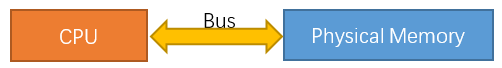
# 第五期 内存管理浅析

# 1 内存管理背景

关于内存的管理使用，

其实最开始内存的使用是这样的：

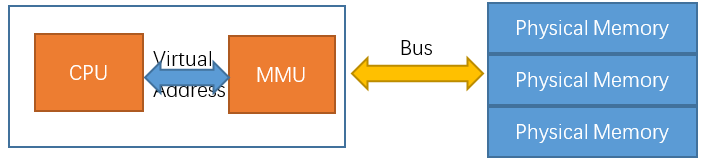


**图1.1** 简单内存访问方式

CPU从进程拿到物理地址，通过总线(Bus)访问对应的物理内存(Physical Memory)地址。目前单片机访问逻辑仍然是这样的，对于单片机程序，它能访问所有的内存；因为是独占物理内存，所以只能run一个程序；所以可以跑一些功能逻辑相对简单的单一程序；

然而，面对当前复杂的应用场景，这种直接访问内存的方式就显得力不从心了；这里的问题原因在于，需要对多个程序进行调度(进程调度管理)，同时对程序访问内存的时机进行管理（内存管理）；而操作系统的重要职责之一就是内存管理；

于是，操作系统下的内存管理是这样的：



**图1.2** 现代内存访问方式

CPU从进程里读到的是虚拟地址(Virtual Address)，虚拟地址经过MMU(Memory Management Unit )映射到实际的物理地址，再通过总线(Bus)访问物理内存；这样的好处在于，对于每个进程而言，都是独占所有的虚拟“内存空间”，比如32bit系统，每个进程可以看到的虚拟地址空间size都是4GB。其中，MMU是一个负责CPU访问物理内存的硬件器件，它的主要作用是：虚拟地址à物理地址转换、内存访问保护、高速cache控制等。

# 2 内存管理

目前主流的内存管理方式，如表2.1：

**表2.1** 主流内存管理方式

|  |  |  |  |
| --- | --- | --- | --- |
| **内存管理** | **原理** | **优点** | **缺点** |
| **分页式管理** | 将内存空间按页的大小划分成片或者页面，然后把页式虚拟地址和内存地址建立一一对应的页表，并用相应的硬件地址变换机构来解决离散地址变换问题 | 无外部碎片 | 有内部碎片 |
| **分段式管理** | 程序是由若干个逻辑分段(Segmentation)组成的，如可由代码段、数据段、栈段、堆段等组成。 | 无内部碎片 | 有外部碎片 |
| **段页式管理** | 系统为每个作业或者进程建立一张段表以管理内存分配与释放，缺段处理等。  一个段又被划分成了若干页。每个段又必须建立一张页表以把段中的虚页变换成内存中的实际页面。页表中也要有相应的实现缺页中断处理和页面保护等功能的表项。  段页式管理是段式管理与页式管理方案的结合。 | 解决内部/外部碎片问题 | 执行速度相对慢一些，一次地址寻址最坏情况需要三次访问内存； |

