

# Virtual memory

---

## 仮想記憶定义

主記憶データを補助記憶上に配置・管理 在辅助存储器上配置和管理主存储器数据 这种机制允许将主存储器（物理内存条）中的数据，与辅助存储器（如硬盘或SSD）上的空间进行统一的配置和管理。

## 利点

- 提供比物理内存更大的存储空间
- 实现内存的安全共享与保护
- 另一种看法 (別の見方): 在主内存之下，再设置一个位于辅助存储器上的缓存层级，可以把主内存 (DRAM) 看作是辅助存储器 (硬盘/SSD) 的一个高速缓存 (Cache)

下面对以上三点进行详细解读

让主存储器看起来更大

当系统中同时存在多个进程时引发的问题

程序无法使用“绝对地址”，因为它在物理内存中的“配置场所”（起始位置）完全是不确定的，它取决于“其他进程”占用了哪些位置。

虚拟内存带来的缓存效应

它利用了“局部性原理”，仅仅将进程当前最活跃的“工作集”加载到了物理内存中。因为加载到内存中的（只是）工作集这一小部分，所以（我们）才能够同时运行多个进程

**用有限的物理内存，撬动无限的虚拟空间**

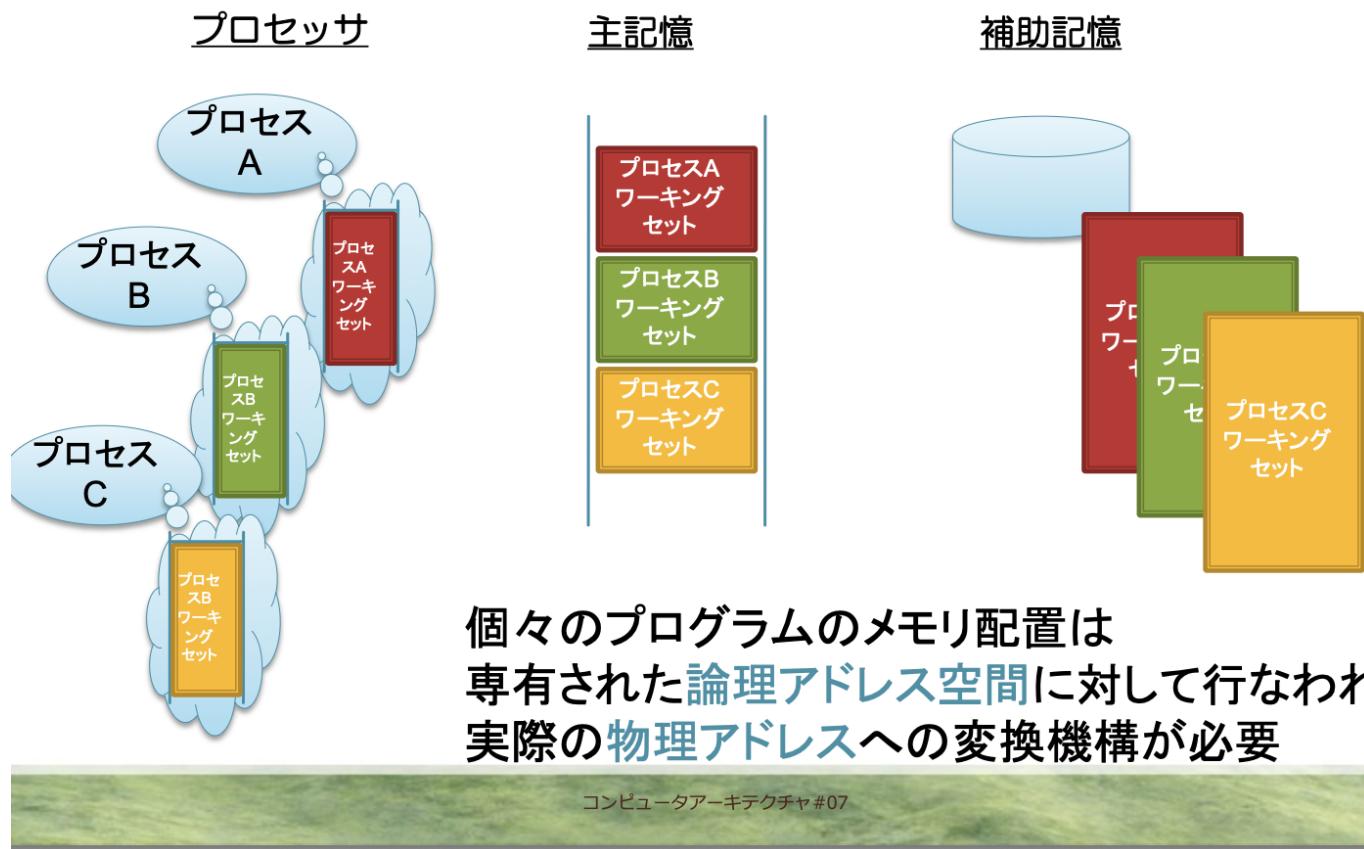
逻辑地址空间与物理地址空间

每个程序（进程）的内存配置，都是针对其‘专属的逻辑地址空间’来进行的，必须有一个将其转换为‘实际物理地址’的机制

这张图现在被用来具象化“逻辑地址空间”和“物理地址空间”的区别。

- 逻辑地址空间 (左侧): 进程A“以为”自己拥有的那个巨大蓝色云朵，就是它的“逻辑地址空间”。
- 物理地址空间 (中间): 主存储器上那个大小有限、插着A、B、C三个工作集的插槽，就是“物理地址空间”。

## 論理アドレス空間と物理アドレス空間



### 虚拟内存的功能

- 将常用数据加载到内存，溢出数据换出到辅助存储
- 将逻辑地址转换为物理地址
- 保护其他进程的地址空间
- 当缓存用（管理数据换入换出）。
- 当翻译官（转换逻辑和物理地址）。
- 当保安（保护进程间内存隔离）。

### 缓存与虚拟内存

两者虽独立发展，但本质类似，这两者在系统设计原理上几乎是相同的，都是“存储器层次结构”思想的体现。是为了解决两个不同问题而独立发展起来的：

- Cache 是为了解决 CPU 与 主内存 (DRAM) 之间的速度差距问题。
- 虚拟内存主要是为了解决 主内存 (DRAM) 与 辅助存储器 (硬盘) 之间的容量差距和多进程管理问题。

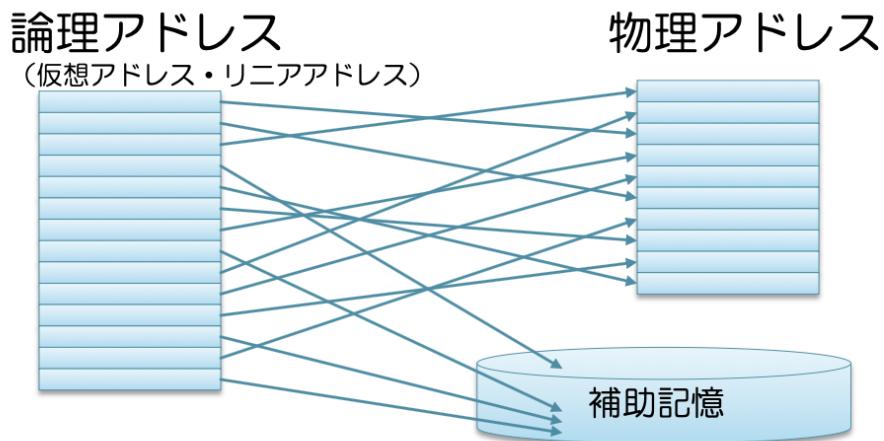
### 地址映射 (アドレスマッピング)

