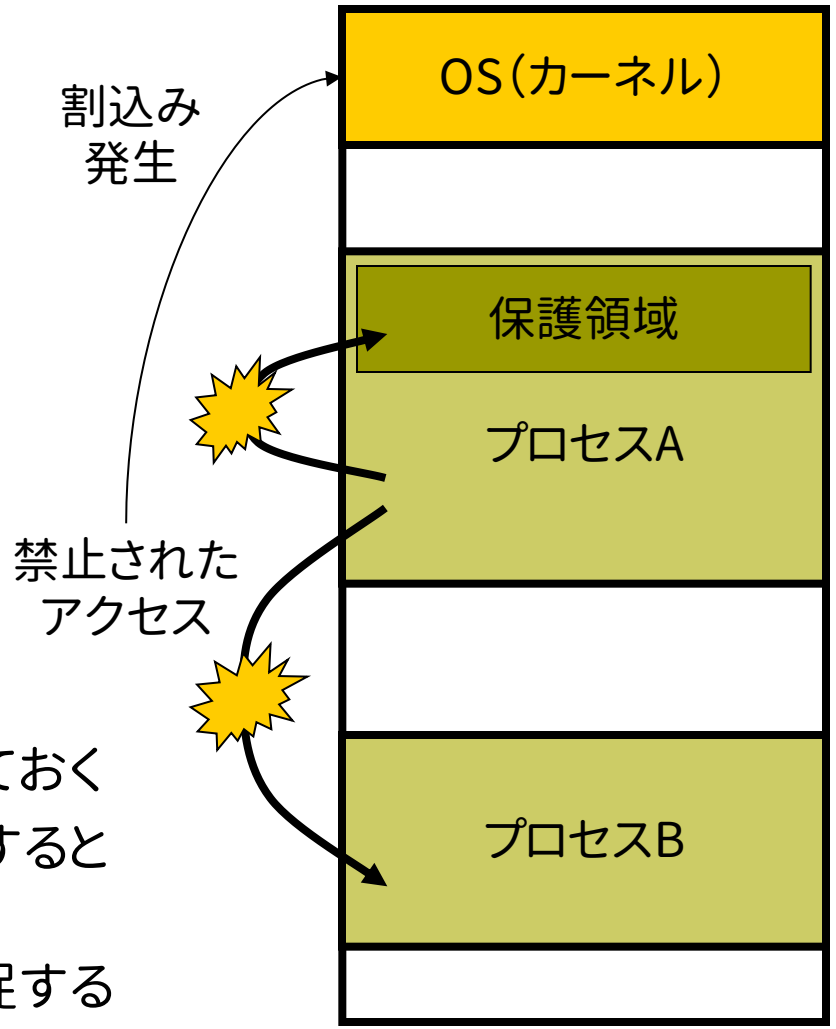
```
#include <stdio.h>
#include <stdlib.h>

int g = 1, h = 2;

int func(int u, int v)
{
    int w;

    w = u + v + g + h;

    printf("uの番地: %p¥n", &u);
    printf("vの番地: %p¥n", &v);
    printf("wの番地: %p¥n", &w);
    printf("gの番地: %p¥n", &g);
    printf("hの番地: %p¥n", &h);
    return w;
}
```

```
int main(void)
{
    int a, b, *p;

    p = (int *) malloc(sizeof(int));
    scanf("%d %d", &a, p);
    b = func(a, *p);

    printf("aの番地: %p¥n", &a);
    printf("bの番地: %p¥n", &b);
    printf("pの番地: %p¥n", &p);
    printf("malloc領域の番地: %p¥n", p);
    printf("mainの番地: %p¥n", &main);
    printf("funcの番地: %p¥n", &func);
    printf("printfの番地: %p¥n", &printf);
    printf("scanfの番地: %p¥n", &scanf);

    scanf("%d", &a); return 0;
}
```

