オペレーティングシステム
第８章 主記憶（メモリ）

https://github.com/tctsigemura/OSTextBook

主記憶

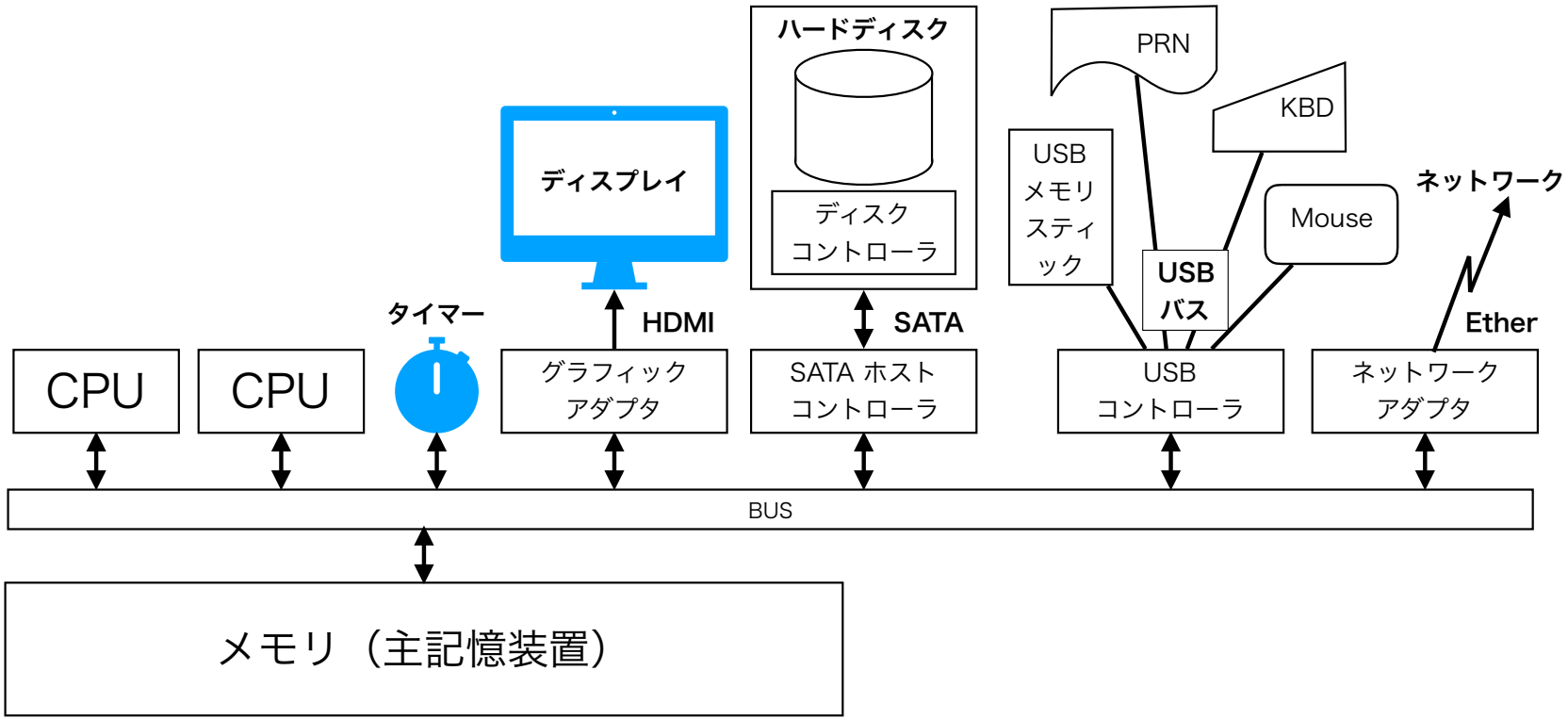
主記憶はプログラム，データ，スタック等を置くメモリのこと
CPU と同様に重要な装置

- TeC の 256 バイトの RAM
- H8/3664 の 32KiB の ROM と 2KiB の RAM
- PC のメモリ（4GiB 〜 16GiB 程度？）

この章では，主記憶を管理し複数のプロセスに適切に割り振り，かつ，
プロセス同士が干渉しないように分離する方法を学ぶ.