“地址映射”就是我们之前讨论的**地址转换机制**的正式术语。它是虚拟内存系统的核心动作。

- 将逻辑地址由系统(HW+OS)转换为物理地址
- 根据物理地址访问主存储器

## ■ アドレスマッピング

- 論理アドレスをシステム(HW+OS)で物理アドレスへ変換
- 物理アドレスにしたがって主記憶アクセス



当CPU要访问一个逻辑地址时，这个“映射机制”会告诉它去哪里找：

- “翻译”成功，指向“物理地址”：这就是“缓存命中”（Slide 11），程序继续运行。
- “翻译”失败，指向“辅助存储”：这就是“缺页中断 (Page Fault)”（Slide 11），系统必须暂停程序，去硬盘上把对应的数据页 (Page) 调入物理内存中，然后更新映射箭头，最后再恢复程序运行。

配置上的优点(配置上の利点)

讨论的是虚拟内存（地址映射）在“内存配置/分配”方面的好处。

**核心问题：**“要求一整块大尺寸区域的程序”

这是在没有虚拟内存的“物理地址世界”里，操作系统（OS）最头疼的问题之一。比如，一个程序启动时请求1GB的连续内存。OS就必须在物理内存中找到一个完整的、未被占用的、连续的1GB空间。

随着系统长时间运行，物理内存会变得“碎片化”（Fragmented）——这里有200MB空闲，那里有500MB空闲，但就是没有一个完整的1GB空间。最终，即使总空闲内存（如2GB）远大于程序的需求（1GB），但因为没有连续的空间，OS也不得不拒绝程序的请求，导致程序无法运行。

**虚拟内存的解决方案：**

- 在物理地址空间中，以‘Page’为单位进行再配置

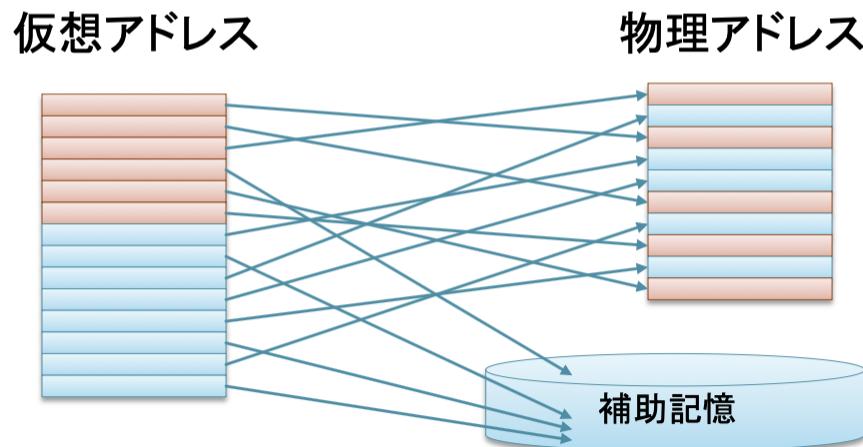
- 因此) 没有必要确保(分配)一块连续的区域

这两句话是核心，完美地解决了上述“外部碎片”问题。

当程序请求 1GB 逻辑空间时，OS 不再需要去寻找 1GB 的物理连续空间。相反，OS 只需要在物理内存中找到任意 262,144 个空闲的“页”（假设  $1\text{GB} / 4\text{KB} = 262,144$ ），这些页可以完全是东一块、西一块、分散在各处的。