Visual Studio では scanf_s

メモリの保護

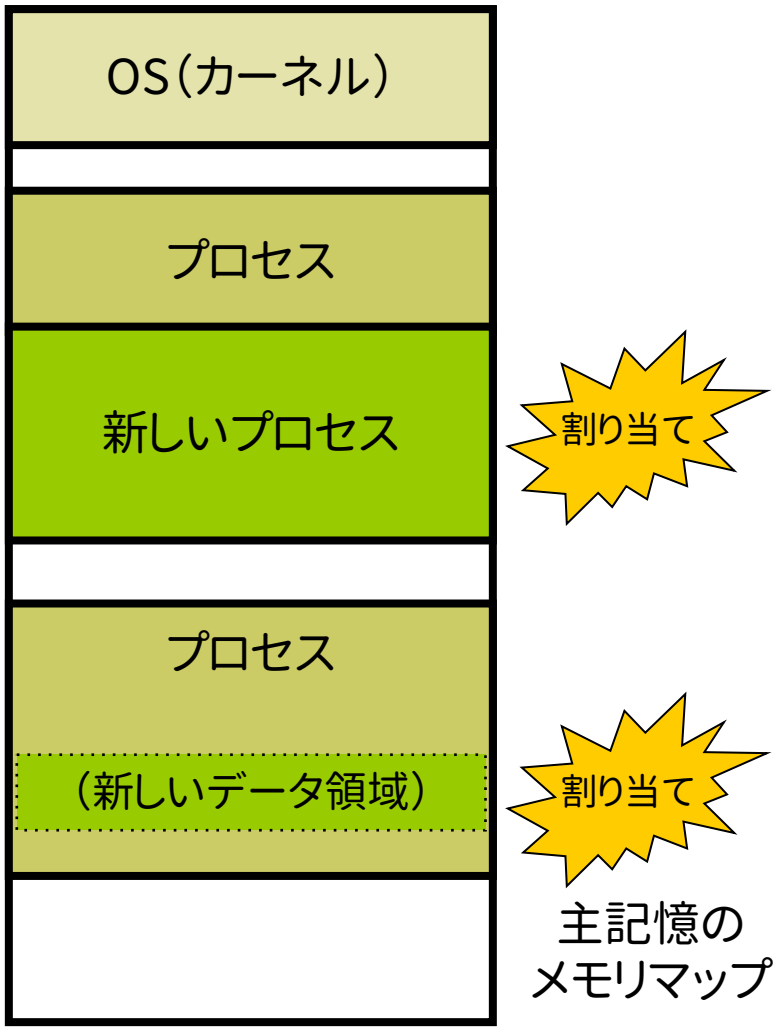
- 記憶保護の目的
  - 実行中のプロセスを保護し，プロセス同士を独立させる
  - 同じプロセス内でバグ等による不正なアクセスを検出する
  - ⇒ 安定性やセキュリティの向上
  - 組み込み用OSではメモリの保護機能がない場合もある
- アクセス違反の検出
  - CPUの“境界レジスタ”を設定しておく
  - プロセスが禁止領域をアクセスすると“アドレスエラー割込み”が起きる
  - カーネルの割込みハンドラで捕捉する