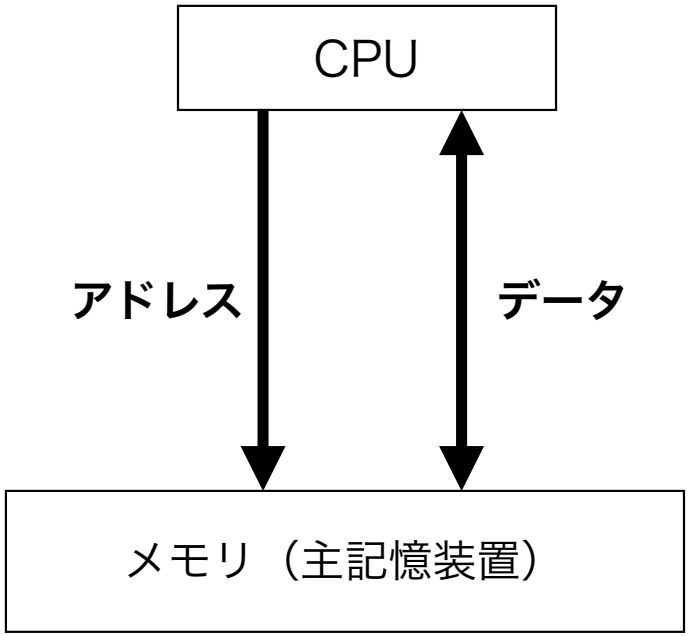
ハードウェア構成

メモリを共有する SMP（Symmetric Multiprocessing）システム

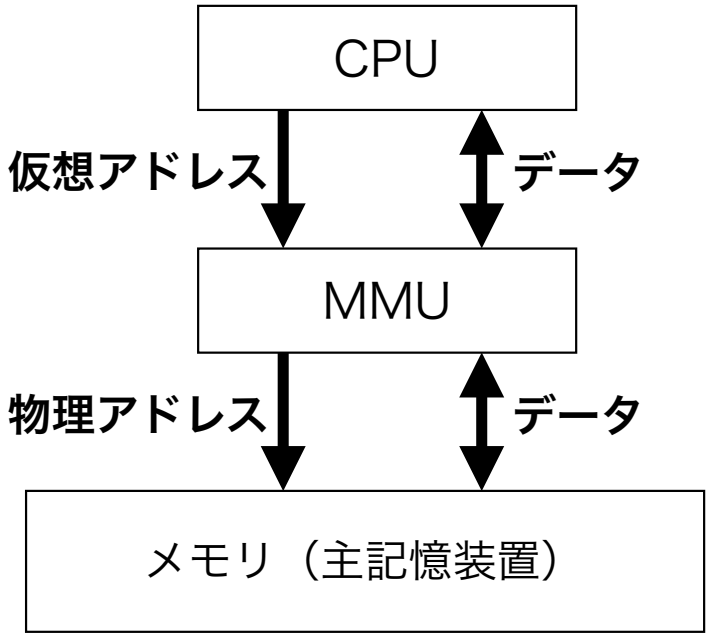


この講義で前提にしているコンピュータの構成

本章で用いるモデル



(a) 単純なモデル



(b) 仮想化が可能なモデル

MMU（Memory Management Unit：メモリ管理装置）

- メモリ保護機構，メモリ再配置機構，仮想記憶
- 仮想アドレス，物理アドレス

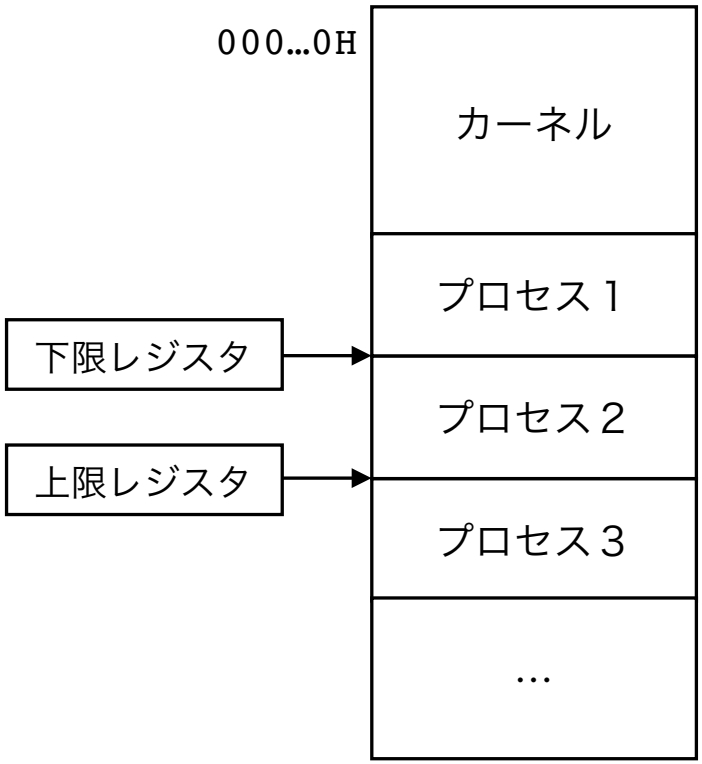