## 2.1 内存碎片

关于内存碎片，我们来看看这样一个例子。

假设有 1G 的物理内存，用户执行了多个程序，其中：

• 游戏占用了 512MB 内存

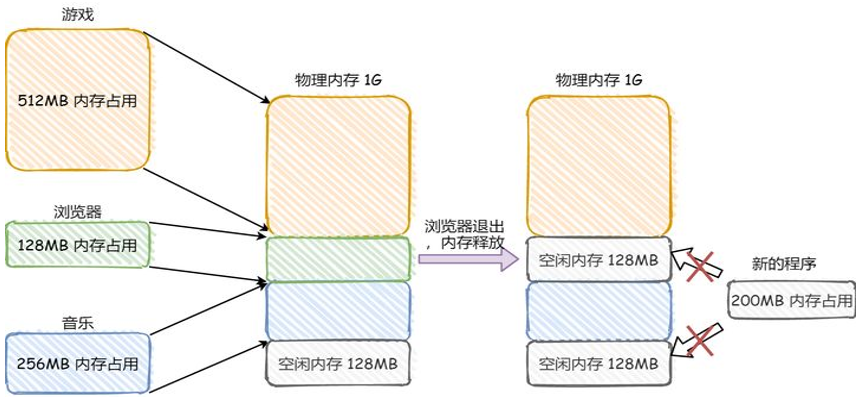
• 浏览器占用了 128MB 内存

• 音乐占用了 256 MB 内存。

这个时候，如果我们关闭了浏览器，则空闲内存还有 1024 - 512 - 256 = 256MB。

如果这个 256MB 不是连续的，被分成了两段 128 MB 内存，这就会导致没有空间再打开一个 200MB 的程序。

来看看这样一个例子。假设有 1G 的物理内存，用户执行了多个程序，其中：



**图2.1** 内存占用示意图

这里的内存碎片的问题：

• 外部内存碎片，也就是产生了多个不连续的小物理内存，导致新的程序无法被装载；

• 内部内存碎片，程序所有的内存都被装载到了物理内存，但是这个程序有部分的内存可能并不是很常使用，这也会导致内存的浪费；

针对上面两种内存碎片的问题，解决的方式会有所不同。

解决外部内存碎片的问题就是内存交换。可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回来到内存里。不过再读回的时候，我们不能装载回原来的位置，而是紧紧跟着那已经被占用了的 512MB 内存后面。这样就能空缺出连续的 256MB 空间，于是新的 200MB 程序就可以装载进来。

这个内存交换空间，在 Linux 系统里，也就是我们常看到的 Swap 空间，这块空间是从硬盘划分出来的，用于内存与硬盘的空间交换。

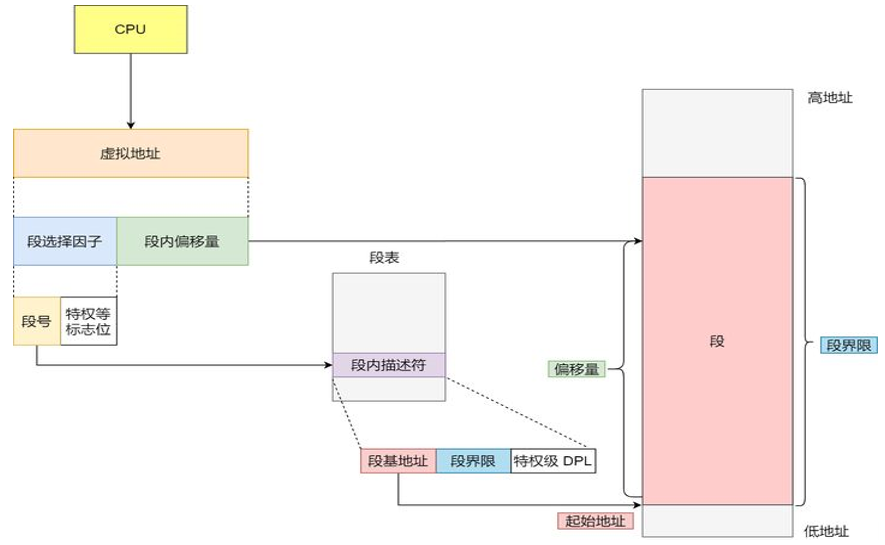
对于多进程的系统来说，用分段的方式，内存碎片是很容易产生的，产生了内存碎片，那不得不重新 Swap 内存区域，这个过程会产生性能瓶颈。因为硬盘的访问速度要比内存慢太多了，每一次内存交换，我们都需要把一大段连续的内存数据写到硬盘上。