メモリの割り当て

- OSによる大域的な割り当て
  - 主記憶の空き領域に，新しいプロセス（※）を配置する
  - マルチタスクでは，主記憶に複数のプロセスが共存する

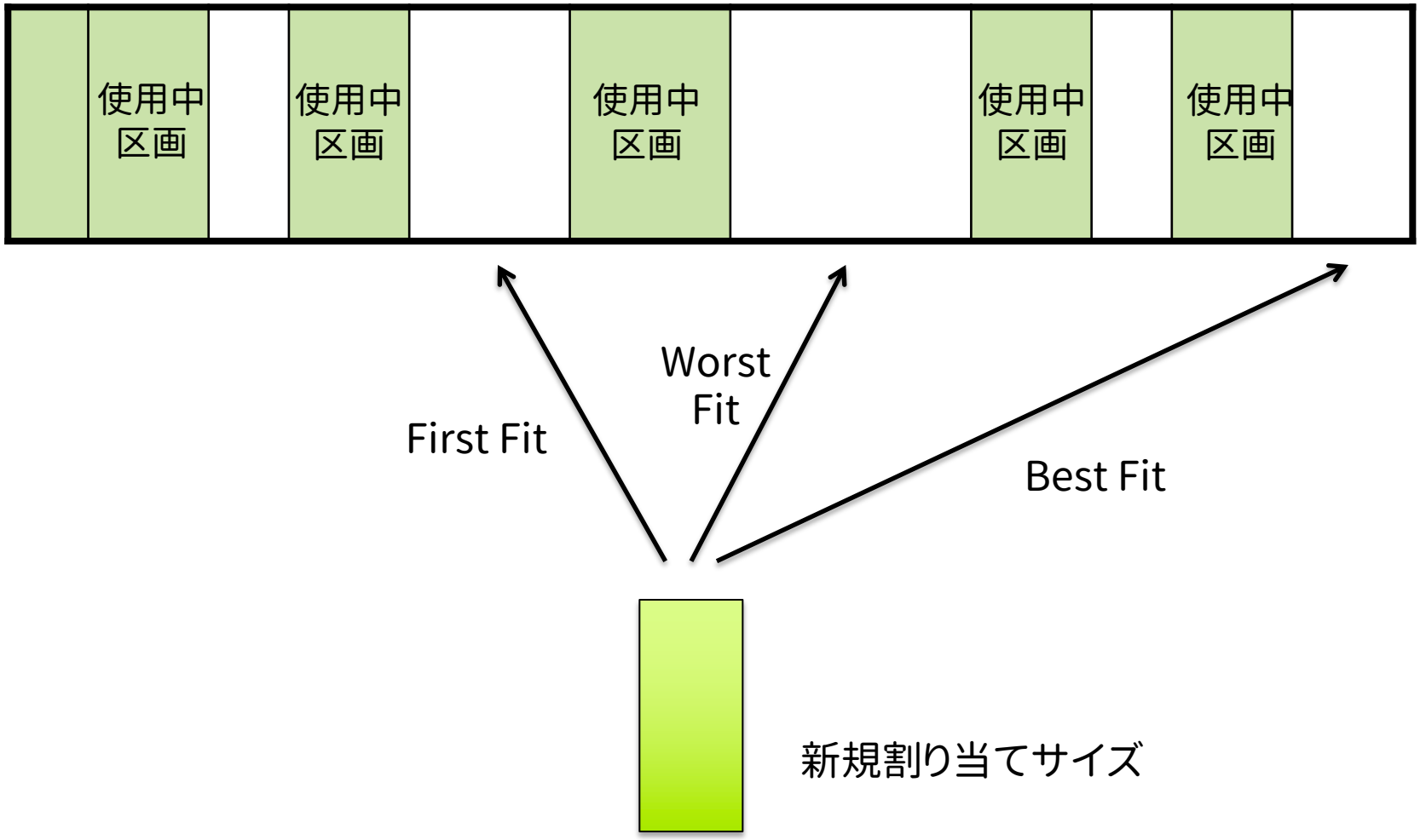
  ※ 共有メモリにも用いられる
- プロセス空間内の割り当て
  - 各プロセスがヒープ領域に，データの領域を割り当てる
  - C言語なら，malloc & free
  - 各プロセスが管理している
- 基本的な考え方は同じ！



メモリ割り当てアルゴリズム

- 先頭一致(First Fit)
  - メモリの空き領域を先頭から探索する
  - 最初に見つけた十分な大きさの領域から,必要な大きさを切り取って割り付ける
- 最良一致(Best Fit)
  - 割り付けたい領域が入る最も小さな空き領域から区画を切り出す
  - 最もよさそうだが,細かい断片化が生じるので効率が悪いことが多い
  - 決まったサイズのデータばかりを割り当てるプロセス内では有効
- 最悪一致(Worst Fit)
  - 最良一致の逆ですべての空き領域の中で最大のものから割り付ける
  - 均等化されて大きな空き領域がなくなってしまう欠点がある

