廖威雄: 学习Linux必备的硬件基础一网打尽

LinuxDev 2019-07-29 阅读数：3584

作者简介：
廖威雄，目前就职于珠海全志科技股份有限公司从事linux嵌入式系统(Tina Linux)的开发，主要负责文件系统和存储的开发和维护，兼顾linux测试系统的设计和持续集成的维护。
拆书帮珠海百岛分舵的组织长老，二级拆书家，热爱学习，热爱分享。

内容简介：
出来混，迟早要还的.....
本文详细论述大学时候的基本功，MMU,CACHE, TLB，Page Fault, 进程切换.....

1. 案情回溯

某一个夜黑风高的晚上，宋宝华老师组织的微信群一片祥和宁静。我刷着手机，心血来潮往微信群中提出一个疑惑：

" 宋老师，cache是一个程序维护一份还是所有程序维护一份？你昨天说进程调度损耗还包括cache命中的问题，那如果一个进程一份cache不就好了？ "

正所谓一石激起千层浪，这个傻逼竟然问这么弱智的问题？!宋老师急怒攻心，就差扯着我的衣领大吼：

" 别问这么傻的问题了看书！！你这样真吐血！ "

在宋老师的淫威之下瑟瑟发抖，于是狠下心来，翻箱倒柜找出了封印多年的《计算机操作系统》，于是便有了此文。

本文尽可能从基础部分讲起，适合完全没学过计算机操作系统的小白，也适合乱七八糟学过一些但没理清脉络的读者；不适合完全掌握计算机操作系统原理的大牛。

本人水平有限，若存在理解错的地方，欢迎提出。

2. 存储结构

2.1. 经典存储结构图

图 2-1经典存储结构图

结构图要点解析：

a. 从L0到L6的7个层次，都是硬件设备，并不是软件的概念。尤其注意图中的高速缓存，常称为CPU cache，与我们常说的Page cache是不一样的，前者是实在硬件设备，后者是软件上的概念

b. 金字塔越往上，速度越快，价格也越高。出于成本考虑，实际产品中价格越高的存储器实际使用容量越小，例如寄存器以个计数，1级缓存单位是KB，3级缓存单位是MB，内存几GB，最底层的磁盘却可达到TB级别

c. 磁盘是非易失性存储器，而磁盘以上的都是易失性存储器。这也意味着掉电后，磁盘数据依然存在，。于此相反，内存是易失性存储器，掉电则数据丢失。

d. 上层存储器的数据来自下层，且是直接下层，例如L1高速缓存的数据直接来自于L2高速缓存

结合b、c和d三点，我们初步形成一个认识：

“每次从上电开始，只有磁盘有数据。CPU都是从磁盘读取原始数据，然后根据一些算法挑选数据往上传递，例如从磁盘选部分数据放到内存，从内存提取部分数据到高速缓存。”

2.2. 本章总结图

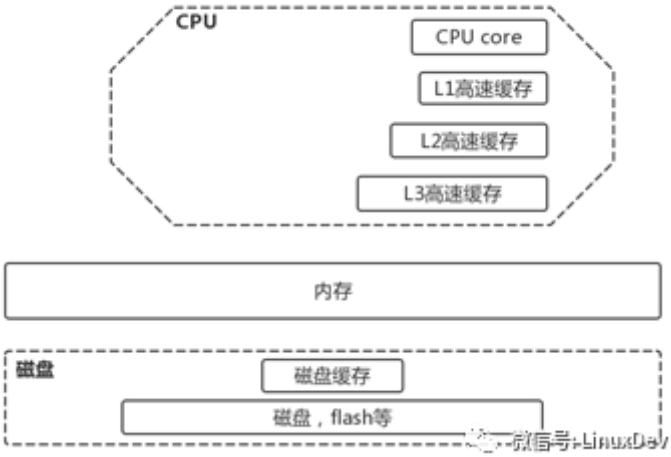


图 2-2总结图1

Ps. 后面章节逐渐完善此图，让章节之间，知识之间构建联系。

3. 什么是页？什么是页表？什么是快表？什么又是MMU

3.1. “页”是什么?

在了解"页"之前，务必初步了解"虚拟存储器"。

3.1.1. 虚拟存储器

那么，什么又是虚拟存储器呢？《计算机操作系统 第三版》有这么一句话：

> “(虚拟存储器是)具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储系统”

本文关注的重点是：“从逻辑上对内存容量加以扩充”。什么是逻辑上扩充内存容量呢?
例如，在旧一点的32位cpu的手机中有且只有2G内存，但是每一个进程在执行过程中都仿佛手机有4G内存，且每个进程都能独占4G内存。注意了，是每一个进程都独占4G内存，每一个！每一个！
这就是从逻辑上对内存进行扩充，但这也是赤裸裸的欺骗。
实际物理内存明明只有2G，页表的机制让每个进程都仿佛有4G。进程被欺骗了，但进程可爽了。为什么爽呢？对应用来说，你告诉我有4G内存啊，我不管你实际有多少，我需要用到4G的内存时，你要给到我！于是内核就苦逼了。
我们把APP访问到的4G虚拟内存地址叫做：虚拟地址
我们把内核实际管理的2G物理内存地址叫做：物理地址
先形成这样一个初步认识：CPU访问虚拟地址，MMU把虚拟地址转为物理地址，CPU再通过物理地址访问物理内存。（Ps. MMU是CPU中的一个硬件模块，硬件模块！硬件模块！专门负责虚拟地址到物理地址的转换，下文会有更详细的介绍）

3.1.2. 页

清楚了什么是虚拟地址，什么是虚拟存储器，我们再来看看，“页”是什么?

> “把APP的虚拟地址空间切分为若干个大小相等的片，每一片，就是我们说的页（Page）”

注意，这里说的是虚拟地址空间，虚拟地址空间！虚拟地址空间！在Linux上，页大小通常为4K。每一个页都有编号，从0开始，如第0页，第1页等。

> ” 把物理内存也按"页"的大小切分为若干大小相等的片，每一片，就是我们说的页框（frame） “

页框就是用来装页的呀，就像相框用来装相片，务必大小一致!
页框也叫页帧、物理块(≠I/O概念上的块)，由于契合相框的形象，本文统一称为页框。
因此，我们说：

> “页是内存管理的基本单位”