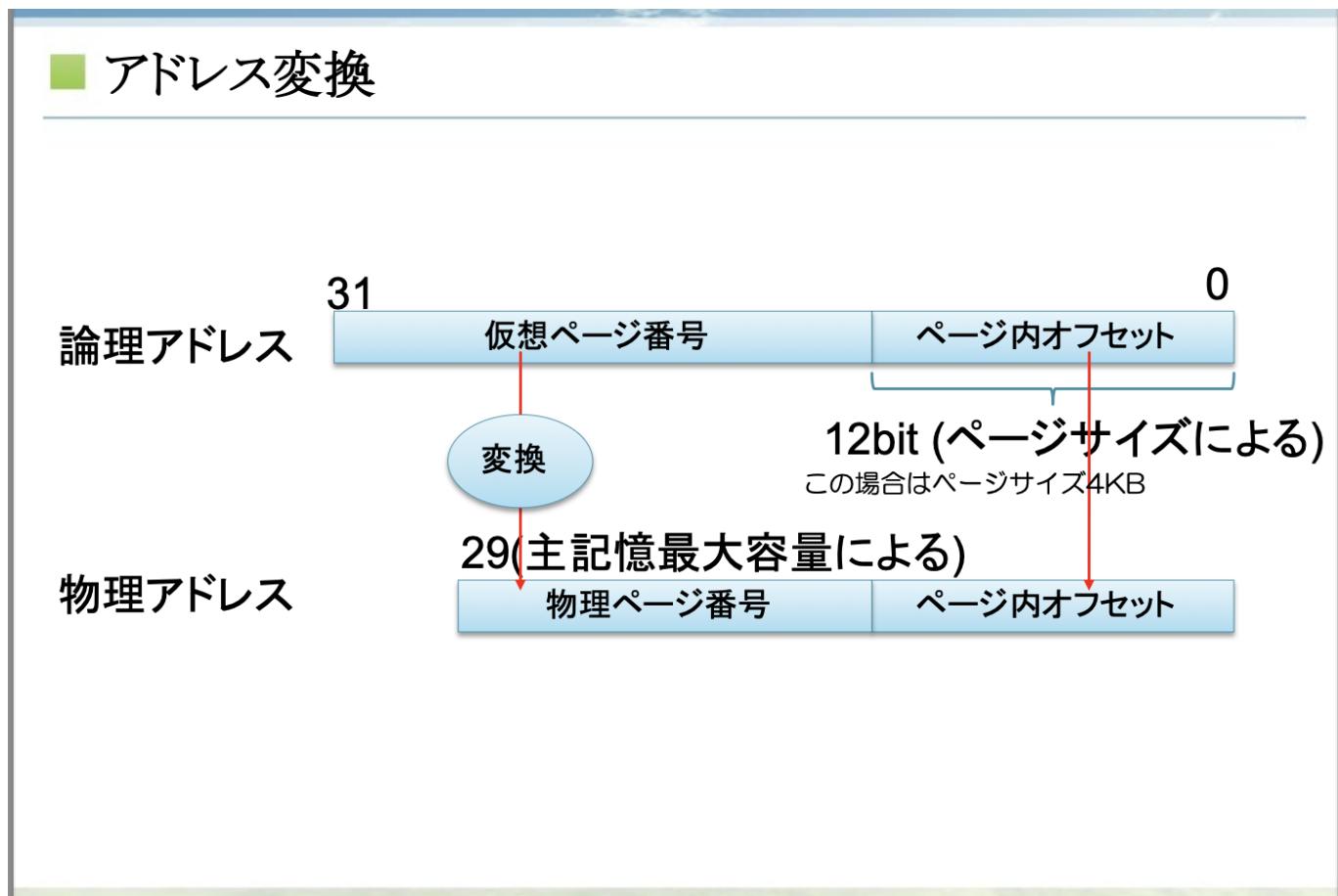
## ■ 配置上の利点

- まとめたサイズの領域を要求するプログラム
- 物理アドレス空間ではページ単位で再配置
  - 連續した領域を確保する必要がない



- 左侧(虚拟地址): 展示了程序“以为”自己拿到的完美内存——一个由粉色块和蓝色块组成的、连续的、完整的大空间。这就是它向 OS 请求的“一整块大尺寸区域”。
- 右侧(物理地址): 展示了 OS 的“实际操作”。它并没有给程序一块连续的物理空间，而是将粉色页和蓝色页\*\*“杂乱地”、“碎片化地”\*\*插入到物理内存的各个可用空隙中。

逻辑地址（虚拟地址）如何被分解和转换成物理地址。



### 逻辑地址 (論理アドレス)

- 左侧: 仮想ページ番号 (Virtual Page Number, VPN): 这是用来标识程序所使用的、虚拟内存中的第几页。它是地址转换的关键输入。
- 右侧: ページ内オフセット (Page Offset): 页内偏移用来标识所请求的数据位于该页内部的哪个位置 (字节)。

页内偏移不参与地址转换，它在逻辑地址和物理地址中是完全相同的。

### 物理地址 (物理アドレス)

- 左侧: 物理ページ番号 (Physical Page Number, PPN)
- 右侧: ページ内オフセット (Page Offset)

### 转换流程

- 转换 (変換) 箭头清晰地从 虚拟页号 (VPN) 指向 物理页号 (PPN)。这个转换是由 页表 (Page Table) 来完成的。
- CPU 使用 VPN 作为索引去查询页表（通常先查TLB, TLB Miss时再查主存中的页表），页表返回对应的 PPN。
- 将返回的 PPN (物理页号) 与原始的 Page Offset (页内偏移) 拼接起来，就得到了完整的物理地址，用于访问主存储器。

### 虚拟内存的设计平衡

1. 未命中惩罚 (Miss Penalty) 是极端的 将主内存 (DRAM) 作为辅助存储器 (硬盘) 的缓存，其未命中惩罚 (即缺页中断) 比 CPU 缓存未命中的惩罚大得多。
2. 块大小 (页面大小) 大 在虚拟内存系统中，数据交换的基本单位是\*\*“页” (Page) \*\*。由于硬盘访问延迟 (Latency) 极高 (主要是机械寻道时间)，一次性传输更多的数据 (即增大 Page Size) 可以提高数据吞吐量，以更好地利用局部性原理中的空间局部性，从而分摊巨大的访问延迟，降低平均访存时间。
3. 倾向于全相联映射
  - 这意味着任何虚拟页都可以存放在主内存的任何一个物理页框中。
  - 全相联映射能够最大限度地减少前面提到的冲突性未命中 (Conflict Miss)，这是为了满足“不惜一切代价降低 Miss Rate”的首要目标。
  - 在硬件缓存中，全相联成本极高。但在虚拟内存中，映射表 (Page Table) 是存放在主内存中的，可以很容易地包含所有虚拟页的映射信息。
  - 软件 (OS) 参与控制 由于单次缺页中断的惩罚 (1,000,000 周期) 远远大于 OS 软件处理的开销 («)，所以这个软件开销是可以接受甚至值得的。

### (ソフトウェアのオーバーヘッド « ミスペナルティ)

## 页表 (ページテーブル)

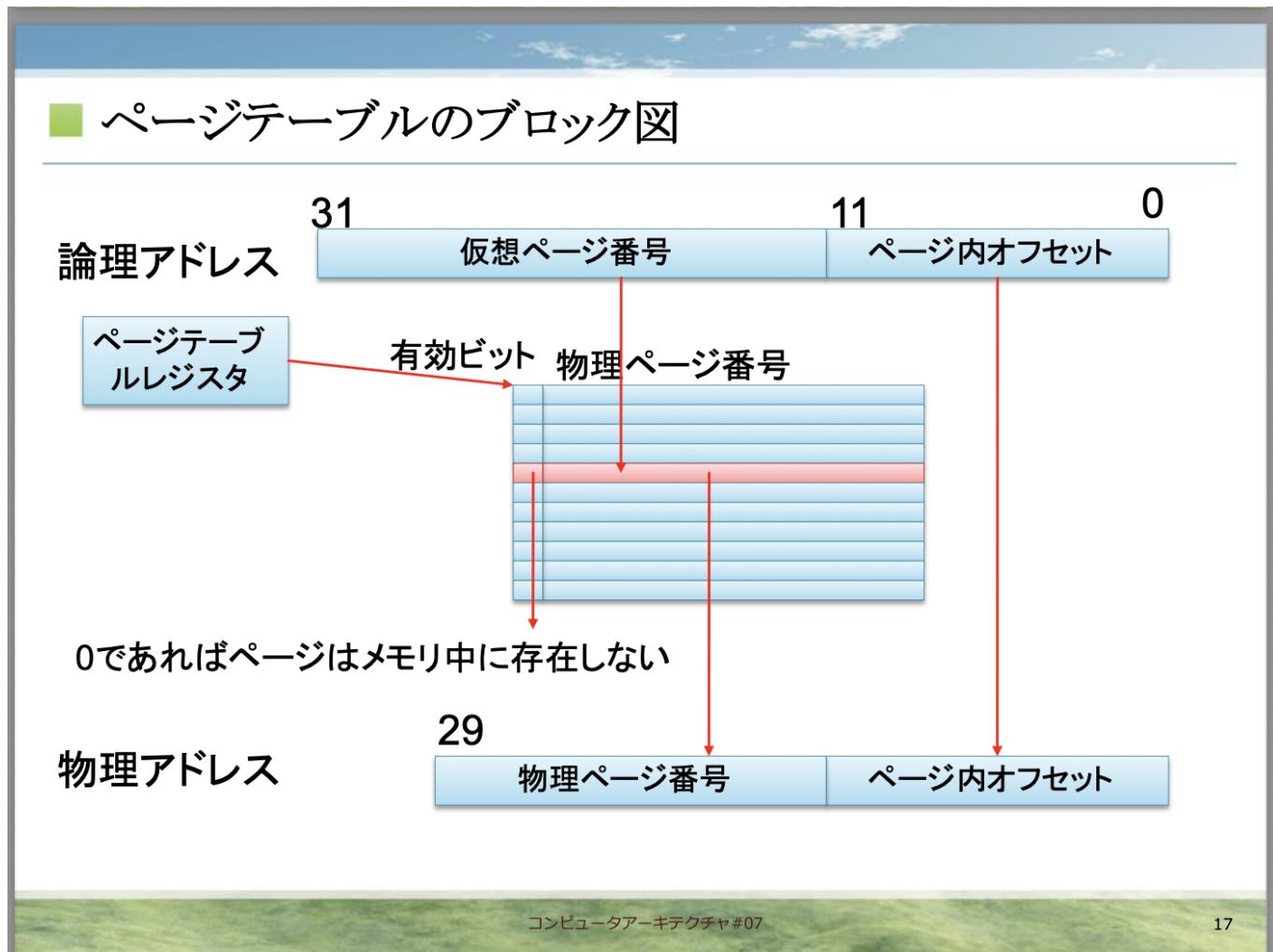
页表是操作系统用来实现“逻辑地址到物理地址转换”的核心查找表。它是虚拟内存的“翻译字典”。