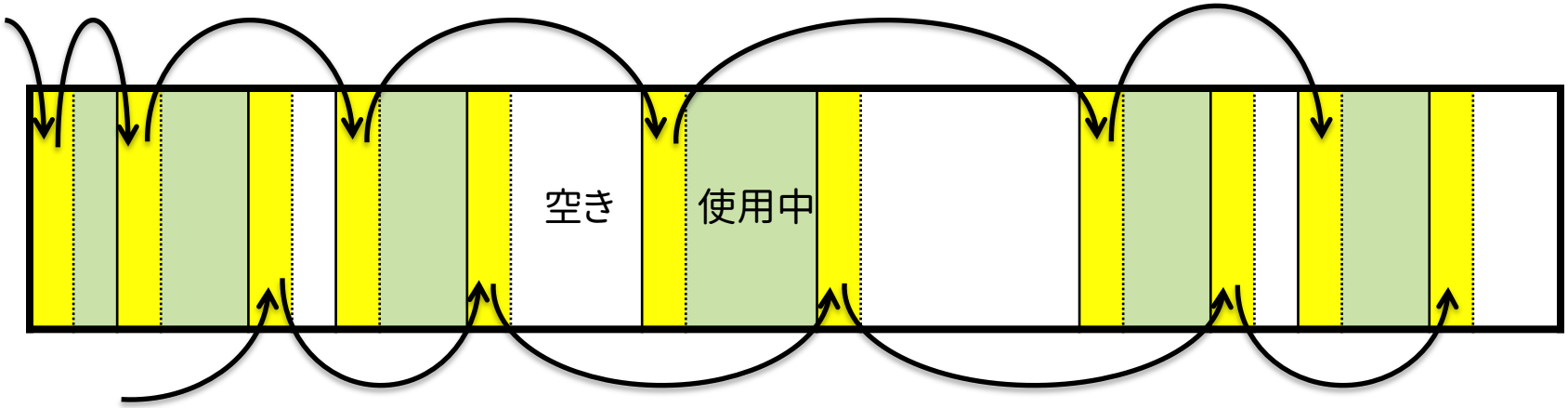
メモリ割り当てアルゴリズム



メモリ領域の管理

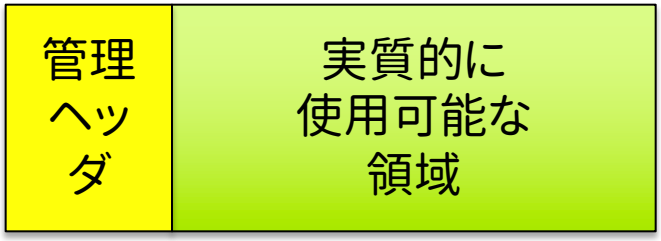
- メモリ使用状況の管理
  - リスト構造や木構造でメモリ領域を管理する

使用中メモリリスト



空きメモリリスト

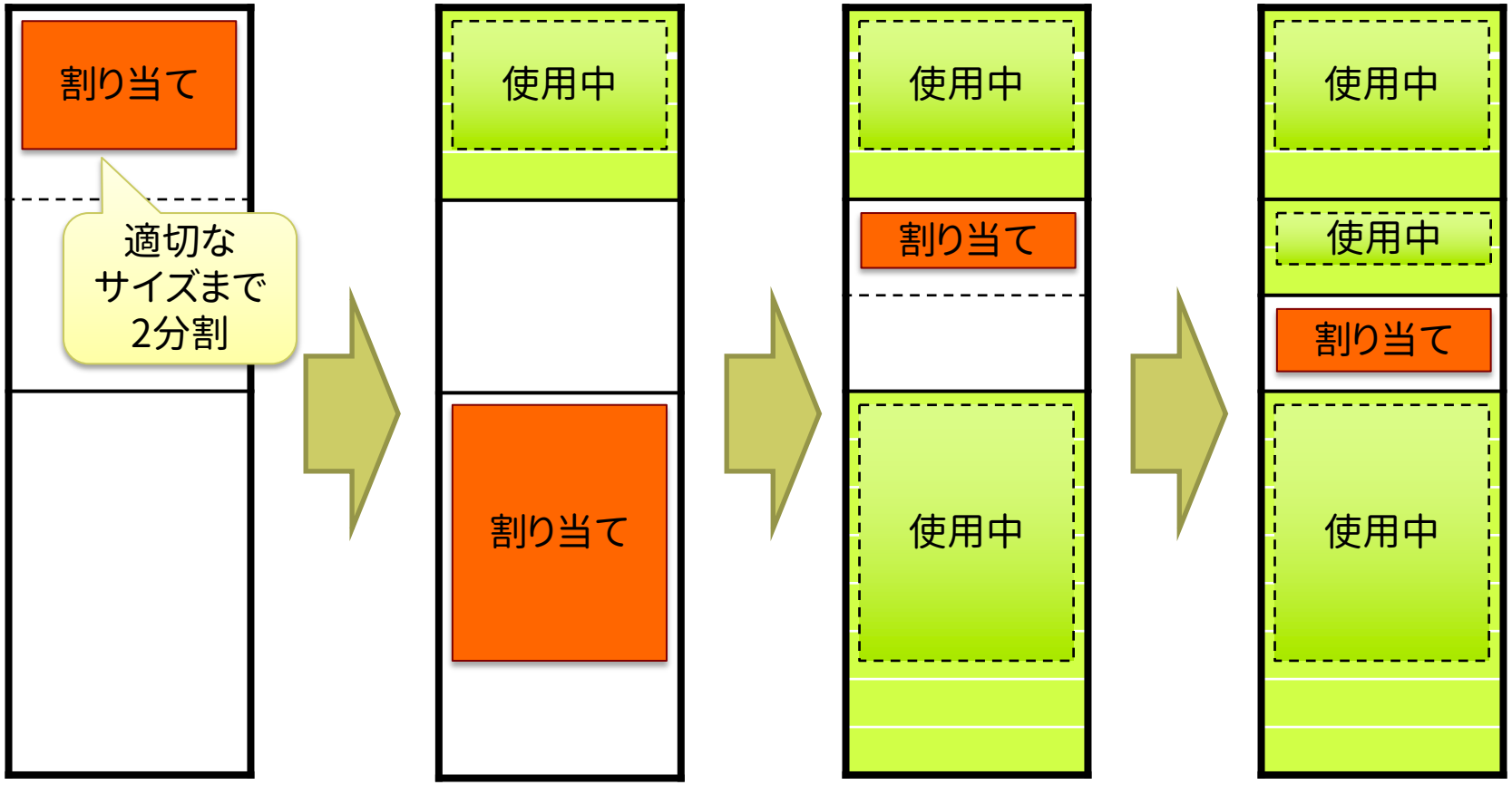
メモリ区画に管理のためのヘッダを付ける



メモリの断片化

- “断片化”の発生
  - メモリの割り付けと回収を繰り返すと，メモリ領域が分断される
  - 時間とともに，空きメモリ領域が断片化していく
  - 断片化が進むと，連続な空き領域が不足し，メモリの総量は十分でも新たに必要な量のメモリを割り当てられなくなってしまう
- 断片化の対策
  - メモリが断片化しづらいアルゴリズムを用いる ⇒ バディシステム
  - ときどき使用メモリをコンパクションし，大きな空き領域を作る
- メモリのコンパクション
  - 断片化した空き領域を，いくつかの大きな領域にまとめる処理