所以，如果内存交换的时候，交换的是一个占内存空间很大的程序，这样整个机器都会显得卡顿。

## 2.2 分段式管理

我们写程序，经常会遇到内存越界等引起的段错误，进而导致进程crash掉。要弄清段错误原因，首先需要了解分段式内存管理机制。

分段机制下的虚拟地址由两部分组成，段选择子和段内偏移量。

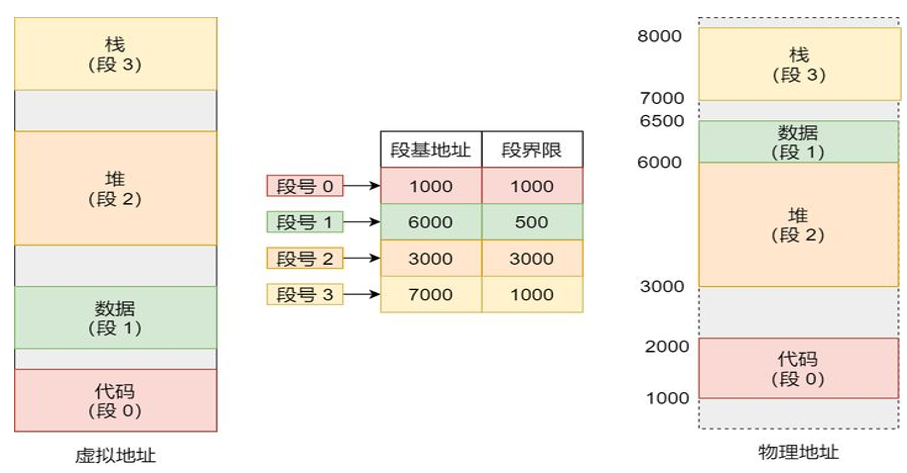


**图2.2** 段式内存管理寻址

· **段选择子**就保存在段寄存器里面。段选择子里面最重要的是**段号**，用作段表的索引。**段表**里面保存的是这个**段的基地址、段的界限和特权等级**等。

· 虚拟地址中的**段内偏移量**应该位于 0 和段界限之间，如果段内偏移量是合法的，就将段基地址加上段内偏移量得到物理内存地址。

分段式管理机制就是通过上述段表与物理地址之间的映射完成程序的寻址操作。实际上，分段机制会把程序的虚拟地址分成 4 个段，每个段在段表中有一个项，在这一项找到段的基地址，再加上偏移量，于是就能找到物理内存中的地址，如下图2.3：



**图2.3** 段式内存管理地址映射

如果要访问段 3 中偏移量 500 的虚拟地址，我们可以计算出物理地址为，段 3 基地址 7000 + 偏移量 500 = 7500。

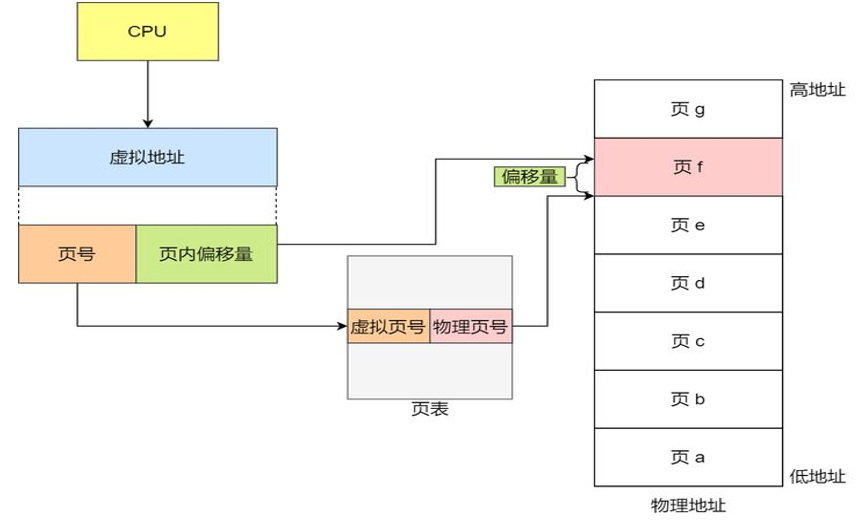
假设我们给出的寻址偏移地址是10000，此时，超出此例中的段最大可寻址范围，会触发内存访问段错误bug；

再假设我们给出试图写的一个偏移地址是1200，刚好落在代码段(默认情况代码段是只读权限)内，也会触发内存访问保护段错误bug；

## 2.3 分页式管理

缺页中断也称缺页异常，是指当软件试图访问已映射在虚拟地址空间中，但是并未被加载在物理内存中的一个分页时，由内存管理单元(MMU)所发出的中断。

在分页机制下，虚拟地址分为两部分，**页号**和**页内偏移**。页号作为页表的索引，**页表**包含物理页每页所在**物理内存的基地址**，这个基地址与页内偏移的组合就形成了物理内存地址，见下图2.4。