### 核心结构

- 将虚拟页号作为索引来存储物理页号
- 物理页号 (PPN) 就是输出 (Output)。页表在这个索引位置存储的数据，就是对应的 PPN。
- 页表记录着 VPN → PPN 的一对映射关系。

### 页表的存储位置与访问

- 页表是存放在主存储器 (DRAM) 中的一个大型数据结构。
- “页表寄存器 (ページテーブルレジスタ)” 页表寄存器 (Page Table Register) 是一个位于 CPU 内部的专用硬件寄存器。它存储着当前运行进程的页表在主内存中的起始地址 (或根地址)。当操作系统切换进程时 (上下文切换)，它只需要更新这个寄存器，使其指向新进程的页表起始地址，CPU 就能立即开始对新进程进行地址转换。
- “首先假设页表在内存中占据一个固定的连续区域” 物理地址 = (页表寄存器起始地址) + (VPN × 页表项大小)。这是一个简化假设，用于说明最基本的页表查找机制



这页PPT是关于页表查找机制的流程图，它将前面的概念（逻辑地址结构、页表、页表寄存器）整合在一起，展示了地址转换的完整过程。

1. 输入：逻辑地址 (論理アドレス)
2. 转换机制：页表查找

起点：ページテーブルレジスタ (页表寄存器)，它提供了页表在物理内存中的起始地址。这是查找的第一步。

VPN (虚拟页号) 被用作索引/下标，从页表起始地址开始，定位到页表中对应的页表项 (图中被高亮的红色行)。

有效ビット (Valid Bit - 有效位): 标识当前这个页表项的映射是否有效。

3. 输出：物理地址 (物理アドレス)

从页表项中提取的 PPN。它决定了数据在物理内存中的起始页。

最终形成: PPN 和 Page Offset 拼接起来，形成完整的物理地址，用于访问主存储器。

## 页表的实现 (ページテーブルの実装)

1. “为所有逻辑页准备条目” (全ての論理ページに対応するにエントリを用意)

页表是完备的。对于一个进程的整个逻辑地址空间内每一个虚拟页 (Virtual Page)，页表中都有一个对应的条目 (Entry)。因此，不需要标签 (なので、タグはいらない)，页表中的第 N 个条目，就必然对应虚拟页 N，无

需再存储 N 这个标签。

2. “对每个进程是独立的” (プロセス毎に独立) 是上下文的一部分 (PCやレジスタ同様、コンテクストの一部)

“上下文 (Context) ”是操作系统用来描述一个进程所有状态的信息集合，包括程序计数器 (PC) 、寄存器 (Registers) 内容等。

当操作系统进行\*\*进程切换 (Context Switch) \*\*时，除了保存和恢复PC和寄存器外，它必须同时更新 页表寄存器，使其指向新进程的页表起始地址。

“如果进程不同，则页表也不同” (プロセスが異なればページ表も異なる)

## 缺页中断 (ページフォルト)

缺页中断 (Page Fault) 是当CPU访问一个逻辑地址时，发现对应的数据页当前不在物理内存中，需要从硬盘调入时发生的一种异常。这正是虚拟内存作为“缓存”时发生“未命中 (Miss) ”的表现。

1. 缺页中断的发生条件

- “对应逻辑地址的有效位无效” (対応する論理アドレスに対応する有効ビットが無効) → 物理页号没有被分配 (不存在于内存中) ” (物理ページ番号が割り振られていない (メモリ内に存在しない) )

2. 缺页中断发生后的处理流程

- “发生异常，将控制权转移给OS” (例外を発生してOSに制御を移す)
- 操作系统 (OS) 的处理步骤：
  1. “从交换区寻找对应的页面” (スワップ領域から該当ページを探す)
  2. “决定物理内存中的存储位置” (物理メモリ中の格納場所を決める)

## 交换区 (スワップ領域)

交换区是硬盘上的一块空间，专门用于临时存放从物理内存中被“换出”的页面数据，是实现“让主内存看起来更大”这一功能的基础。

1. 交换区的创建和用途：

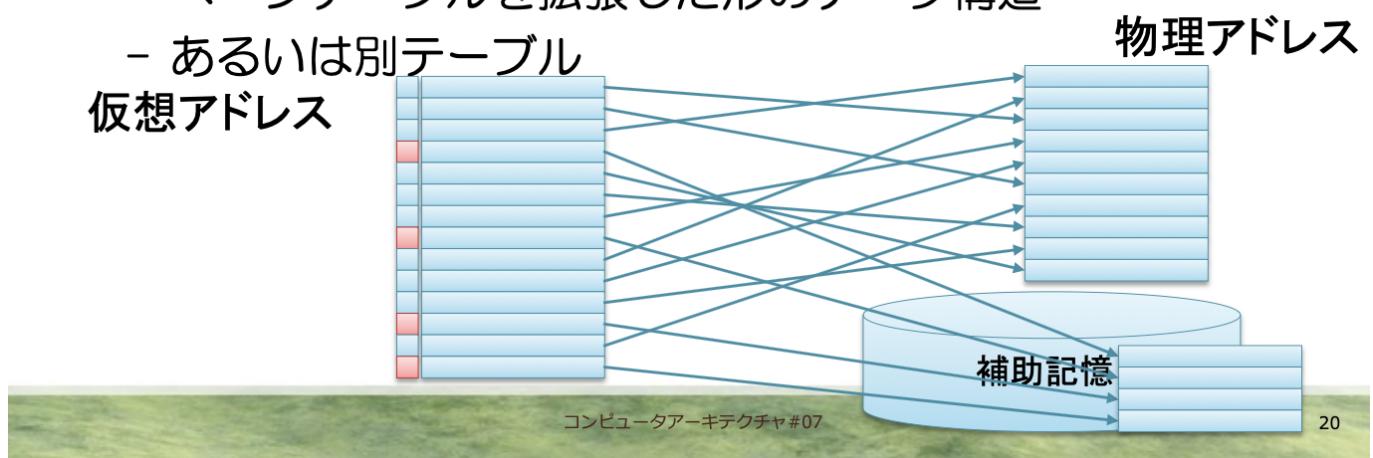
- 交换区是硬盘上的一块空间，专门用于临时存放从物理内存中被“换出”的页面数据，是实现“让主内存看起来更大”这一功能的基础。
- “一个可以存储进程所有页面的空间” (プロセスの全てのページを格納できる空間) 理论上，交换区需要足够大，以存放所有当前进程可能被换出的页面。它是进程完整逻辑地址空间的“仓库”。