  - そのために，使用中のメモリ領域を中身ごと移動して再配置する
  - プロセスを停止せずに実現するには，“動的再配置”への対応が必要

バディシステム



メモリを再帰的に2分割してペアで管理し，断片化を軽減する

ガーベジコレクション（ゴミ集め）

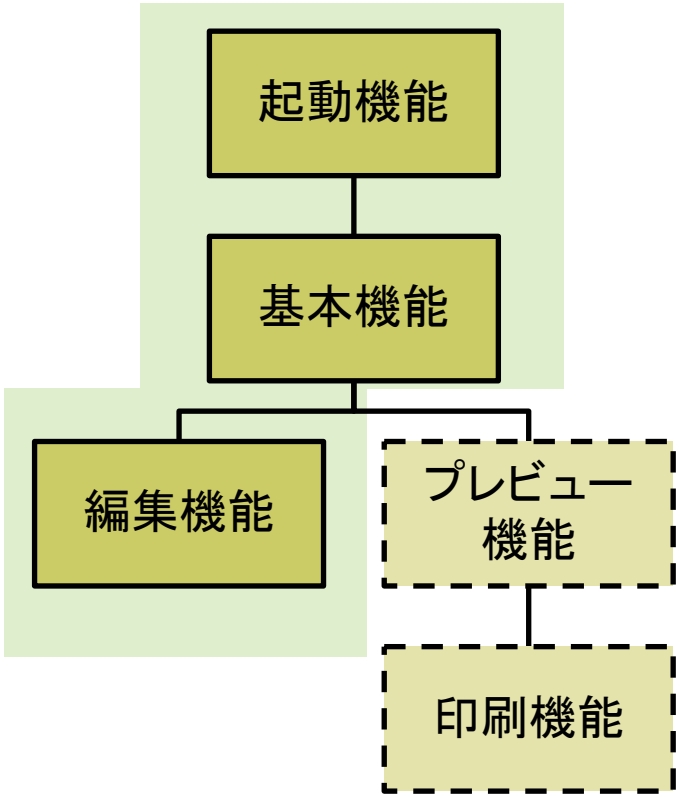
- メモリの自動管理手法
  - 使用しなくなった領域を，システムが自動的に検知して解放する
  - よって，プログラマーはメモリの解放処理（free）を書かなくてもよい
  - インタプリタ言語では一般的であり，近年はコンパイラでも利用可能
  - 速度が低下するので，OS自体のメモリ管理での採用はほとんどない
- 参照カウント法
  - 各メモリ区画のヘッダに，参照数（そこを指しているポインタの数）を保持させて管理し，参照数がゼロになったら解放処理を行う
- マーク・アンド・スイープ法
  - 一定時間ごとにシステムがすべてのメモリ区画の相互参照関係をマークをつけながらチェックし，参照がない領域を発見し解放する