内存管理就是把物理内存划分为一个一个的页(框)，进程在申请内存时，内核就以页为单位，把进程的一个个（虚拟）页装入到多个可能不连续的（物理）页框中。

我们以一张图来形象描述页和页框的关系：

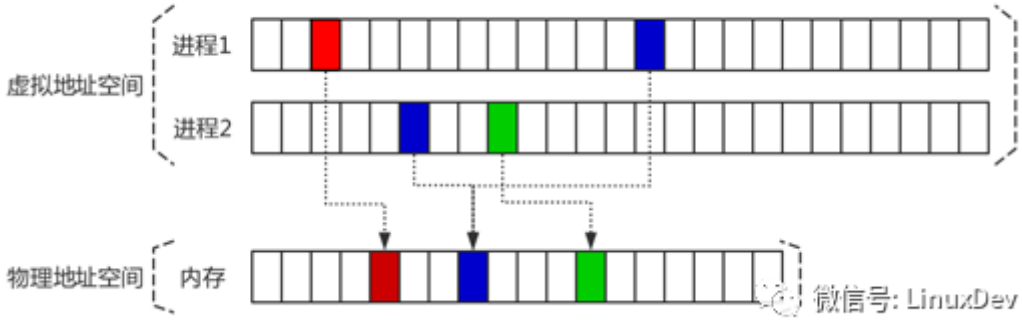


图 3-1页与页框

上图除了描述页与页框的关系，还隐藏了两个重要的信息：

a. 一个进程的页并不是全部映射到物理内存页框中

Linux内核只有在万不得已得情况下才会真正为进程分配物理内存。例如进程1中的白色部分，由于进程还没使用，因此白色部分并不会指向物理内存的页框。

b. 多个进程的页可以映射到共同的物理内存页框中

上图中进程1与进程2共用蓝色页框。这些共享的页框一般是动态库，例如所有进程使用相同的C库指令。物理内存空间有限，内核没必要为每个进程单独维护一份C库指令。

3.2. 页表是什么?

3.2.1. 页表

从页与页框的图，我们知道虚拟的进程页与物理内存页框有映射关系，而且是离散映射的。那么是什么记录了这种映射关系？是页表。页表记录了“页”与“页框”的对应关系。

我们以一张图来形象描述页表：

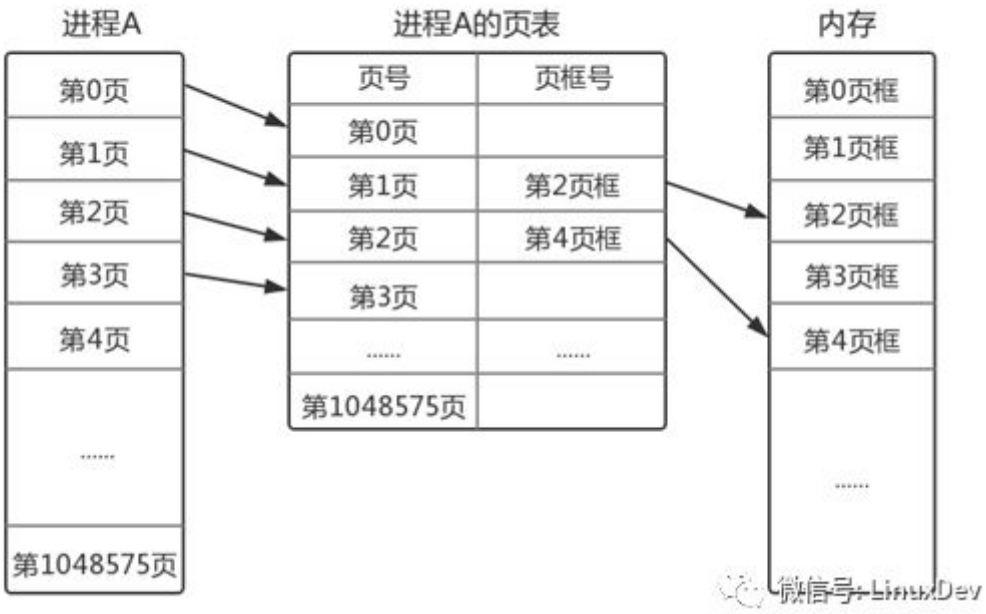


图 3-2页表

上图有以下几个要点：

a. 内核为每一个进程维护一份页表，一对一！

b. 页表记录了进程页与内存页框之间的对应关系。

c. 进程的页离散存储在物理内存的页框中。例如进程第1页存放在物理第2号页框中，第2页却放在第4页框。

d. 只有在万不得已情况下，内核才真正为进程页分配物理内存。例如第0页，第3页因为进程还没真正使用，是没有页框号的。

e. 进程的页，不管有没使用，都完整记录在页表中。例如从第0页到最后的第1048575页，都记录在进程A的页表中。

3.2.2. 多级页表

是否思考过上图最后一页为什么是第1048575页?

以32位CPU有4G虚拟内存为例，以linux的4K页大小计算：

```
虚拟页数量 = 4G/4K = 1M ( 1048576)
```

从0开始算，因此最后一个页号是1048576 - 1 = 1028575。

一个页表项（就是页表的一行）通常只占4Byte，因此描述一份4G虚拟地址空间大小的页表，就有1048576*4B=4M。而且由于页表是一个表的结构，且换算页表项地址是通过表内偏移实现，因此存放页表的物理页框必须连续。也就是说，一个进程还没开始跑就要先占用4M的内存存放页表？这么浪费，还要是连续物理地址的内存？不能忍！为了解决此问题，于是设计了2级页表，甚至多级页表。

多级页表，简单来说就是对那1级页表需要的4M连续内存再次分页，再使用另外一个页表保存分页的记录，内核只需要记录最上一层的页表，通过多次页表的转换，取得真正的物理页框号。因此，最上一层的页表就可以很小，且支持离散存储。

篇幅有限，不展开多级页表，只需要了解个概念即可，即使不理解也不影响本文的讨论。

3.3. “快表”是什么?

页表存在于内存中，咱们暂时忽略高速缓存，那么CPU每次获取数据都需要2次访问内存：

a. 第一次访问内存的页表，获取物理地址

b. 第二次根据物理地址获取真实数据

内存访问速度虽然比磁盘要快，但跟CPU比依然不在一个数量级，CPU得浪费非常非常多时间等待内存。

为了提高极致提高访问页表的速度，于是有了快表。

快表，又名TLB（Translation Lookaside Buffer），是单独的一组寄存器，也有说是高速缓存，用于记录页表中正在频繁使用的页表项。

> “由于程序和数据的访问往往具有局限性（局部性原理），因此，据统计，快表命中概率可达90%以上”

局部性原理是什么？后面还会有更详细的介绍。在这里，我们要有个认识即可：

快表是为了从概率命中的角度加速虚拟地址到物理地址的转换，设计的一组高速存储硬件模块。

1个CPU多核之间共享Cache，但是一个核却有独占一个快表。

由上一章《存储结构》我们知道， 寄存器的速度最接近于CPU，或者说，完全跟得上CPU的节奏，但很贵，因此：

- 快表容量有限，通常只能记录16~512个页表项
- 转换虚拟地址到物理地址时，优先从快表寻找
- 如果快表中找不到对应页表项，则从内存页表中获取，同时更新快表
- 更新快表时，如果快表已满，则淘汰一个最老的且已被认为不再使用的页表项

3.4. “MMU”是什么?