メモリ保護機構

- CPU を仮想化した
  複数のプロセスを同時にロードし並列実行できるようになった
- ユーザプロセスが複数存在する
  プロセスが他のプロセスや OS を破壊しないか？
  他のプロセスの**メモリを保護**する機構が必要

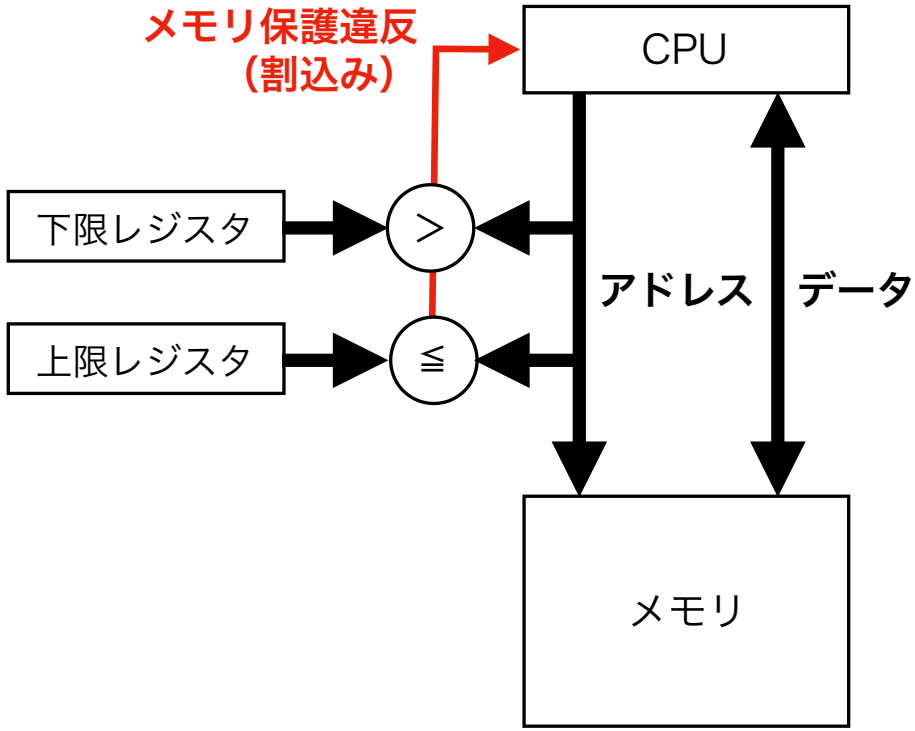
プロセスは自身に割当てられたメモリ以外をアクセスできないようにする.

上限・下限レジスタ

- プロセスがアクセスしても良いアドレスの範囲を設定する.
- プロセスのメモリアクセスはアドレスをチェックする（ハード）
- レジスタを操作できるのはカーネルだけ.