大きなプログラムの実行

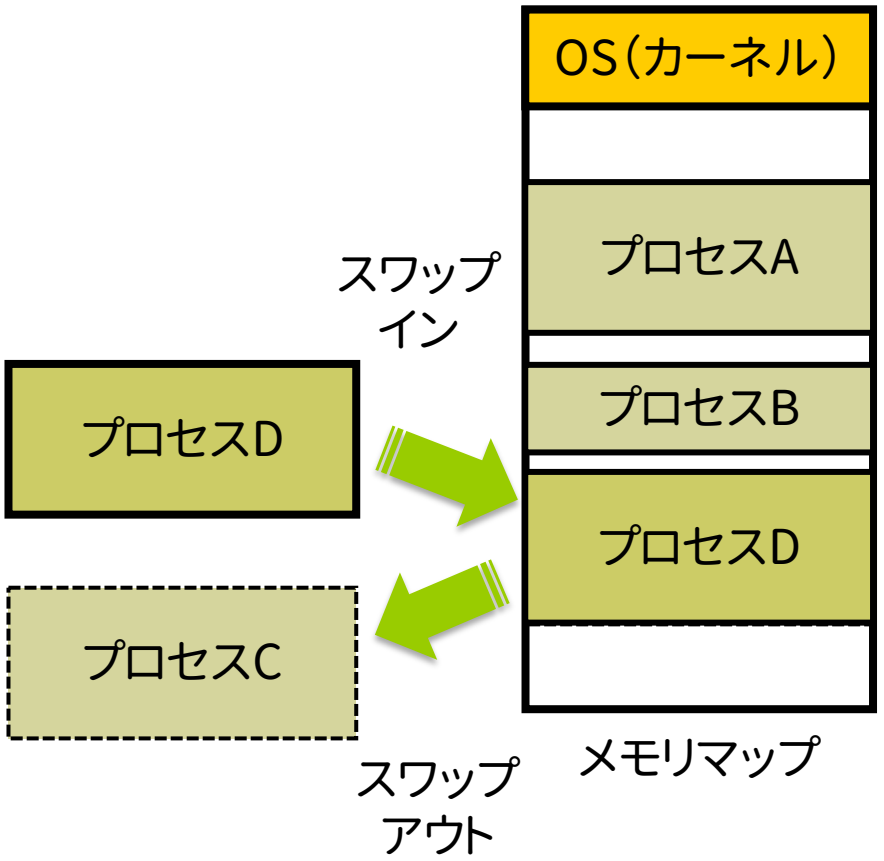
- 空き領域に入りきらないプログラムはどうするか？
  - 仮想記憶（virtual memory）によって，見かけのメモリを広げる
  - または，仮想記憶がなかった時代は以下のような方法を用いた
- メモリオーバレイ
  - プログラムをいくつかのモジュール（部分）に分割して開発する
  - 同時に使うことがない機能は，別々のモジュールに分けておく
  - 必要なモジュールだけをメモリに置くように交換しながら実行する
- 単純なメモリスワップ
  - 主記憶（メモリ）と2次記憶（ディスク）にあるプロセスを交換する方法
  - スワップアウト：待ち状態のプロセスを（丸ごと）2次記憶に追い出す
  - スワップイン：実行状態になったプロセスを主記憶に呼び戻す