2. 追踪页面位置的数据结构

- “也要创建表来追踪各个虚拟页在磁盘上的配置位置” (各仮想ページがディスクのどこに配置されているかのテーブルも作成) 可能是页表的扩展形式的数据结构 (ページテーブルを拡張した形のデータ構造)/或者是一个单独的表 (あるいは別テーブル)

## ■ スワップ領域

- ・プロセス生成時にOSがディスク上に作成
  - プロセスの全てのページを格納できる空間
- ・各仮想ページがディスクのどこに配置されているかのテーブルも作成
  - ページテーブルを拡張した形のデータ構造
  - あるいは別テーブル



交换区是虚拟内存机制在辅助存储器上的物理体现。它的存在使得主内存可以作为硬盘的“缓存”，并通过 OS 维护的额外映射表来精确追踪和管理所有被换出页面的位置。

## 页面替换 (ページ置き換え Page Replacement)

页面替换是操作系统 (OS) 在发生缺页中断 (Page Fault)，但物理内存已满时必须执行的操作。它决定了应该“牺牲”物理内存中的哪一页，来腾出空间给新页。这对应于硬件缓存 (Cache) 中的替换策略 (Replacement Policy)。

1. 页面替换的基本概念：“新页面 → 替换出物理内存的页面”(新しいページ → 代わりに物理メモリから追い出されるページ)(由 OS 决定)。这是虚拟内存与硬件缓存的主要区别之一，虚拟内存的替换策略由软件 (OS) 控制。
2. 替换策略的目标与原则
  - “管理哪个进程的哪个虚拟地址被分配给了哪个物理页”(どのプロセスがどの仮想アドレスがどの物理ページに割り当てられているかを管理)，OS 的页面替换策略必须在一个全局范围内工作，它需要知道当前所有进程的页面在物理内存中的分配情况。
  - “驱逐最不可能被使用的页面”(最も使われる可能性の低いページを追い出す)，由于理想的 LRU (Least Recently Used, 最近最少使用) 算法需要昂贵的硬件开销来追踪所有页面的精确使用时间戳，OS 通常会使用一种近似或模拟的 LRU 算法。这些算法通过使用页表中的\*\*引用位 (Reference Bit)\*\* 等机制，以较低的软件开销来近似达到 LRU 的最佳效果。
3. 被替换页面的去向
  - “被驱逐的页面去往交换区”(追い出されたページはスワップ領域へ)

## 页表 (Page Table) 的尺寸是如何由地址空间和页面大小决定的

问题 1：在以下参数下，页表的大小是多少？

参数：

- 逻辑地址 (Logical Address): 32 位 (bit)
- 页面大小 (Page Size): 4KB (4096 字节)
- 页表各条目大小 (Page Table Entry Size, PTE Size): 4 字节 (Byte)

4MB (bytes=8bits)

问题 2：如果是逻辑地址 64 位的情况呢？

这是在问，如果系统升级到 64 位，页表的大小会如何变化。

16PB

## 提出的核心难题：如何减小页表 (Page Table) 的巨大尺寸

1. 动态增长 (增量式)：“每当进程使用新的虚拟页时才增加条目”(プロセスが新しい仮想ページを使用するたびにエントリを増やす)
  - “需要保证新地址的获取方向是固定的”(新しいアドレスの取得方向が一定である必要がある)
  - “通过 2 个段 (Segment) 进行管理：栈和堆”(2セグメントで管理 スタックとヒープ)
2. 使用散列 (Hash)
  - “使用散列 (Hash) 使条目数与物理页数相同”(ハッシュを用いてエントリ数を物理ページ数と同じにする)
  - 具体实现：逆向页表 (逆引きページテーブル - Inverted Page Table)
  - 逆向页表只为\*\*每个物理页框 (Page Frame) \*\*设置一个条目。它存储的是 物理页号 → 虚拟页号 的映射。
3. 多级化 (分层页表)
  - 结构：“将高位地址分成几块，分别持有指向表的指针 → 在最下层的表中存储页地址”(上位アドレスをいくつか区切り、それぞれテーブルへのポインタを持つ\$rightarrow\$最下位のテーブルにページアドレスを格納)
  - 缺点：“表查找需要多次内存访问”(テーブル引きに複数回のメモリアクセスを要することになるが、使用率の低いページテーブルを圧縮)
  - 查找一个地址可能需要 3 到 4 次内存访问 (从根目录查到一级表，再到二级表...)。
  - 优点/补救：“如果有后面将提到的 TLB，则缺点减轻”(後述するTLBがあればデメリット軽減)

## TLB (Translation Lookaside Buffer)

1. TLB 的核心定位：页表的缓存。

- 这是TLB最简洁的定义。页表（Page Table）是存放在内存中的，TLB则是存放在CPU内部的一个小型、高速的缓存，用于存放页表中最近使用过的条目。
- “页表是内存数据，所以每次内存访问都需要两次访问”（ページテーブルはメモリデータなのでメモリアクセスのたびに二回アクセスすることになる）
- “隐藏这个开销”（このオーバーヘッドを隠蔽），TLB的主要目标就是通过高速缓存这些映射结果，来隐藏掉第一次访问页表所产生的巨大开销。

## 2. 内存访问的完整流程（有了 TLB 之后的理想流程）

- 逻辑地址计算（論理アドレス計算）：程序计算出需要访问的逻辑地址。
- 页表访问（ページテーブルアクセス）：首先检查 TLB。如果 TLB 命中，直接跳到下一步。如果 TLB Miss，才需要真正访问内存中的页表。
- 内存访问（メモリアクセス）：使用转换得到的物理地址访问主内存获取数据。
- 结论：在 TLB 命中的情况下，地址转换和数据访问几乎是同时完成的，性能接近于没有虚拟内存。

## 3. TLB 的结构特性

- “与缓存相同的结构”（キャッシュと同様の構造）
- “用虚拟地址进行相联搜索”（仮想アドレスで連想検索），为了追求极高的速度，TLB 通常是全相联（Fully Associative）或高相联度组相联的。CPU 需要同时检查 TLB 中的所有或多个条目（Way），看哪个条目的标签（Tag，即 VPN）与当前请求的地址匹配。
- “保持最近使用的地址的标签和页表条目”（最近使われたアドレスについてタグとページテーブルのエントリを保持）TLB 存储的内容包括：
  - 标签（Tag）：对应于虚拟页号（VPN）。
  - 页表条目（Entry）：对应于物理页号（PPN）以及权限位、有效位等信息。