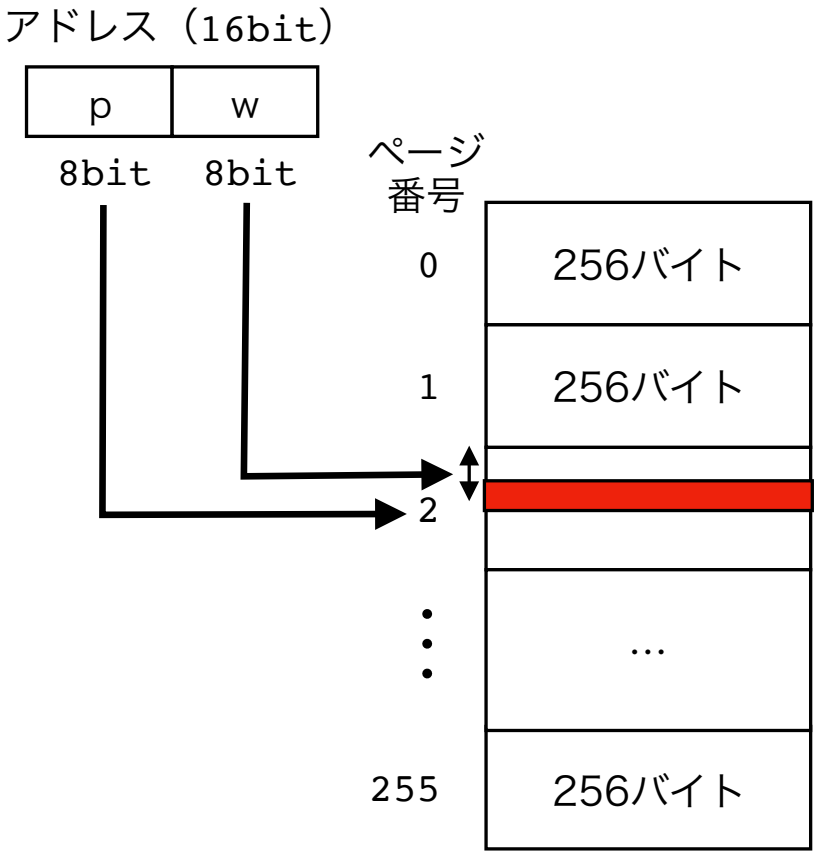
(a) 物理アドレス空間



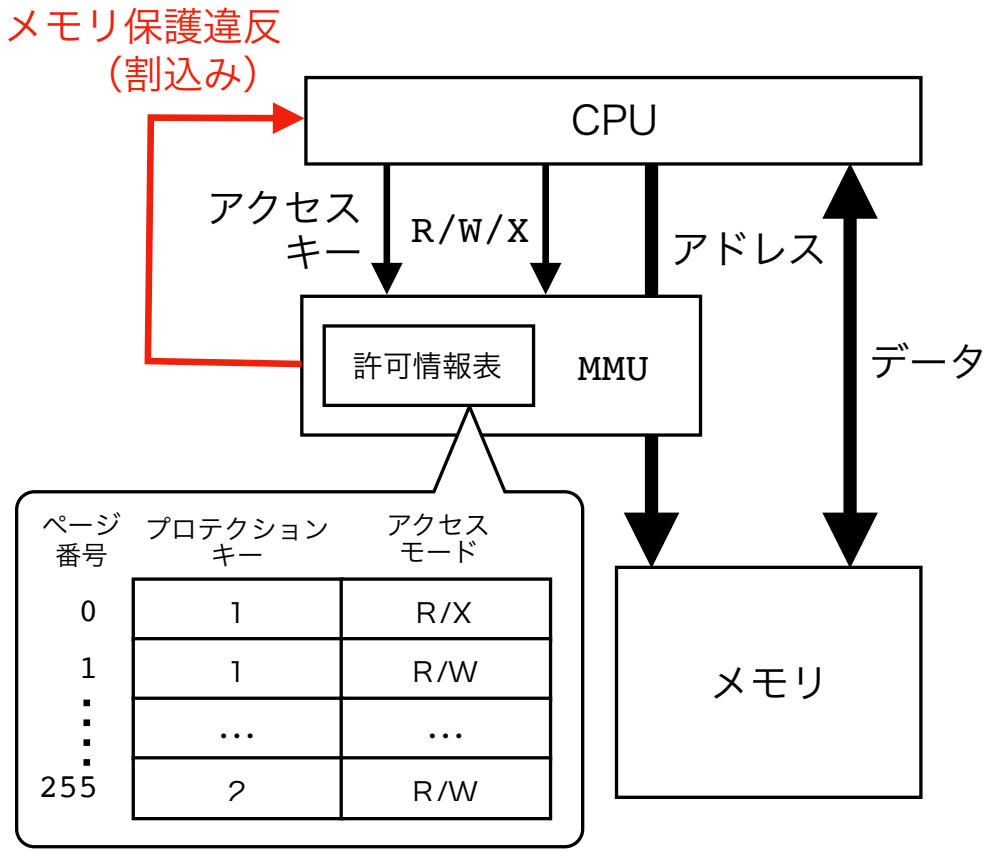
(b) ハードウェア構成

ロック／キー機構

- メモリをページに分割する（例：64KiB を 256 ページに）
- ページ毎に，アクセスしてよいプロセスとアクセス方法を記録



（a）メモリ空間

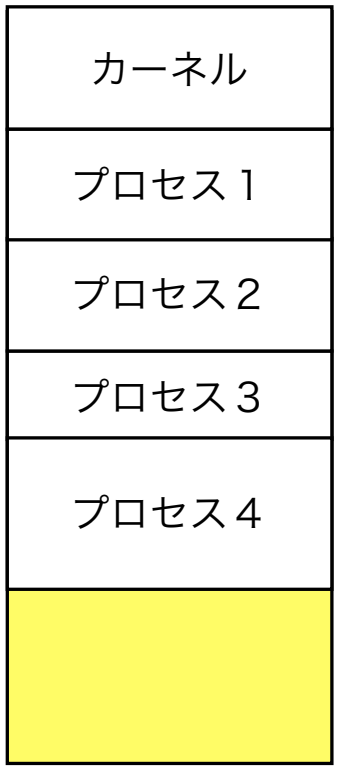


（b）ハードウェア構成

メモリフラグメントとコンパクション

- 様々なサイズのプロセスが存在する.
- プロセスの生成・終了が繰り返される.
- メモリフラグメント（断片）が沢山できる.
- フラグメントを解消すために**動的再配置**が必要！

(a) メモリフラグメント



(b) メモリコンパクション

再配置可能オブジェクトファイル

ロード時に機械語に含まれるアドレスを確定

- コンパイル時にはロードアドレスが分からない.
- ロード時にアドレスを確定できる仕組みが必要
- 実行可能形式ファイルに機械語と**再配置表**を格納
- ロード時にプログラムやデータに含まれるアドレスを変更