大きなプログラムの実行

- オーバーレイ
  - プログラムを分割し，必要な部分だけメモリに読み込む



- メモリスワップ
  - プロセス単位で退避する

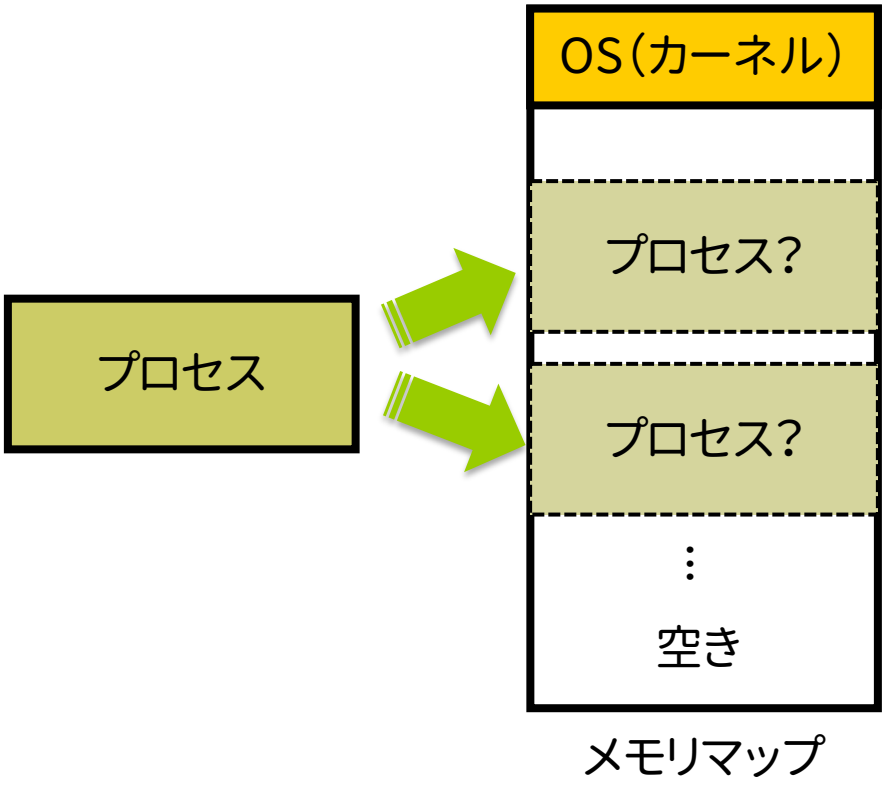


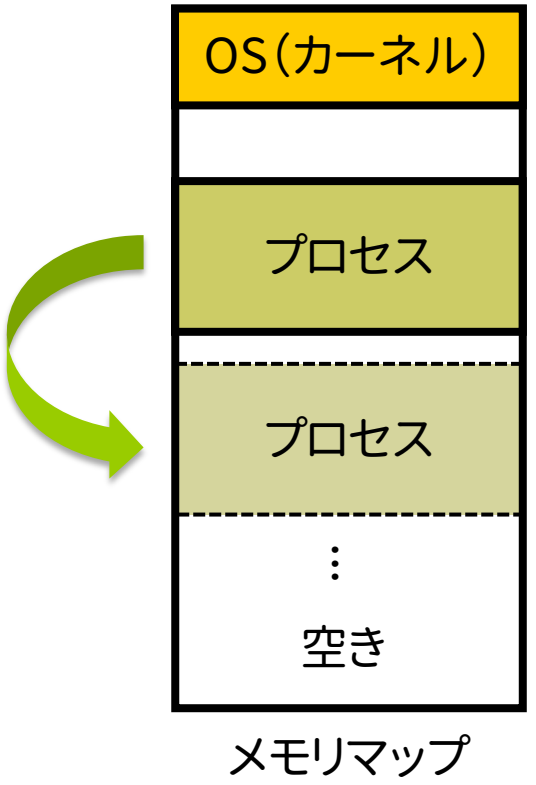
プログラムの再配置

- 再配置可能(relocatable)プログラム
  - 物理メモリのどの場所にロードされても正しく動くプログラム
  - 実行時の変数や関数の配置に合わせて,プログラムの中でそれらを参照するアドレス(番地)の値を,すべて適切に変換する
- 変数名とアドレスの関係
  - プログラムの変数名や関数名は,人間のための便宜的な名前
  - 実行形式では,それらはすべてアドレス(番地)への参照になる
- 再配置の方式
  - 静的再配置: プログラムをロードするとき全アドレスを一括変換する
  - 動的再配置: プログラム動作中でもメモリ内で移動可能(次に説明)
  - ページング技術を使うことで,再配置自体を不要にする(次回説明)

プログラムの再配置

- 静的再配置
  - プロセス起動時の配置対策
  - 最初どこに読み込んでも動く
- 動的再配置
  - プロセス実行中の配置対策
  - 実行中に主記憶内で移動可