## TLB 的动作 (TLBの動作)

- 搜索阶段 CPU 提取逻辑地址中的虚拟页号（VPN），并将其作为搜索键，在 TLB 中进行高速的相联搜索（Associative Search）。
- 结果路径 I：命中（Hit）“标签一致 → 命中 → 高速获取物理地址”（タグ一致 → ヒット → 高速に物理アドレス取得）
- 结果路径 II：未命中（Miss）如果 TLB 中没有找到与请求 VPN 匹配的条目，则发生 TLB Miss。系统必须进行额外的操作来获取地址映射信息。
  - “判定是 TLB Miss 还是 Page Fault”（TLBミスなのか、ページフォルトなのかの判定）
  - “向页表询问”（ページテーブルに問い合わせ）
  - “如果只是单纯的 TLB Miss，则将信息加载到 TLB”（単純なTLBミスであればTLBに情報をロード）
  - “如果是 Page Fault，则产生异常”（ページフォルトであれば例外），触发Page fault

TLB 的工作流程是一个精细的分流系统：

```
 $$\text{逻辑地址} \rightarrow [\text{TLB搜索}] \{ \begin{cases} \text{命中} & \rightarrow \text{高速访问内存} \\ \text{未命中} & \rightarrow \text{查询页表} \end{cases} \} \begin{cases} \text{页表命中} & \rightarrow \text{更新TLB, 重试, 高速访问内存} \\ \text{页表未命中} & \rightarrow \text{Page Fault, OS介入 (调入硬盘数据)} \end{cases} $$

```

## 与缓存的集成 (キャッシュとの統合)

既然 CPU 访问数据需要经过地址转换（虚拟地址 → 物理地址），而数据本身又存储在**高速缓存Cache**中，那么必须决定：应该用哪种地址（虚拟地址还是物理地址）来对缓存进行索引和标记？

1. 逻辑地址缓存 (論理アドレスキャッシュ), (論理アドレスによって連想検索 - 用逻辑地址进行相联搜索) 缓存直接使用 虚拟地址 (或逻辑地址) 来进行索引和查找 (即图C所示的 VI/VA Cache)。但是存在同义词问题。
  2. 物理地址缓存 (物理アドレスキャッシュ) (アクセス前に物理アドレスを取得する必要がある - 在访问前必须获取物理地址)
  3. 折衷方式 (折衷方式 / VIPT Cache) 这是现代处理器为了兼顾速度 (逻辑地址) 和一致性 (物理地址) 而采取的一种巧妙的折衷方案，通常被称为 VIPT (Virtually Indexed, Physically Tagged) Cache，即虚拟索引、物理标签缓存。
- 索引部分：(インデックスづけに論理アドレス - 用逻辑地址进行索引) 缓存使用 逻辑地址的低位部分 (页内偏移) 来确定缓存所在的 组 (Set)。因为这部分与物理地址的页内偏移是相同的，所以可以立即开始查找，无需等待 TLB。这是为了提高速度
  - (タグは物理アドレス (キャッシュアクセスと並行してTLBによる解決) - 标签是物理地址 (与缓存访问并行，由 TLB 解决))，在缓存进行索引 (Set 查找) 的同时，TLB 也并行地进行地址转换。当缓存和 TLB 都完成工作后，缓存取出的 标签 (Tag, 这里必须是物理地址) 与 TLB 转换出的 物理页号 (PPN) 进行比较。保证一致性

总结了缓存设计在虚拟内存环境下的三种选择：

1. 逻辑地址缓存 (VA Cache) : 速度最快，但有同义词问题。
2. 物理地址缓存 (PA Cache) : 速度慢，但管理简单，无同义词问题。
3. 折衷/VIPT 缓存：结合了两者的优点，通过并行查找来达到高速且安全的目的，是目前最常用的 L1/L2 缓存设计方案之一。

## 虚拟内存带来的保护 (仮想記憶による保護)

虚拟内存带来的保护是 软件 (OS) 和硬件 (Dual Mode/MMU) 协同作用 的结果：

1. 软件 (OS) : 划清界限，通过页表将每个进程隔离在自己的逻辑地址空间内。
2. 硬件 (Dual Mode) : 确保隔离不被破坏，通过“只有监控者模式才能修改页表”的机制，防止用户程序篡改地址映射规则来窃取或破坏其他内存。

## I/O

---

I/O 设备的功能范围 (与非内存元素的交互) :

- “与主存储器以外的元素进行交互” (主記憶以外の要素とのやりとり)
- 与辅助存储器交互 (補助記憶とのやりとり):
- 与系统外部交互 (システム外とのやりとり (ネットワーク I/F など)):

I/O 设备的物理连接方式：

- “物理上一般利用 I/O 总线” (物理的には一般に I/O バスを利用)
- “经过处理器的 I/O 引脚” (プロセッサの I/O ピンを経由)

- 主存储器连接到内存总线的引脚”(主記憶はメモリバスのピンに接続)

这页PPT总结了 I/O 设备在计算机系统中的作用和位置：

- 功能：它们负责处理所有与 CPU/主内存核心之外的通信。
- 物理结构：它们通过I/O 总线连接，与连接主内存的内存总线是分离的。

## I/O 设备的工作 (I/O デバイスの動)

### 1. I/O 设备的软件/硬件构成

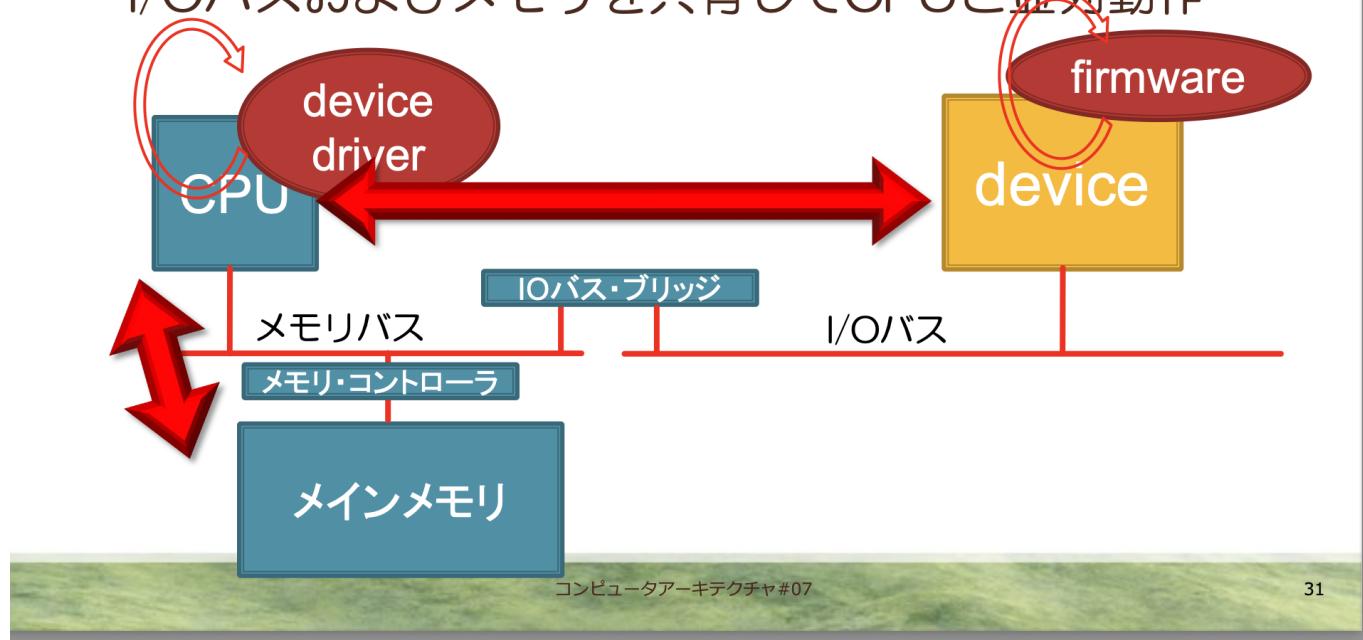
- “各设备由控制器驱动”(各デバイスではコントローラが動作)读：每个 I/O 设备 (device, 黄色方块) 都有其自己的控制器（通常是一个专用芯片）。
- “执行驱动设备的程序”(デバイスを動かすプログラムを実行)固件 (firmware, 红色椭圆): 是设备的底层控制程序。例如，CPU 向硬盘发送一个“读取”命令，是固件负责将这个命令翻译成硬盘磁头的物理运动