本章到这，我们已经知道，什么是"页"，什么是"页表"，什么是"快表"，也知道了他们都是为了从虚拟地址向物理地址转换服务。但我们似乎忽略了一个问题，谁来负责执行这项频繁的地址转换工作？CPU？不不，CPU已经够忙的了。这就到了我们勤勤恳恳的MMU表现的时候了。

MMU（MemoryManagememt Unit），内存管理单元。是CPU内的一个硬件模块，硬件！硬件！负责从虚拟地址转换物理地址，并提供硬件机制的内存访问权限检查。

MMU转换虚拟地址到物理地址是其内的硬件电路做的，硬件电路！硬件电路！

我们需要形成个认识即可，下文会有更详细的介绍：

CPU需要访问进程的虚拟地址，于是把虚拟地址发送给MMU，由MMU内的硬件电路实现访问快表、页表，最终得到的物理地址。

3.5. 本章总结图

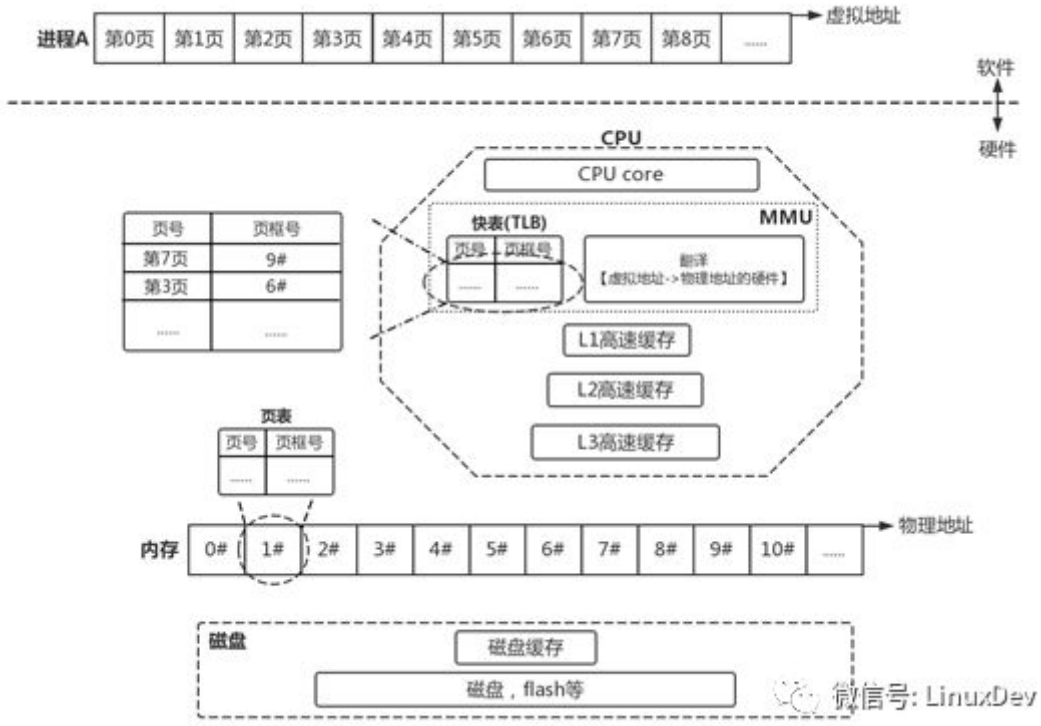


图 3-3总结图2

Ps. 请试试对比总结图1，并结合本章知识解剖此图

Ps. 此处的CPU core是指运算单元，而非类似Cortex-A7等 的core，因为MMU、TLB、CPU Cache等也归属于Cortex core。

4. 什么是页面调入和页面置换？如何置换页？

4.1. 什么是页面调入和页面置换?

4.1.1. 页面调入

在第3章"什么是页"的探讨中提到：

Linux内核只有在万不得已得情况下才会实际为进程分配物理内存

这个结论同样适用于开始执行程序时的加载程序，例如一个可执行程序100M，这100M指令、数据必须要加载到内存CPU才可访问。但是物理内存容量有限，不可能为了这个程序，一执行立马为程序分配100M物理内存，否则你多执行几个程序不就没物理内存了么?

因此，这100M的程序只是逻辑上"加载"到了虚拟内存空间，为数据指定了虚拟页号，实际上还没为进程的页分配物理页框。在程序指令需要CPU读取这100M数据时，内核才"磨磨蹭蹭"的真正分配物理页框，而且是程序要什么内核才加载什么，且一次只加载一部分。

这个从下层获取进程页面的操作就叫"页面调入"。

4.1.2. 页面置换

在第2章《存储结构》中有提到，实际产品中，越上层价格越贵，因此出于成本考虑，实际使用容量越小。因此，上层只能从下层挑选一部分页通过页面调入加载到上层。

挑选什么数据加载到上层？这是由一些算法决定的，例如常见的LRU算法。算法下文继续讨论，本节我们继续探讨，什么是页面置换?

上层存储容量有限，如果上层本身已经放满数据了，要调入新的页，只能淘汰旧的页。

这个淘汰旧页，拥抱新页的过程就是"页面置换"。

淘汰的旧页不都是直接丢弃，例如脏文件数据就会回刷，不常用的页会置换到交换分区等。

4.1.3. 缺页中断

CPU/内核中有2个需要调入页面和页面置换的地方：

a. 从磁盘加载数据到内存

b. 从内存加载数据到CPU高速缓存

前者是内核软件实现的置换算法，软件！软件！后者是CPU硬件实现的置换算法，硬件！硬件！

那么，我们所说的缺页中断是怎么一回事呢？就是内核提供的软件实现页面调入和页面置换的中断处理代码。

在进程所要访问的页如果不在内存时，由MMU触发一次缺页中断，执行内核提供的中断处理程序，将数据通过页面调入和页面置换的方式，从磁盘加载到内存。

由于上层的数据来自下层，且只是下层的一个拷贝，因此当内存都没有对应数据时，上层的高速缓存也不可能有这些数据。
行文到此其实已经暴露了我在《案情回溯》中的提问有什么不对。
我的提问：

" 宋老师，cache是一个程序维护一份还是所有程序维护一份？你昨天说进程调度损耗还包括cache命中的问题，那如果一个进程一份cache不就好了？ "

提问有哪里不对呢？

a. CPU cache是一个物理设备，而不是软件层面的Pace Cache类似的概念。因此cache不是用"份"的量词来描述的，这不是软件概念。
b. 一个CPU中多个核共用CPU cache，且cache的页面调入是物理算法实现的。硬件算法只挑选最活跃的页进入高速缓存，这不是软件所能控制的。

4.2. 如何置换页
4.2.1. 局部性原理
在探讨"如何置换页"之前，我们必须了解什么是"局部性原理"
程序运行有以下3个特点：

时间局部性：如果一个数据/指令正在被访问，那么在近期它很可能还会被再次访问
空间局部性：在最近的将来将用到的信息很可能与现在正在使用的信息在空间地址上是临近的
顺序局部性：在典型程序中，除转移类指令外，大部分指令是顺序进行的