**图2.4** 页式内存管理寻址

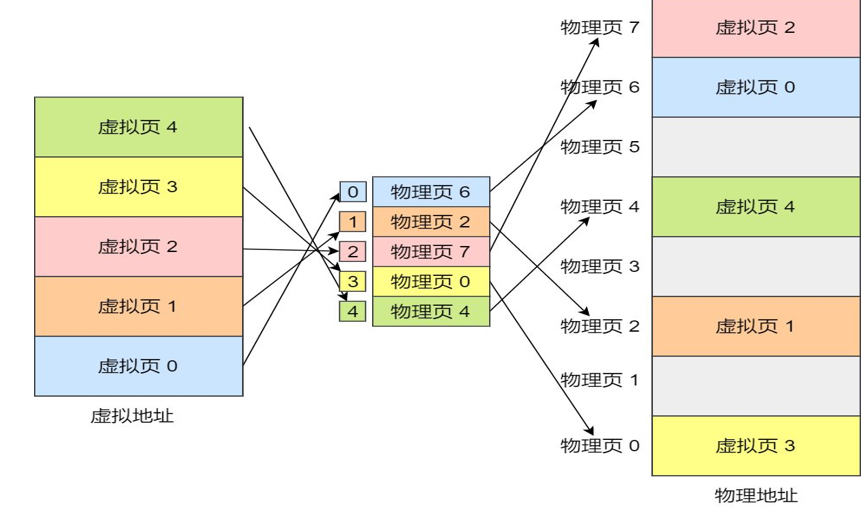
总结一下，对于一个内存地址转换，其实就是这样三个步骤：

· 把虚拟内存地址，切分成页号和偏移量；

· 根据页号，从页表里面，查询对应的物理页号；

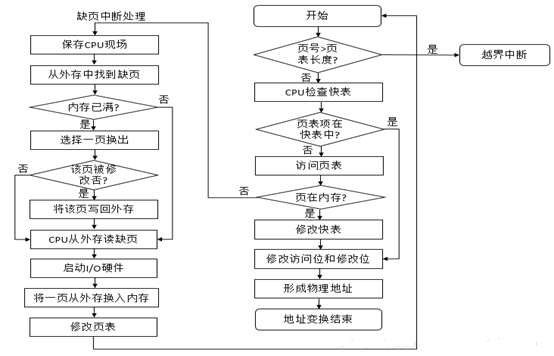
· 直接拿物理页号，加上前面的偏移量，就得到了物理内存地址。

虚拟内存中的页通过页表映射为了物理内存中的页，如下图2.5：



**图2.5** 页式内存管理地址映射

如果请求的页不在内存中，则会触发缺页中断：



**图2.6** 缺页中断流程图

# 3 内存应用优化

## 3.1 静态/动态内存分配：

**（1）静态存储分配**

通常定义变量（或对象），编译器在编译时都可以根据该变量（或对象）的类型知道所需内存空间的大小，从而系统在适当的时候为他们分配确定的存储空间。这种内存分配称为静态存储分配；比如初始化的静态变量/全局变量等。

**（2）动态存储分配**

有些操作对象只在程序运行时才能确定，这样编译时就无法为他们预定存储空间，只能在程序运行时，系统根据运行时的要求进行内存分配，这种方法称为动态存储分配。所有动态存储分配都在堆区中进行。

* **malloc函数**

void \*malloc(unsigned int size)；

//在内存的动态分配区域中分配一个长度为size的连续空间//堆区

（1）返回值：通用类型（void\* 类型可以通过类型转换强制转换为任何其它类型的指针）

分配成功，返回一个指向分配起始地址的指针；

分配失败，返回NULL

（2）注意：

a. 使用malloc进行动态内存分配后应判断指针是否为空，即是否分配成功；

b. 申请的内存不会进行初始化，为随机值；

c. 使用完毕以后必须手动释放内存空间，否则会造成内存泄露

d. 字节对齐 : 会对齐到机器最受限的类型（具体的实现因机器而异）。

* **calloc函数**

void \*calloc(unsigned int num, unsigned int size);