動的再配置

- 相対アドレスの利用
  - プログラムの中で使うアドレス(変数,関数,ループの開始位置等)を,すべてプログラムの先頭からの相対位置(先頭からの差)にする
  - ⇔ 絶対アドレス形式(メモリアドレスを用いる)
- ベースレジスタ(再配置レジスタ)の利用
  - CPUに,プログラムの先頭(基準)アドレスを保持するレジスタを用意
  - CPUがメモリにアクセスするときは,相対アドレスにこの値を加える
  - [実効アドレス]=[現在のベースレジスタの値]+[相対アドレス]
- 動的再配置の実現
  - プロセスが移動しても,ベースレジスタの値を変更するだけでよい
  - ⇒ プロセスが動作中でも,メモリ内で位置を自由に移動できる

演習課題

課題 10a プロセス空間の各領域の役割

- この課題の狙いは，一般的なOSにおけるプロセス実行時のメモリの使われ方を理解することである。
- ノイマン型コンピュータでは，プログラムもデータもメモリに読み込まれて実行される。その際，プロセス空間はコード領域，静的データ領域，ヒープ領域，スタック領域のように分割されて利用される。
- この課題ではHOSは使わない。通常のC言語のコードを入力する。

手順

- HOSではなく，WindowsのVisual Studioなどの適当な環境で（Mac等でもよい），この資料の「変数や関数のアドレス」のページのプログラムを入力して実行し，表示結果の意味を詳しく考察せよ。
- 実行環境（OSやVisual Studioのバージョン）を変えたり，プログラムを修正したりして，表示結果がどう変わるか調べ，考察せよ。

変数や関数のアドレス(課題10a)

```c
#include <stdio.h>
#include <stdlib.h>

int g = 1, h = 2;

int func(int u, int v)
{
    int w;

    w = u + v + g + h;

    printf("uの番地: %p¥n", &u);
    printf("vの番地: %p¥n", &v);
    printf("wの番地: %p¥n", &w);
    printf("gの番地: %p¥n", &g);
    printf("hの番地: %p¥n", &h);
    return w;
}
```

```c
int main(void)
{
    int a, b, *p;

    p = (int *) malloc(sizeof(int));
    scanf("%d %d", &a, p);
    b = func(a, *p);

    printf("aの番地: %p¥n", &a);
    printf("bの番地: %p¥n", &b);
    printf("pの番地: %p¥n", &p);
    printf("malloc領域の番地: %p¥n", p);
    printf("mainの番地: %p¥n", &main);
    printf("funcの番地: %p¥n", &func);
    printf("printfの番地: %p¥n", &printf);
    printf("scanfの番地: %p¥n", &scanf);

    scanf("%d", &a); return 0;
}
```

Visual Studio では scanf_s

演習課題

- 課題 10b HOSにおけるメモリ割り当て
  - この課題の狙いは，基本的なメモリの割り当ての仕組みとそれにともなう問題点を実際のプログラムで学ぶことである。
  - HOSのような組み込みOSでは，ヒープ領域からメモリを割り当てるのではなく，静的データ領域にメモリプールと呼ばれる固定サイズの領域をあらかじめ確保しておき，そこからメモリを割り当てることが多い。

- 手順
  - まず，win-memory のプログラムを実行して結果を考察せよ。
  - 次に，コメントアウトを解除してタスクの数を増やして実行し，メモリの空き領域のバイト数は足りているはずなのに，新たにメモリブロックを確保できない現象が発生することを確認せよ。
  - 上記の現象について，なぜ発生するのか考察せよ。また，このプログラム中でセマフォを利用している理由についても考えてみよ。

HOSのメモリ割り当て機能

サービスコール	意味	説明
cre_mpf	create mpf (fixed-length memory pool)	固定長メモリプールの生成
acre_mpf		同上（ID自動割り当て）
del_mpf	delete mpf	固定長メモリプールの削除
get_mpf	get mpf	固定長メモリブロックの獲得
pget_mpf		同上（ポーリング）
rel_mpf	release mpf	固定長メモリブロックの解放
cre_mpl	create memory pool	可変長メモリプールの生成
acre_mpl		同上（ID自動割り当て）
del_mpl	delete memory pool	可変長メモリプールの削除
get_mpl	get memory pool	可変長メモリブロックの獲得
pget_mpl		同上（ポーリング）
rel_mpl	release memory pool	可変長メモリブロックの解放