简单来说，对同一个进程而言，CPU访问了某个逻辑地址的数据/指令，则CPU在将来很有可能再次访问这个虚拟地址或相邻的一小片连续地址，这就是局部性原理。
根据"局部性原理"我们知道什么地址的数据是CPU在将来的短时间内会访问的，因此我们就可以把这些数据缓存到更高速的存储设备中。
高速缓存容量有限，为了保证CPU尽可能从高速缓存中获取数据（内存速度<高速缓存），就是我们所说的提高Cache命令率，我们就要尽可能确保高速缓存中的数据是CPU将要用的。
同样的，内存容量有限，且远小于磁盘，但速度远大于磁盘，为了提高CPU获取数据的速率，我们就要尽可能保证内存中数据是CPU将要用的。
CPU优先从高速缓存获取数据，在高速缓存没有命中时，再去内存获取，如果内存也没命中，则必须从磁盘中调入。因此，页面置换尽可能保证：内存的数据是CPU将要用的，而高速缓存的数据来自内存，其数据是CPU很可能"立刻即刻马上"要用的。
因此，我们就需要根据"局部性原理"，淘汰Cache/内存中被认为CPU不再使用的页，腾出空间存放被认为马上要用的页。
4.2.2. LRU置换算法
目前常见的页面置换算法有3种：

- FIFO：先入先出，从时间维度，淘汰最早使用的页
- LRU：最近最少使用（LeastRecently Used），从时间维度，淘汰最久没用的页面
- LFU：最近最不经常使用（LeastFrequently Used），从频率维度，淘汰使用次数最少的

本文主要介绍LRU算法，假设场景如下：
刚上电，此时Cache是空的，数据全部在内存。根据LRU算法从内存获取数据到Cache。某个进程首先开始执行，其执行过程中需要访问到的指令/数据的虚拟地址所在的页顺序依次是：1，2，4，2，3，1
根据上述场景，Cache内数据的变化演示如图：

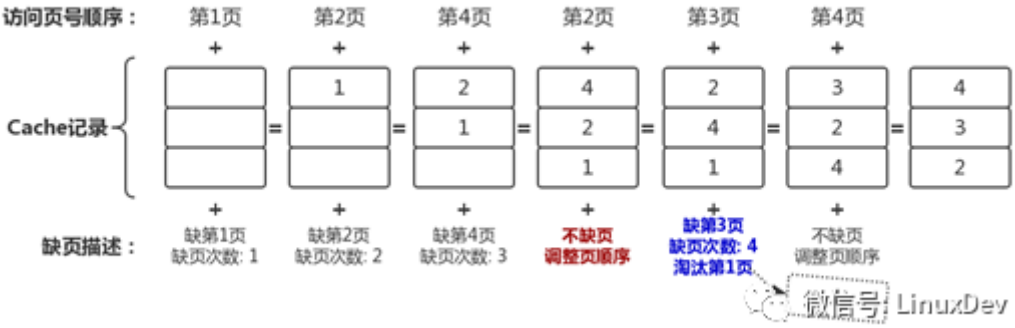


图 4-1 LRU算法演示

a. CPU执行进程指令，首先需要访问第1页，然而Cache是空的，无第1页缓存，因此触发第1次缺页。

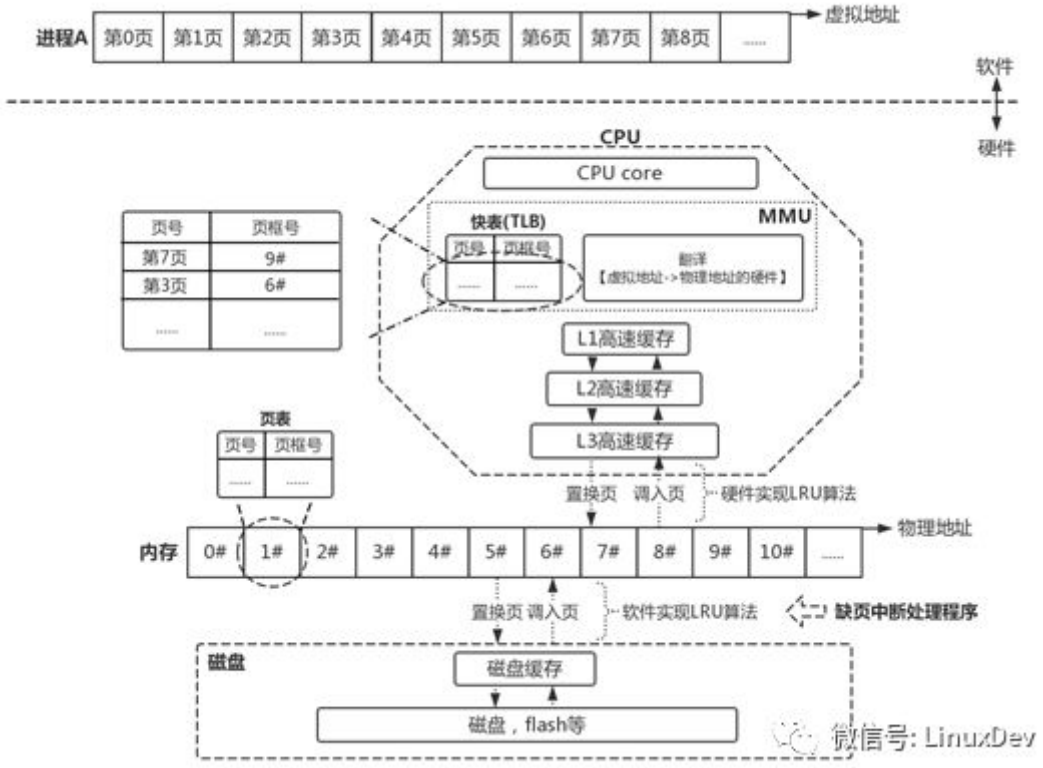
b. 接下来的第2页，第4页，都由于Cache没命中且Cache还有空余，触发第2、3次缺页，而不会淘汰页。