//按照所给的数据项个数和数据类型所占字节数，分配一个 num \* size 连续的空间

（1）返回值：

与malloc函数相同

（2）注意：

①申请内存后会自动初始化内存空间为0；

②使用calloc进行动态内存分配后应判断指针是否为空，即是否分配成功；

③使用完毕以后必须手动释放内存空间，否则会造成内存泄露

* **realloc函数**

void \*realloc(void \*mem\_address, unsigned int newsize);

//更改已经配置的内存空间，即更改由malloc()函数分配的内存空间的大小

//在指针mem\_address指向的新地址上重新动态分配newsize字节大小的内存空间

（1）返回值：

如果将分配的内存减少，realloc仅仅是改变索引的信息；

如果是将分配的内存扩大，则有以下情况：

1）直接拓展：如果当前内存段后面有需要的内存空间，则直接扩展这段内存空间，realloc()将返回原指针。

2）重新分配：如果当前内存段后面的空闲字节不够，那么就使用堆中的第一个能够满足这一要求的内存块，将目前的数据复制到新的位置，并将原来的数据块释放掉，返回新的内存块位置。

3）如果申请失败，将返回NULL，此时，原来的指针仍然有效。

注意:

①对于扩大内存的内存分配成功的两种情况，无论返回原指针还是新指针指向地址，**原来指针均已自动释放过**，不需要使用free再次释放掉原来的指针；②申请的内存空间不会进行初始化；③使用完毕以后必须手动释放内存空间，否则会造成内存泄露

* **free函数**

void free(void\* p)；

//free 函数只有一个参数,就是所要释放的内存块的首地址。

（2）注意：

a. free释放掉指定参数指针指向的内存，并返回void。其实，free函数只是将参数指针指向的内存归还给操作系统，并不会把参数指针置NULL，为了以后访问到被操作系统重新分配后的错误数据，所以在调用free之后，通常需要手动将指针置NULL。从另一个角度来看，内存这种底层资源都是由操作系统来管理的，而不是编译器，编译器只是向操作系统提出申请。所以free函数是没有能力去真正的free内存的。只是告诉操作系统它归还了内存，然后操作系统就可以修改内存分配表，以供下次分配。

b. 在free前后，指针变量 p 本身保存的地址、被释放的内存里面保存的值均不改变，但是它对这个地址处的那块内存却已经没有所有权。

* **new 与 delete**

new 与delete是C++预定的操作符，它们一般需要配套使用。

（1）new用于从堆内存申请一块空间，一般动态用于动态申请内存空间，即根据程序需要，申请一定长度的空间。

自动计算需要分配的空间，在分配类类型的内存空间时，同时调用类的构造函数，对内存空间进行初始化，即完成类的初始化工作。动态分配内置类型是否自动初始化取决于变量定义的位置，在函数体外定义的变量都初始化为0，在函数体内定义的内置类型变量都不进行初始化。

new 有以下的三种格式申请内存空间

new 数据类型

new 数据类型（初始值）

new 数据类型[常量表达式]

*int \* p1=new int;*

*int \*p2=new int(2); //\*p2初始化值是2*

*int \*p3=new int[1000] //申请1000个单位内存空间*

（2）delete则是将new申请的空间释放。

*delete p1;*

*delete p2;*

*delete[] p3;*//注意此处不能用delete p3,因为在申请用了[]，则在释放时要用delete[]

## 3.2 零拷贝

零拷贝主要的任务就是**避免**CPU将数据从一块存储拷贝到另外一块存储，主要就是利用各种零拷贝技术，避免让CPU做大量的数据拷贝任务，减少不必要的拷贝，或者让别的组件来做这一类简单的数据传输任务，让CPU解脱出来专注于别的任务。这样就可以让系统资源的利用更加有效。

我们继续回到引文中的例子，我们如何减少数据拷贝的次数呢？一个很明显的着力点就是减少数据在内核空间和用户空间来回拷贝，让数据传输不需要经过user space。

* **mmap**

我们减少拷贝次数的一种方法是调用mmap()来代替read调用：

buf = mmap(diskfd, len)*;*

write(sockfd, buf, len)*;*

应用程序调用**mmap**()，磁盘上的数据会通过DMA被拷贝的内核缓冲区，接着操作系统会把这段内核缓冲区与应用程序共享，这样就不需要把内核缓冲区的内容往用户空间拷贝。应用程序再调用write(),操作系统直接将内核缓冲区的内容拷贝到socket缓冲区中，这一切都发生在内核态，最后，socket缓冲区再把数据发到网卡去。同样的，看图很简单：