### 2. CPU 与设备的协作：

- “共享 I/O 总线和内存，与 CPU 并行工作”(I/O バスおよびメモリを共有して CPU と並列動作)

## I/Oデバイスの動作

- 各デバイスではコントローラが動作
  - デバイスを動かすプログラムを実行
- I/Oバスおよびメモリを共有してCPUと並列動作



图解中的关键路径和组件：

### 1. CPU (中央处理器):

- 负责运行应用程序和操作系统内核。

- **Device Driver (设备驱动程序, 红色椭圆):** 驱动程序是 OS 内核的一部分, 运行在 CPU 上。它是操作系统与硬件设备之间通信的软件桥梁。当应用程序请求 I/O 时, 驱动程序将请求转换为硬件控制器能理解的命令, 并发送给设备。

## 2. 总线和桥 (バスとブリッジ):

- **内存总线 (メモリーバス):** 连接 CPU 和内存控制器。
- **内存控制器 (メモリー・コントローラ):** 管理主内存 (Main Memory) 的数据读写。
- **I/O 总线 (I/O バス):** 连接各种外围设备。
- **I/O 总线/桥 (I/O バス・ブリッジ):** 连接内存总线和 I/O 总线, 允许 I/O 设备与主内存进行数据交换。

## 3. 数据/控制流 (红色箭头):

- **CPU \$\rightarrow\$ Device Driver \$\rightarrow\$ I/O 总线 \$\rightarrow\$ Device:** 这是 **控制命令** 的路径。CPU 运行驱动程序, 将 I/O 命令 (例如, “读取文件”) 写入设备控制器的寄存器。
- **Device \$\rightarrow\$ Firmware:** 设备接收到命令后, 其**固件**开始执行实际的物理操作。

## 4. CPU \$\leftarrow\$ Main Memory:

- 这是 CPU 的主要工作区域, 通过高速内存总线进行数据交换。

## 5. Device \$\rightarrow\$ Main Memory:

- 图中未直接画出, 但 I/O 设备通常通过 **DMA (Direct Memory Access)** 机制, 直接通过 I/O 总线桥与主内存交换大量数据, 绕过 CPU, 实现并行高效传输。

## I/O 设备的协作模型:

1. 软件控制 (CPU/Driver): CPU 运行设备驱动程序, 发送高层命令。
2. 硬件执行 (Device/Firmware): 设备控制器和固件接收命令并执行物理 I/O 操作。
3. 并行性: I/O 操作与 CPU 运算同时进行, 通过独立的 I/O 总线与内存总线分离。

## 内存映射 I/O (メモリマップド・I/O)

MMIO 是一种将 I/O 设备的操作集成到 CPU 内存访问指令集中的技术 (Memory mapped I/O)

- 核心机制: 地址空间分配, “将内存地址的特定区域分配给各个 I/O 控制器” (メモリアドレスの特定領域を各 I/O コントローラへ割り当てる)
- 最终效果: 简化编程, “从用户角度看, 可以用与内存访问相同的操作访问设备的特殊寄存器” (ユーザから見るとメモリアクセスと同じ操作でデバイスの特殊レジスタへアクセスできる)

内存映射 I/O 是一种高效且简洁的 I/O 机制。它通过将 I/O 设备的寄存器映射到 CPU 的地址空间, 使得 CPU 可以使用通用的内存读写指令来控制 I/O 设备, 从而简化了体系结构和软件设计。

## 设备驱动程序 (デバイスドライバ) (Device Driver)

设备驱动程序是一段运行在操作系统内核中的软件代码, 它是操作系统 (软件) 和 I/O 设备 (硬件) 之间的翻译官和通信接口。

### 1. 驱动程序的核心功能:

- “向设备发送指令的 CPU 程序” (デバイスへの指令を送る CPU プログラム), 驱动程序本身是 CPU 运行的一段程序。它接收来自上层操作系统或应用程序的 I/O 请求 (例如, “打印文件”), 并将这些高层请求转化为设备控制器能理解的低层命令。
- “与固件通信” (ファームウェアとお話しする), 驱动程序实际上是在与设备控制器内部的\*\*固件 (Firmware) \*\*进行通信。驱动程序发送命令到设备的寄存器; 固件则读取寄存器中的命令并执行实际的物理 I/O 操作。

## 2. 驱动程序如何与硬件通信

- “读取和写入设备内的寄存器” (デバイス内のレジスタを読み書き)
- 解读: 驱动程序不直接操作设备本身, 它通过访问设备控制器内部的特殊寄存器来控制设备。
- 通信机制: (I/O 命令またはメモリマップド I/O 経由)专用的I/O指令集或内存映射

### ■ デバイスドライバ

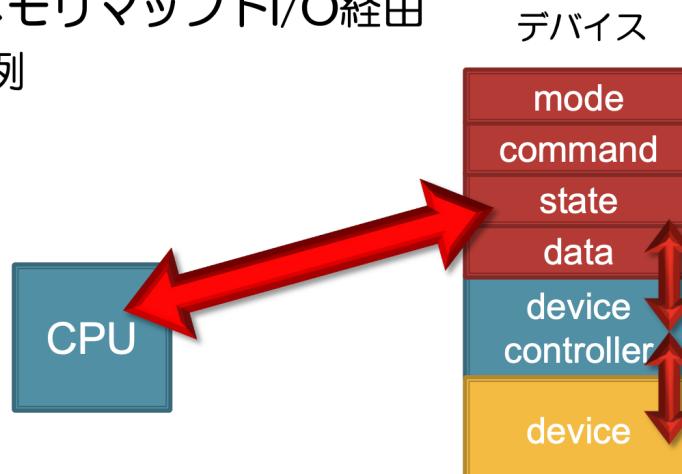
#### • デバイスへの指令を送るCPUプログラム

- ファームウェアとお話しする

#### • デバイス内のレジスタを読み書き

- I/O命令またはメモリマップドI/O経由
- レジスタの内容例

- 動作モード
- コマンド
- ステート
- データ



コンピュータアーキテクチャ#07

33

## CPU 与控制器的协作 (CPUとコントローラの協調)

- 解读: 核心问题是: I/O 设备 (如硬盘) 的速度比 CPU 慢上万倍, CPU 如何在等待 I/O 完成时不浪费时间? 以及数据如何高效传输?