c. 第4次需要访问第2页，此时第2页已经在Cache中，Cache命中。不缺页，但会调整页顺序。

d. 第5次需要访问第3页，Cache不命中，触发缺页。但此时由于Cache已经满了，则按照LRU算法，淘汰最久没使用的页，因此淘汰第1页。

e. 第6次需要访问第4页，此时第4页已经在Cache中，Cache命中。不缺页，但会调整页顺序。

4.3. 本章总结图



5. CPU寻找页面的过程

CPU获取数据主要有两步：

a. 通过虚拟地址获取物理地址

b. 根据物理地址获取数据

本章主要分析这两步过程，其中【步骤a】-【步骤i】为获取物理地址的过程；【步骤A】-【步骤D】为根据物理地址获取数据过程。

5.1. 获取物理地址

参考图如下：

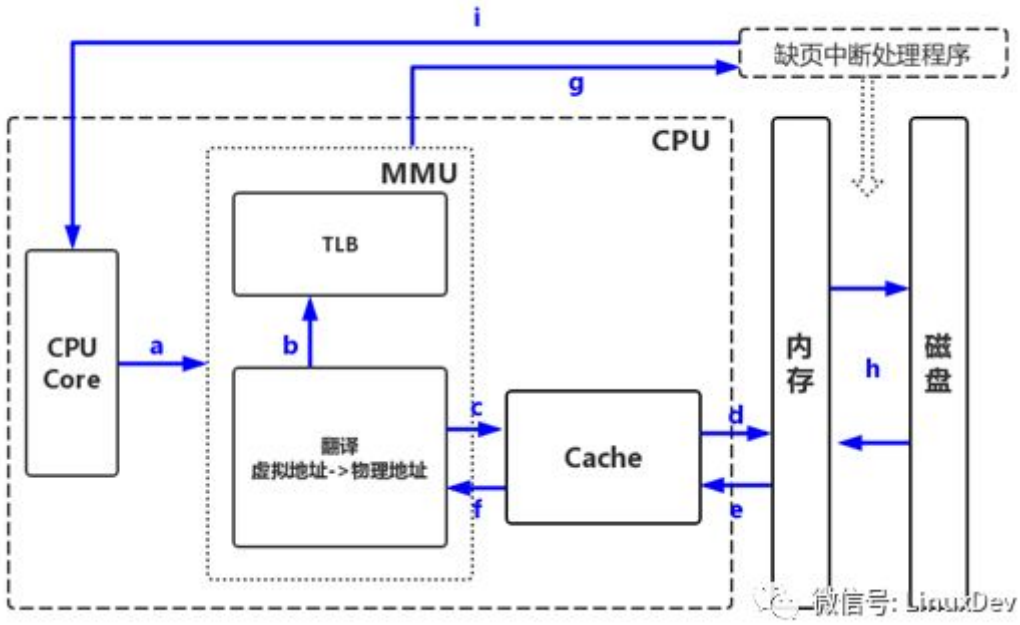


图 5-1获取物理地址

【a】CPU向MMU发送虚拟地址

【b】MMU查询快表TLB

快表命中：从快表获取物理地址，由MMU继续根据物理地址匹配数据页【步骤A】

快表不命中：查询页表【步骤c】

【c】从高速缓存查询页表

"高速缓存"缓存页框，而页表存储于页框中，因此读取页表实际上也是读页（页表的物理地址有单独的寄存器保存）

高速缓存命中：从高速缓存获取物理地址，并更新快表，进入【步骤A】

高速缓存不命中：查询内存中的页表【步骤d】

【d】从内存中查询页表

页表保存了所有虚拟地址的映射关系，而不管页是否有映射的物理页框。因此从内存页表中肯定能“命中”页。

从页表获取虚拟地址的映射关系后，进入【步骤e】

【e】把查询的页表项记录到高速缓存，进入【步骤f】

【f】MMU从高速缓存中获取页表项

虚拟地址有分配物理页框：转换物理地址并更新快表，由MMU继续根据物理地址匹配数据页【步骤A】

虚拟地址没分配物理页框：触发缺页中断【步骤g】

【g】进入缺页中断处理程序

【h】为虚拟页分配物理页框

如果无空闲物理页框，则根据LRU算法选择需要淘汰的页，要淘汰的页如果是脏页则回刷，否则直接丢弃或者置换到磁盘的交换分区中。

然后，从磁盘中获取页信息更新到物理页框，更新内存页表。

【i】缺页中断处理的最后

修改CPU寄存器，使得重新启动导致缺页的指令，返回【步骤a】重新执行。

5.2. 根据物理地址获取实际数据

参考图如下：

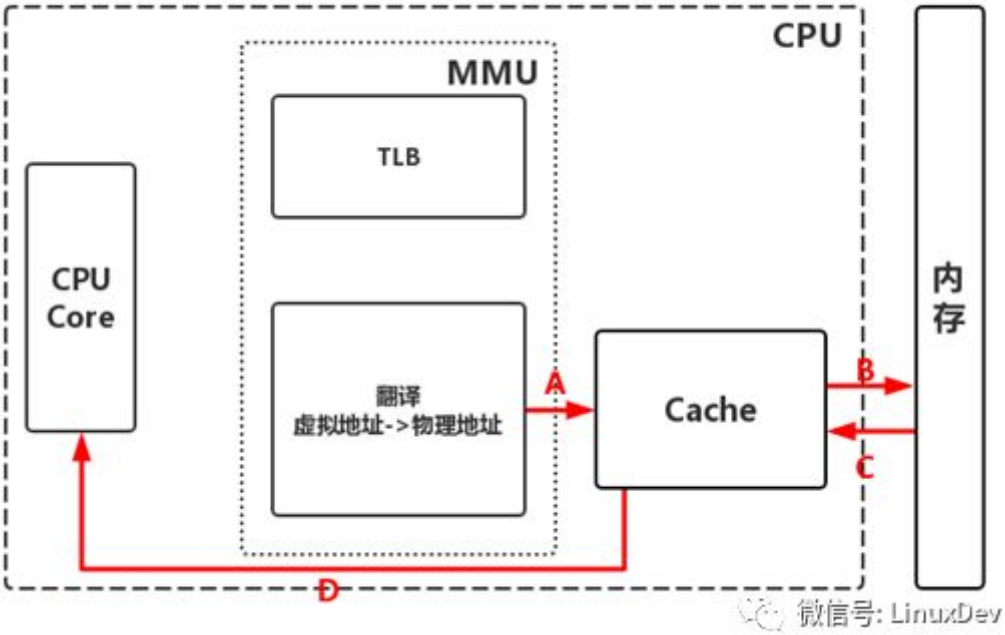


图 5-2根据物理地址获取数据

【A】MMU获取物理地址后，根据物理地址查询高速缓存

高速缓存命中：CPU直接从高速缓存获取数据【步骤D】

高速缓存不命中：查询物理内存【步骤B】

【B】根据物理地址，获取物理内存中的数据

【C】硬件实现把数据页记录到高速缓存中

【D】CPU直接从高速缓存中获取数据

5.3. 最理想（Cache和TLB命中）情况下寻找页面过程

大多时候都会命中，在命中时，CPU只需要通过高速缓存和TLB即可快速获取数据。