**图3.1 零拷贝-mmap**

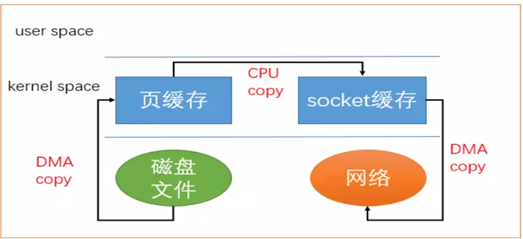
使用**mmap**替代read很明显减少了一次拷贝，当拷贝数据量很大时，无疑提升了效率。但是使用mmap是有代价的。当你使用**mmap**时，你可能会遇到一些隐藏的陷阱。例如，当你的程序map了一个文件，但是当这个文件被另一个进程截断(truncate)时, write系统调用会因为访问非法地址而被SIGBUS信号终止。SIGBUS信号默认会杀死你的进程并产生一个coredump。

* **sendfile**

#include<sys/sendfile.h>

ssize\_t sendfile(int out\_fd, int in\_fd, off\_t \*offset, size\_t count);

系统调用sendfile()在代表输入文件的描述符in\_fd和代表输出文件的描述符out\_fd之间传送文件内容（字节）。描述符out\_fd必须指向一个套接字，而in\_fd指向的文件必须是可以mmap的。这些局限限制了**sendfile**的使用，使**sendfile**只能将数据从文件传递到套接字上，反之则不行。使用**sendfile**不仅减少了数据拷贝的次数，还减少了上下文切换，数据传送始终只发生在kernel space。



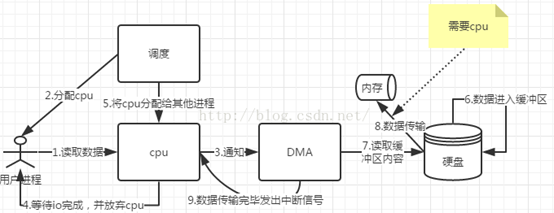
**图3.2 零拷贝-sendfile**

在我们调用sendfile时，如果有其它进程截断了文件会发生什么呢？假设我们没有设置任何信号处理程序，**sendfile**调用仅仅返回它在被中断之前已经传输的字节数，errno会被置为success。如果我们在调用sendfile之前给文件加了锁，**sendfile**的行为仍然和之前相同，我们还会收到RT\_SIGNAL\_LEASE的信号。

**目前为止，**我们已经减少了数据拷贝的次数了，但是仍然存在一次拷贝，就是页缓存到socket缓存的拷贝。那么能不能把这个拷贝也省略呢？

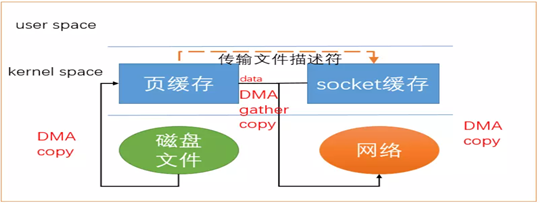
我们需要借助硬件DMA。之前我们是把页缓存的数据拷贝到socket缓存中，实际上，我们仅仅需要把缓冲区描述符传到socket缓冲区，再把数据长度传过去，这样DMA控制器直接将页缓存中的数据打包发送到网络中即可。

先简单看一下通用的DMA工作方式



**图3.3 DMA工作原理示意图**

**sendfile**系统调用利用DMA引擎将文件内容拷贝到内核缓冲区去，然后将带有文件位置和长度信息的缓冲区描述符添加socket缓冲区去，这一步不会将内核中的数据拷贝到socket缓冲区中，DMA引擎会将内核缓冲区的数据拷贝到协议引擎中去，避免了最后一次拷贝。