设备运行速度比CPU慢, CPU 需要等待:

- 如何知道请求处理何时完成? - 轮询/中断
- 等待请求处理完成恢复程序? -Synchronous I/O/ Asynchronous I/O

解决数据传输的问题: 协作接口, 这是 CPU 或 OS 用来指导数据如何在内存和设备之间移动的两种主要机制。

- PIO (Programmed I/O - 程序控制 I/O)
- DMA (Direct Memory Access - 直接内存存取)

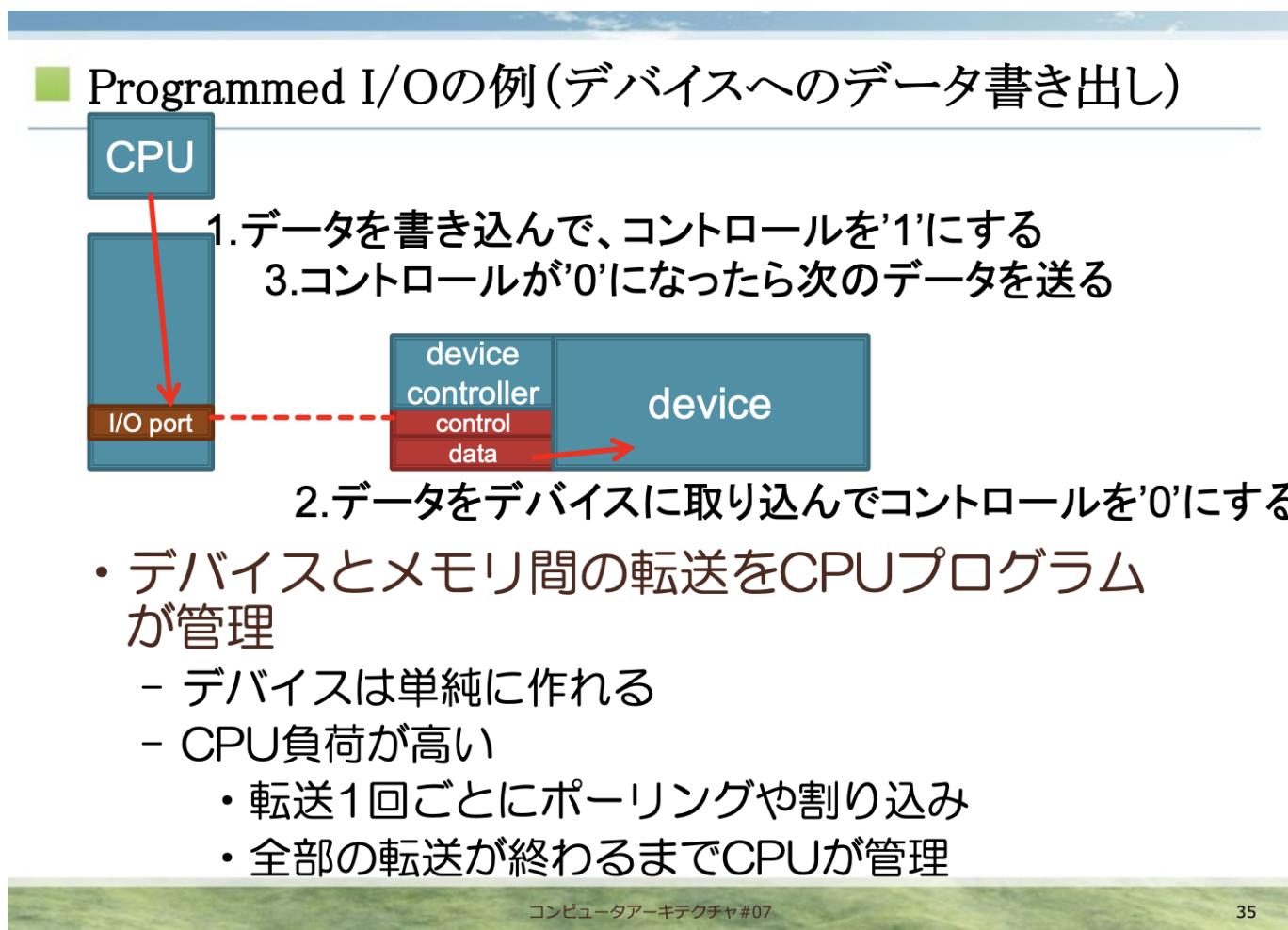
高性能 I/O 系统的设计方向:

- 通知机制: 从浪费资源的轮询, 转向高效的中断。
- 数据传输机制: 从低效的 PIO (CPU 搬运数据), 转向高效的 DMA (让控制器自己搬运数据)。
- 中断 + 异步 I/O + DMA

## 程序控制 I/O (PIO) 的例子 (向设备写入数据)

特性:

- “CPU 程序管理设备和内存之间的数据传输” (デバイスとメモリ間の転送を CPU プログラムが管理)
- 优点: “设备可以简单地制造” (デバイスは単純に作れる)
- 缺点: “CPU 负载高” (CPU 負荷が高い), “每次传输都需要轮询或中断” (転送 1 回ごとにポーリングや割り込み), “直到所有传输结束, 都由 CPU 管理” (全部の転送が終了するまで CPU が管理)



## DMA (Direct Memory Access)

- DMA 的核心价值: “削减高速设备的中断开销” (高速デバイスの割り込みオーバーヘッド削減)
- 解读: DMA 的主要目标是提高 I/O 效率。在 PIO 方式中, 即使使用中断, CPU 也可能需要为每传输一个字 (Word) 的数据而处理一次中断或轮询。对于高速设备 (如 SSD、千兆网卡), 这种频繁的中断或轮询会产生巨大的 CPU 开销 (Overhead)。DMA 极大地减少了这种开销。

2. DMA 的工作流程

- “从 CPU 向 DMA 控制器发送命令” (CPU から DMA コントローラへコマンド送信)
  - DMA 命令包含三个关键信息：
    1. “发送源的起始地址” (送り元の先頭アドレス)
    2. “发送目标的起始地址” (送り先の先頭アドレス)
    3. “传输量” (転送量)
  - “传输中 CPU 独立工作” (転送中は CPU は独立して動作)
3. DMA 的效率体现 (中断开销的对比)
- “1 Byte / 中断 → 1 Block / 中断” (1 Byte / 割り込み → 1 Block / 割り込み)
  - 效率提升：如果一个数据块是 4KB，那么 DMA 将中断频率降低了 4096 倍，极大地释放了 CPU 资源。

DMA 是提高 I/O 效率的关键技术。它将数据搬运这一繁重任务从 CPU 手中转移到了 DMA 控制器手中，实现了：

- 并行性：CPU 运算与数据传输同时进行。
- 低开销：极大地减少了 CPU 必须处理的中断和轮询次数。

The Author is 11\_sum