参考图如下：

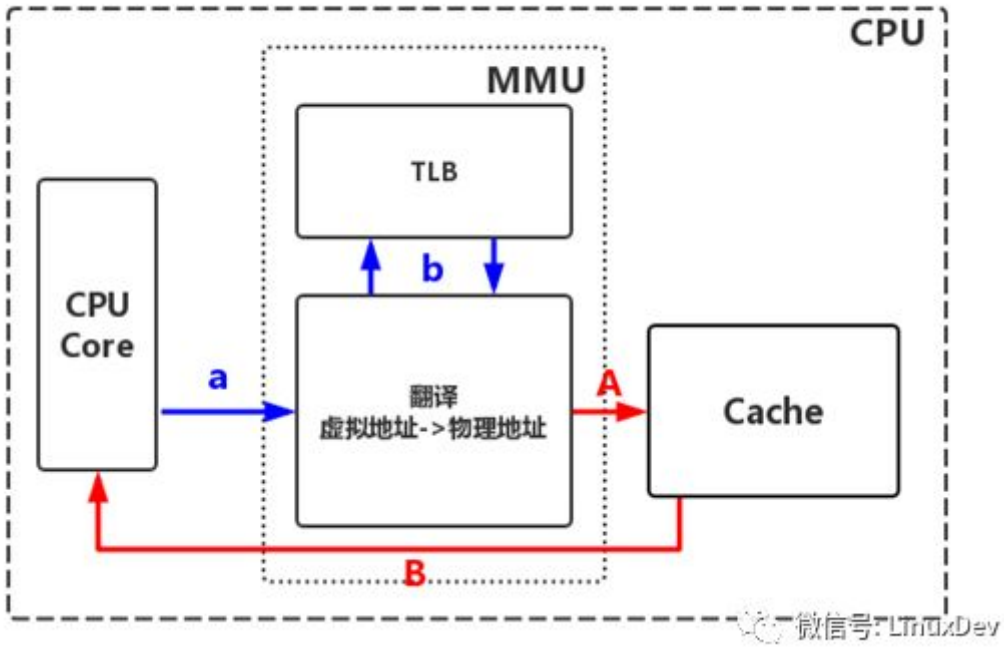


图 5-3最理想情况下CPU获取数据

【a】CPU向MMU发送虚拟地址

【b】MMU查询快表TLB

快表命中：从快表获取物理地址，由MMU继续根据物理地址匹配数据页【步骤A】

【A】MMU获取物理地址后，根据物理地址查询高速缓存

高速缓存命中：CPU直接从高速缓存获取数据【步骤D】

【D】CPU直接从高速缓存中获取数据

5.4. 本章总结图

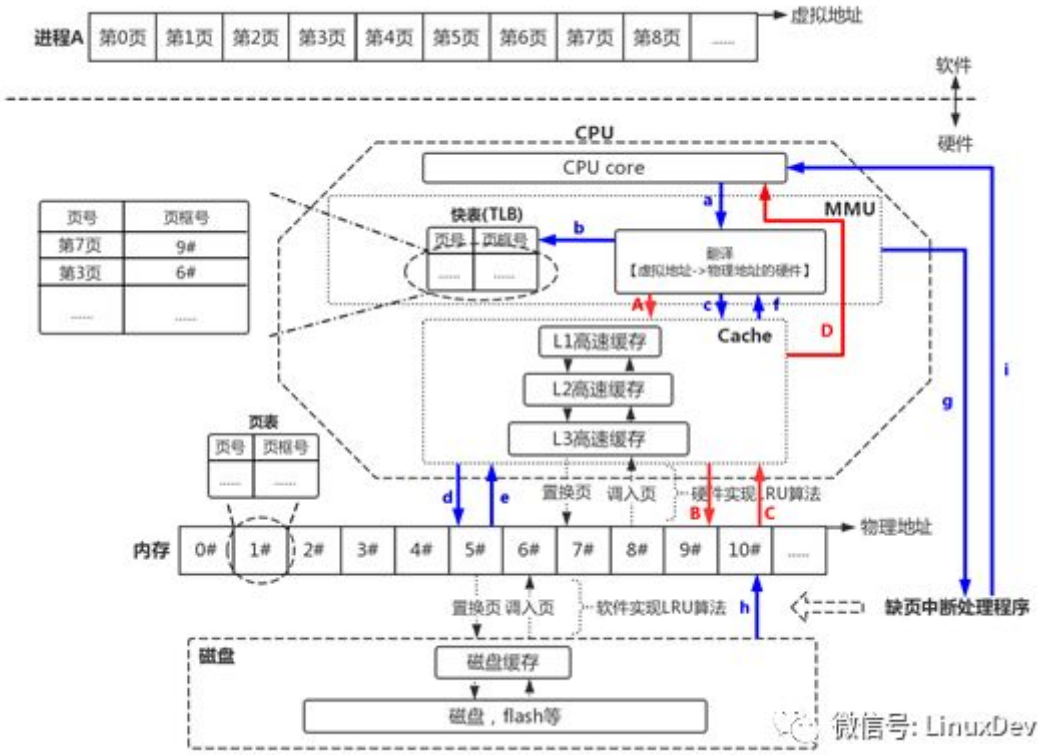


图 5-4总结图4

6. 用3句代码小结

我们以3句C代码的操作流程，对前面章节做一次小结。

```
char *p = malloc(4 * 1024 * 1024);
memset(p, 'g', 1024);
p[1] = '\0';
```

为了方便理解，假设此3行代码编译后执行的设备为32位LinuxPC，假设页大小为4K，且只有1级页表。页表项结构为：

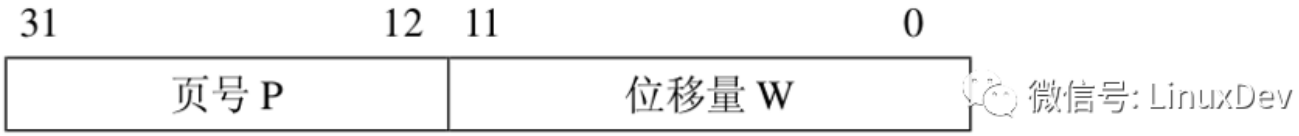


图 6-1页表项结构

需要澄清的是，本章为了方遍理解做了一些简化，实际执行过程肯定更复杂。

6.1. 第1句：char *p = malloc(4 * 1024 * 1024);

6.1.1. 页表情况

假设malloc返回p的值是 0x10240011，表示虚拟地址 [0x10240011, 0x10640011) 的4M空间被这次malloc申请成功。那么其对应的虚拟页号和页内偏移如何计算呢?

为了方便计算，我们把16进制的逻辑地址转为10进制的逻辑地址，则：

开始虚拟地址：0x10240011 => 270794769

结束虚拟地址：0x10640011 => 274989073

如何计算1级页表的页号和页内偏移呢？（多级页表换算更复杂，此处为了简化，只使用1级页表）

一级页表页号 = 地址 / 页大小【求整】

一级页表页内偏移 = 地址 % 页大小【求余】

因此计算的虚拟页号为：

开始虚拟地址：270794769 / (4 * 1024） = 66112

结束虚拟地址：274989073 / (4 * 1024） = 67136

虚拟页内偏移为：

开始虚拟地址：270794769 % (4 * 1024） = 17 = 0x11

结束虚拟地址：274989073 % (4 * 1024） = 17 = 0x11

我们以这样一张图描述执行代码后的页表情况：

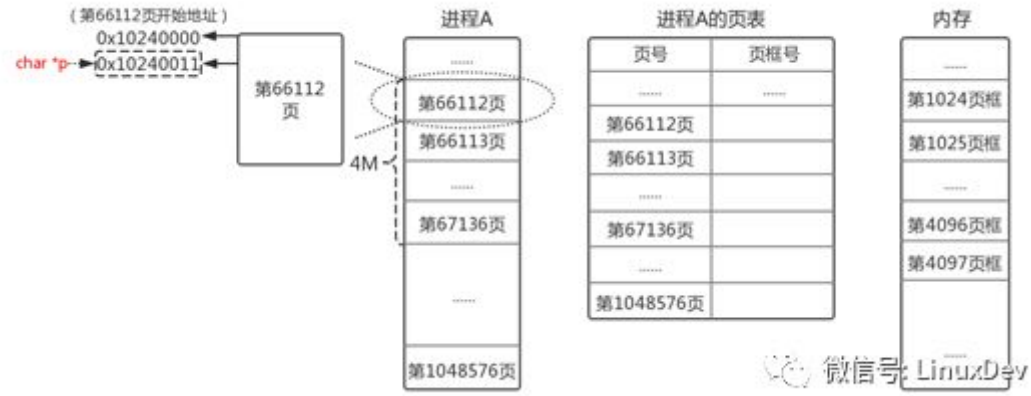


图 6-2第1句代码页表情况

上图要点解析：

a. malloc申请的空间地址是虚拟地址，虚拟地址！虚拟地址！

b. 刚申请时并没有立刻分配页框，因此页表上并没有对应页框号

6.2. 第2句：memset(p, 'g', 1024);