例：1234H 番地にロードされた場合

JMP 0100H の機械語は JMP 1334 に変更

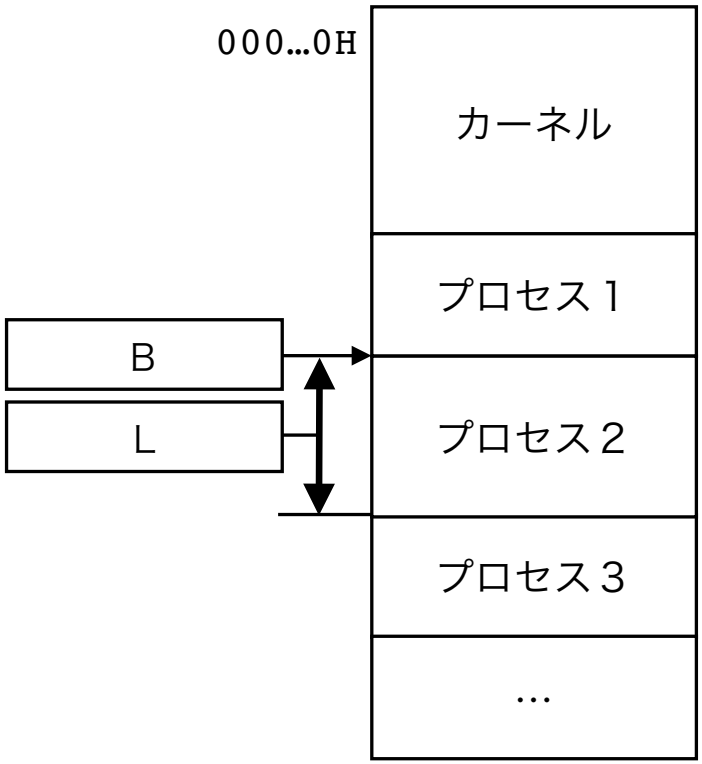
動的再配置に応用できるか？

この方法では動的再配置までは無理

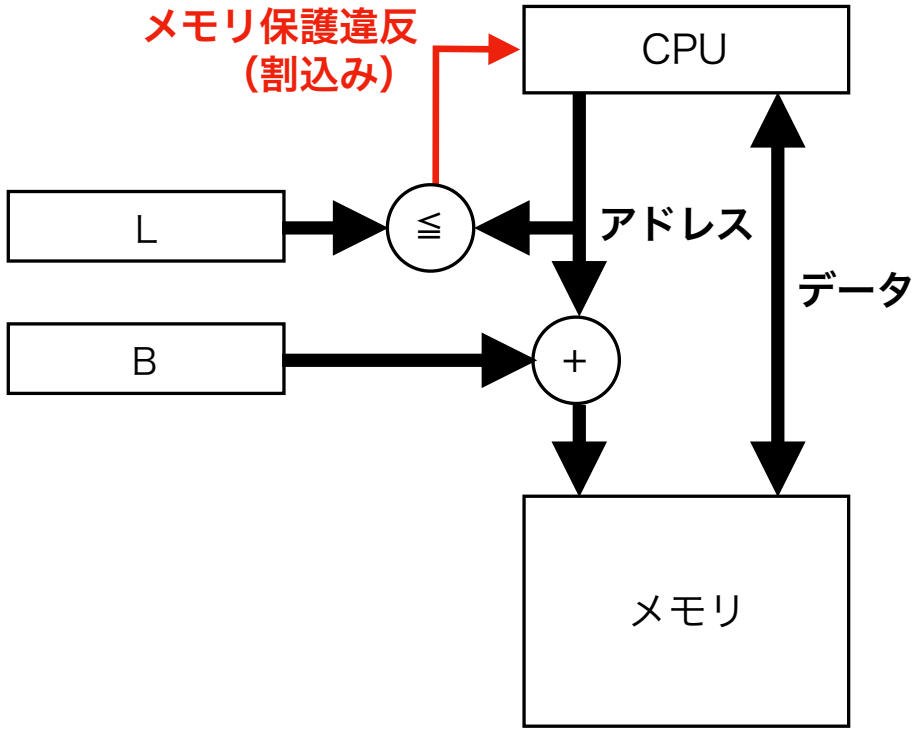
- CPU レジスタにアドレスがあるかも知れない.
- アドレスがスタックに PUSH されているかも.
- `malloc()` で確保した領域にも含まれるかも
  リスト構造の次のノードへのポインタなど