第2句代码把p指向的地址1K大小内存设置为'g'

6.2.1. 页表情况

图 6-3第2句代码页表情况

上图要点解析：

a. 由于只初始化了1K的数据量，在例子中只涉及1个页，且虚拟页号为：66112

b. 查询页表发现第66112页并没有对应物理页框，于是为其分配了第4097页框

c. CPU执行指令，把虚拟地址0x10240011开始的1K内存初始化为'g'，因此虚拟页映射的内存物理页框偏移0x11到0x411被写入'g'

6.2.2. CPU寻找页面过程

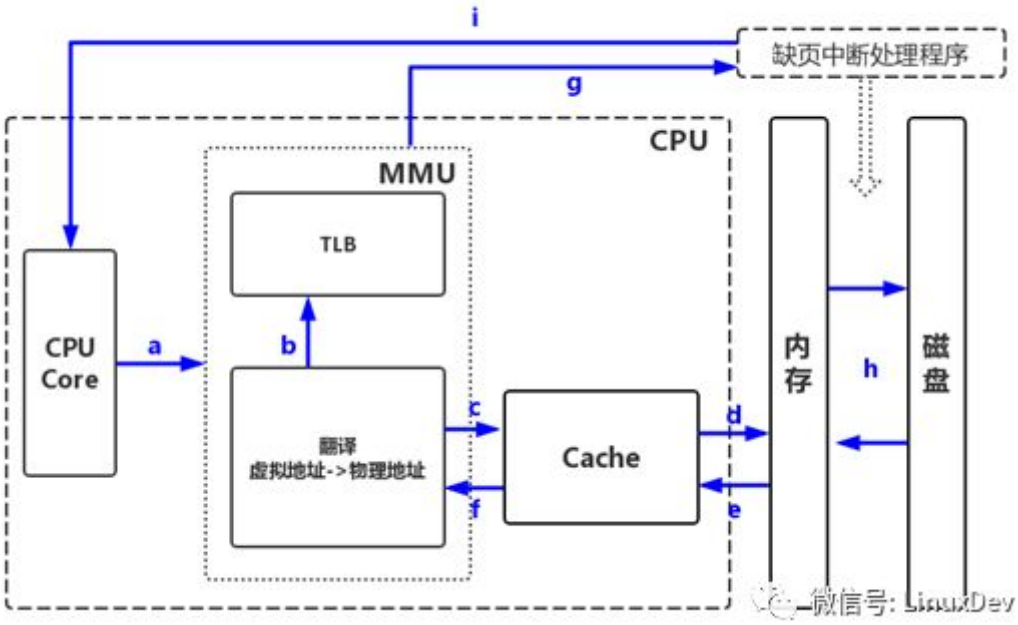


图 6-4获取物理地址

【a】CPU向MMU发送虚拟地址0x10240011，转换出虚拟页号为第66112页，偏移为0x11
【b】MMU查询快表TLB，快表不命中
【c】从高速缓存查询页表，高速缓存不命中
【d】从内存中查询页表，内存中有进程的完整页表
【e】把查询的页表项记录到高速缓存
【f】MMU从高速缓存中获取页表项，发现虚拟地址没分配物理页框，触发缺页中断
【g】进入缺页中断处理程序
【h】为第66112号虚拟页分配第4097页框，更新内存页表
【i】缺页中断处理的最后，修改CPU寄存器，使得重新启动导致缺页的指令

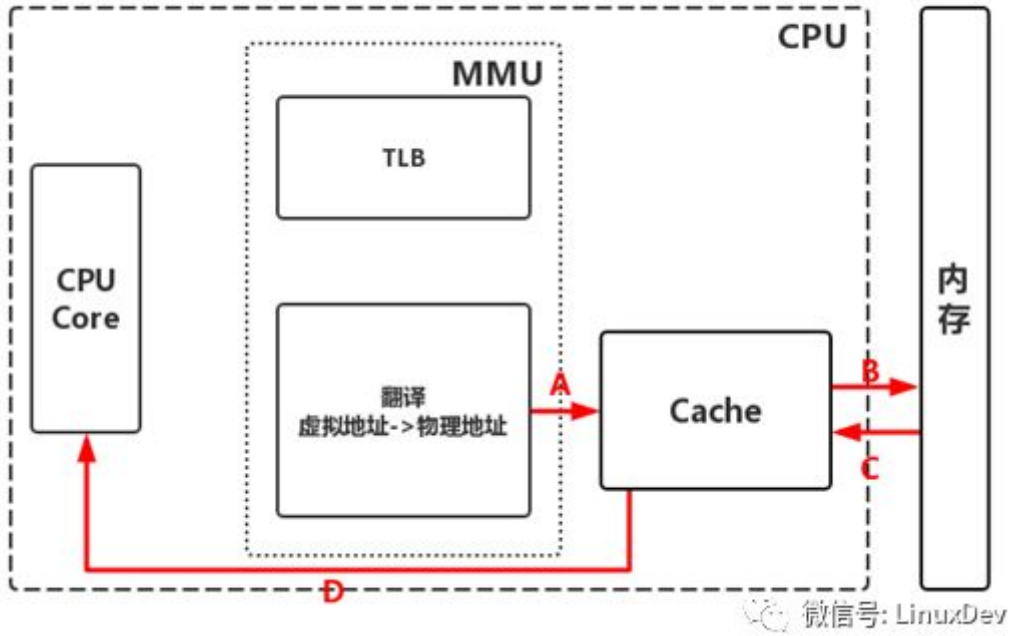


图 6-5根据物理地址获取数据

【A】MMU根据物理页框号4097查询高速缓存，发现没有对应页框缓存
【B】根据物理地址，获取物理内存中的数据
【C】硬件实现把第4097页框加载到高速缓存中
【D】CPU根据物理地址0x01001011，把1K大小的'g'写入高速缓存中，高速缓存再在合适时候同步到内存中

6.3. 第3句：p[1] = '\0';

把p[1]的字节写入'\0'

6.3.1. 页表情况

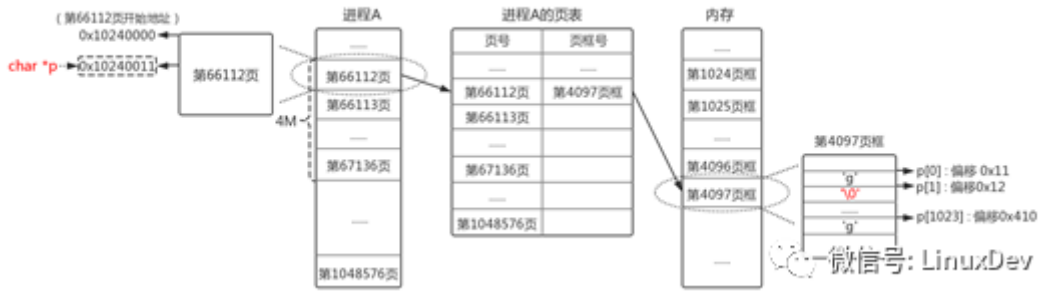


图 6-6 第3句代码页表情况

上图要点解析:

a. 由于上一行代码已经访问过对应物理地址，因此TLB和CPU Cache都会命中

6.3.2. CPU寻找页面过程

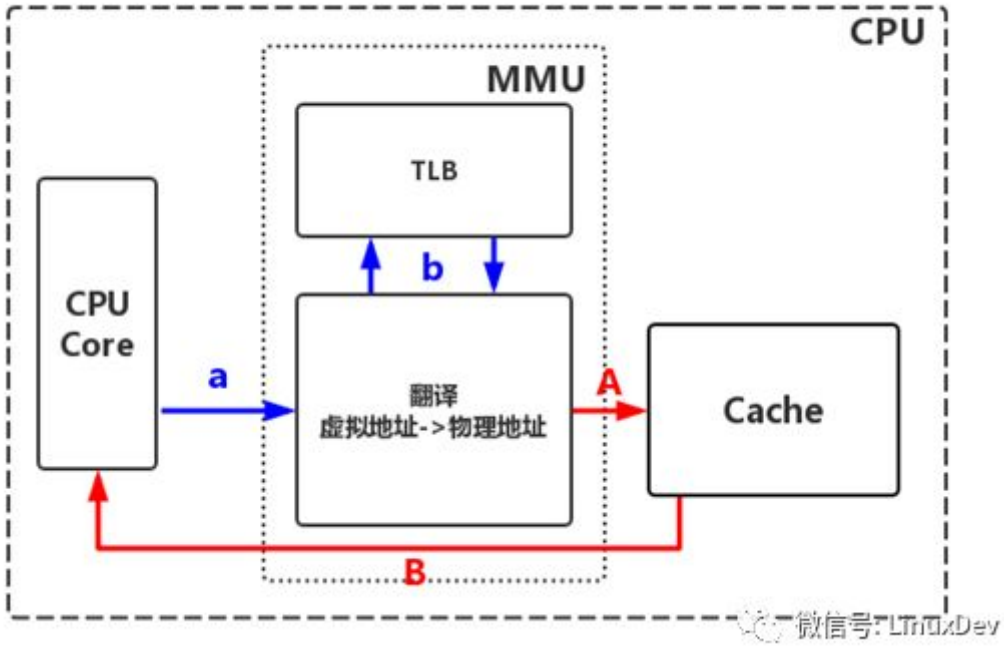


图 6-7 命中时获取数据过程

【a】CPU向MMU发送虚拟地址 0x10240012，虚拟页号为第66112页，偏移为0x12

【b】MMU查询快表TLB快表命中，从快表获取物理地址为0x01001012，页框号为4097，偏移为0x12。

【A】MMU获取物理地址后，根据物理地址查询高速缓存。高速缓存命中，CPU直接从高速缓存获取数据

【D】CPU把高速缓存内物理地址为0x01001012的1字节内容改为'\0'，高速缓存再在合适时候同步到内存中

7. 进程切换和花销

7.1. 进程切换时内核要做什么?

进程切换也叫进程上下文切换。什么叫进程上下文?

> “切换时，一个进程存储在处理器各寄存器中的中间数据叫做进程的上下文”

所以进程的切换实质上就是被中止运行进程与待运行进程上下文的切换，简单来说就是把正在CPU中执行的程序的所有信息保存在进程的堆栈中，腾出CPU以及紧缺的CPU寄存器给待执行进程。

进程切换(进场上下文切换)时，内核需要保存旧进程的什么？需要恢复新进程的什么?

a. 保存处理器PC寄存器的值到被中止进程的私有堆栈;

PC寄存器指向存放的是下一步要访问的内存地址

b. 保存处理器PSW寄存器的值到被中止进程的私有堆栈;

PSW寄存器反映处理器的状态及某些计算结果以及控制指令的执行

c. 保存处理器其他寄存器的值到被中止进程的私有堆栈;

d. 保存处理器SP寄存器的值到被中止进程的进程控制块;

SP寄存器指向进程私有堆栈栈顶

e. 刷掉部分TLB和Cache;

参考网友分析文

（http://www.wowotech.net/process_management/context-switch-tlb.html）

其提到TLB和Cache在进程切换时会flush部分TLB和Cache

f. 自待运行进程的进程控制块取SP值并存入处理器的寄存器SP；

g. 自待运行进程的私有堆栈恢复处理器各寄存器的值；

h. 自待运行进程的私有堆栈中弹出PSW值并送入处理器的PSW；

i. 自待运行进程的私有堆栈中弹出PC值并送入处理器的PC

总结起来，就两个步骤：

保存：把各个寄存器保存在进程私有堆栈中，把私有堆栈指针保存在进程控制块(struct task)中

恢复：从进程控制块(structtask)中获取私有堆栈指针，再从私有堆栈中弹出之前保存的各种寄存器

7.2. 为什么说大量进程切换花销大?

进程切换花销主要包含直接花销和间接花销的2个方面。

直接花销：

CPU寄存器需要保存和加载，系统调度器的代码需要执行。

间接花销：

TLB和Cache由于进程切换导致不命中，前期触发大量的缺页中断。

因此，当存在大量的进程切换时，CPU需要做大量的进程上下文切换工作和缺页中断，这些花销是非常可观的。

8. 本文参考资料

8.1. 参考书

《计算机操作系统 第三版》

8.2. 参考链接

8.2.1. 存储结构

https://blog.csdn.net/u013471946/article/details/41456055

8.2.2. 页的调入、置换、缺页、DMA、Cache

http://blog.sina.com.cn/s/blog_a018ffa90101smj8.html

https://baike.baidu.com/item/CPU%E7%BC%93%E5%AD%98/3728308?fr=aladdin

http://blog.chinaunix.net/uid-13246637-id-5185352.html

https://blog.csdn.net/zssmcu/article/details/6836869

https://blog.csdn.net/chinesedragon2010/article/details/5922324

8.2.3. TLB

https://baike.baidu.com/item/TLB/2339981?fr=aladdin

http://www.wowotech.net/process_management/context-switch-tlb.html

8.2.4. 进程切换

https://blog.csdn.net/gettogetto/article/details/74629805

https://blog.csdn.net/gatieme/article/details/51872594

（完）