リロケーションレジスタ

- 動的再配置を可能にするハードウェア機構
- メモリ保護の機能も持つ.
- ロードアドレス（B:Base）と大きさ（L:Limit）を記録



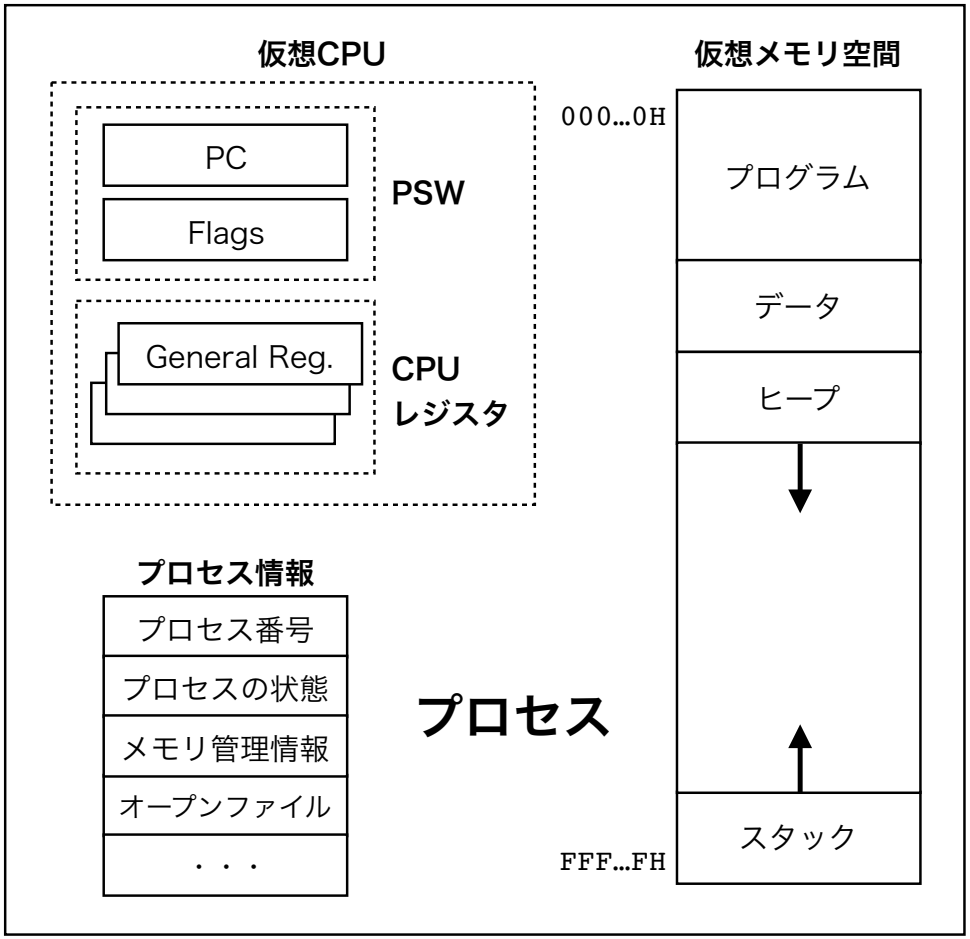
(a) 物理アドレス空間



(b) ハードウェア構成

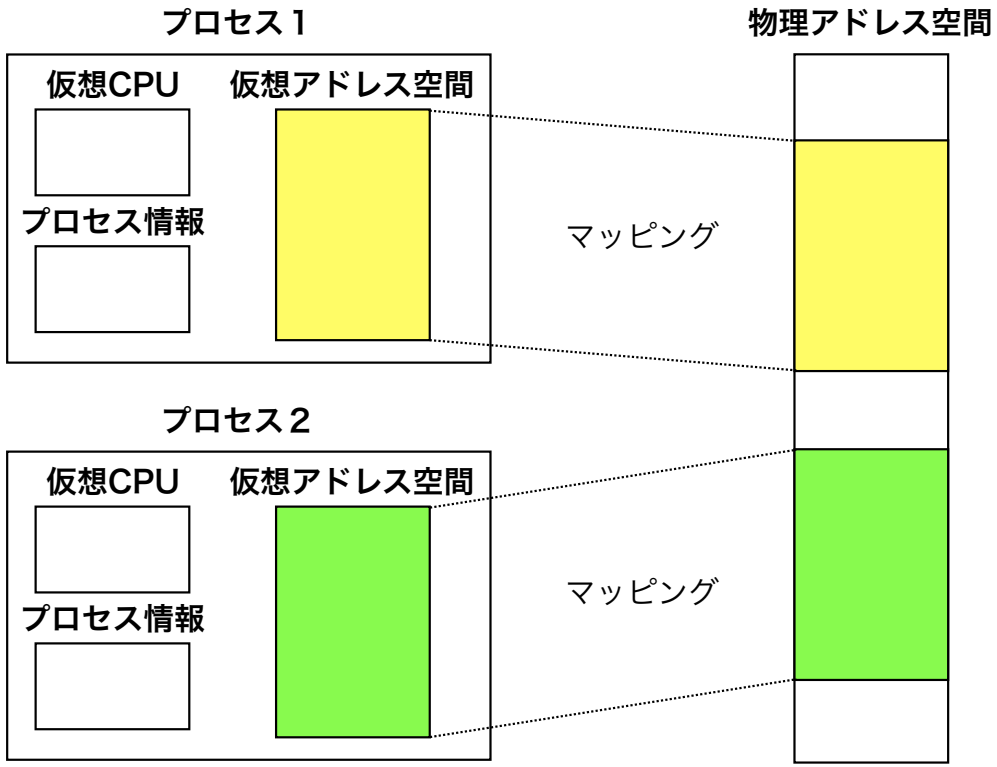
仮想アドレス空間

プロセスは専用の仮想アドレス空間を持っていた（参考）



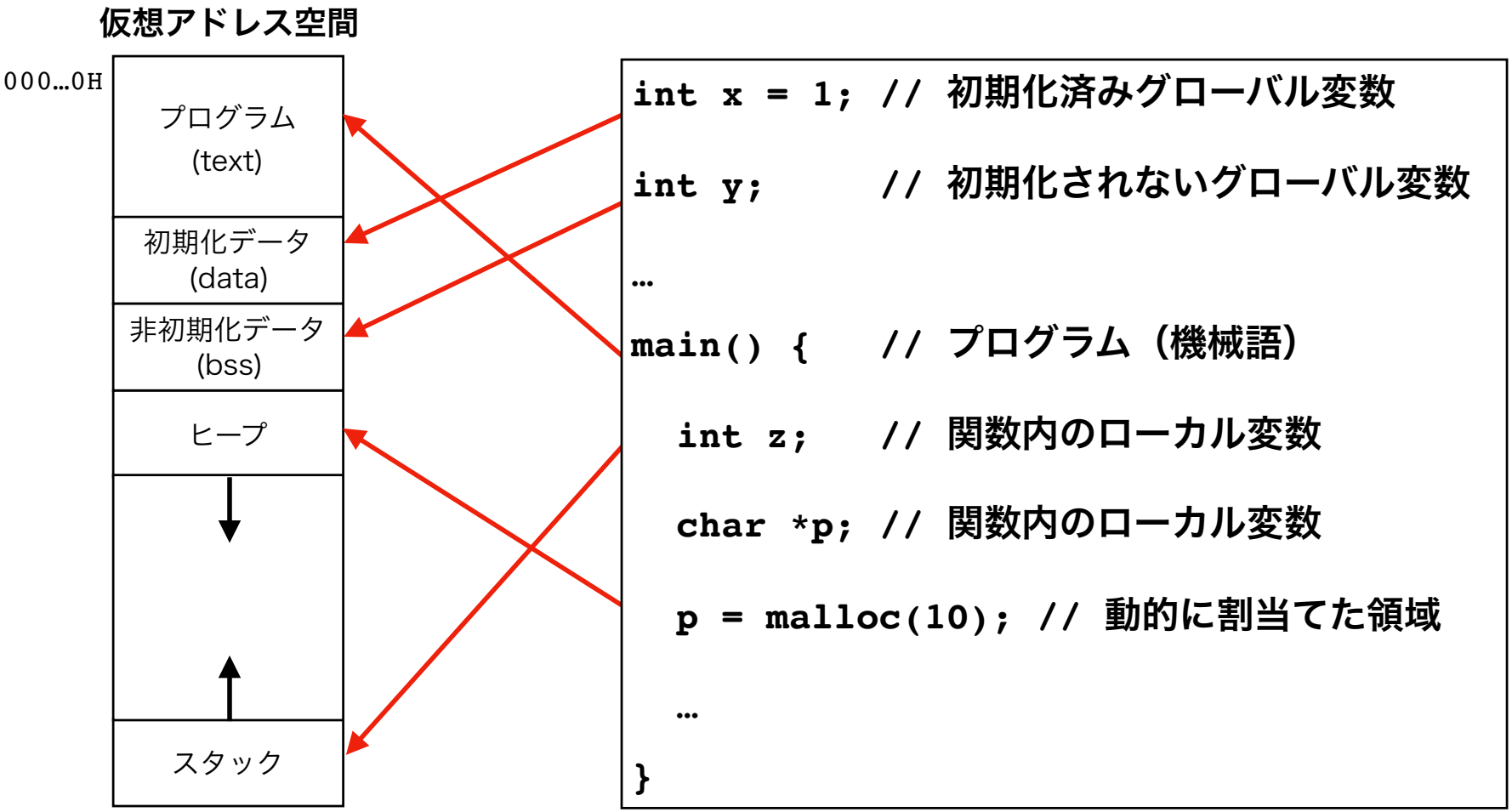
主記憶の仮想化

- プロセス毎に独立した仮想アドレス空間を持つ.
- 仮想アドレスから物理アドレスに MMU がマッピングする.
- 通常は**多重仮想記憶**方式を用いる.



仮想化アドレス空間の配置

- （伝統的な）UNIX の配置を C 言語プログラムと対比
- ヒープとスタックの間はどれだけ空けておくか？



前のプログラムをアセンブリ言語に変換したもの

```
_x      DW      1       // int x = 1;
_y      WS      1       // int y;
_main   PUSH    FP      // void main() {
        LD      FP,SP
        PUSH    G3      //   int z;
        PUSH    G4      //   char *p;
        LD      G0,#10  //
        PUSH    G0      //
        CALL    _malloc //   p = malloc(10);
        ADD     SP,#2   //
        LD      G4,G0   //
        POP     G4
        POP     G3
        POP     FP
        RET             // }
```