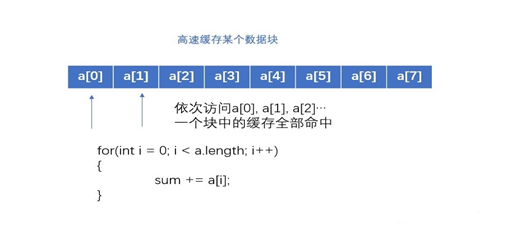
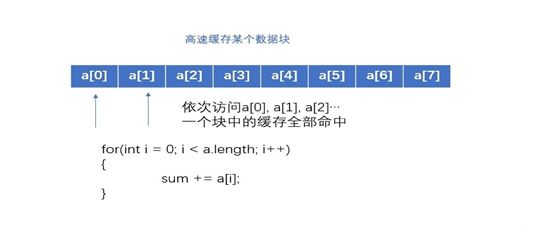
**图3.4 零拷贝-sendfile DMA方式**

## 3.3 局部性原理

一个优秀的程序通常具有良好的局部性，它们通常会重复使用已用过的数据，或者使用已用过数据的邻近数据，也就是说，程序常常会使用集中在一起的局部数据。

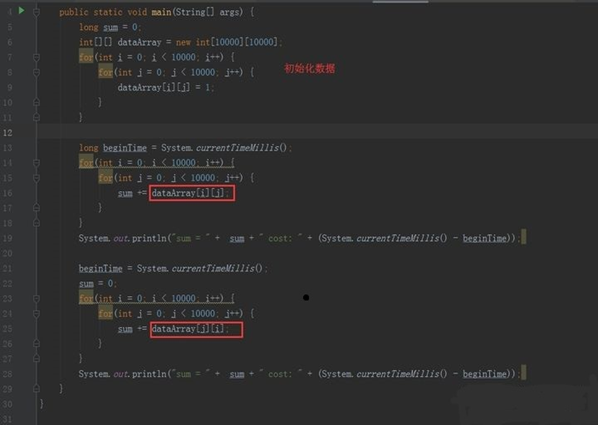
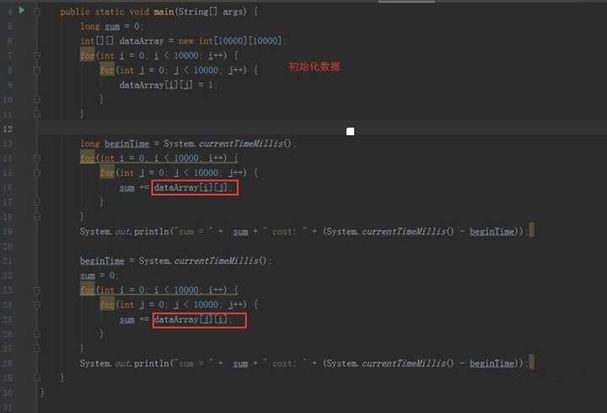
局部性分为：时间局部性和空间局部性。如果一个内存位置被重复的引用，那就是有了时间局部性，如果一个内存位置被引用了，很快这个位置的附近位置也被引用了，这就有了空间局部性。

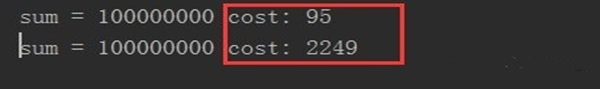
由于缓存中的数据是一个个数据块，每个数据块包含几十到几千字节不等，如果某个程序要访问数组a，第一次缓存没命中，cpu会从主存中取出包含数组a的一个数据块，复制到缓存中来，下次访问a[1],a[2],a[3]的数据时每次都缓存命中，极大的提高了效率，实现了空间的局部性。



**图3.5 高速缓存局部性访问**

【实例验证】





**图3.6 局部性原理应用实例**

在此例中，循环累加int[10000][10000] 的二维数组，按i，j 的顺序访问利用了缓存的局部性访问原理，效率提高了200多倍。

## 3.4 内存屏障

内存屏障，是保证屏障前的读写指令必须在屏障后的读写指令之前顺序执行。

解决如下三个问题：

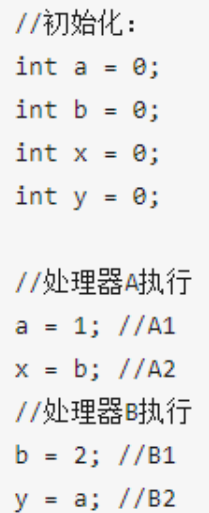
* 1、**编译器优化的重排序**。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序；

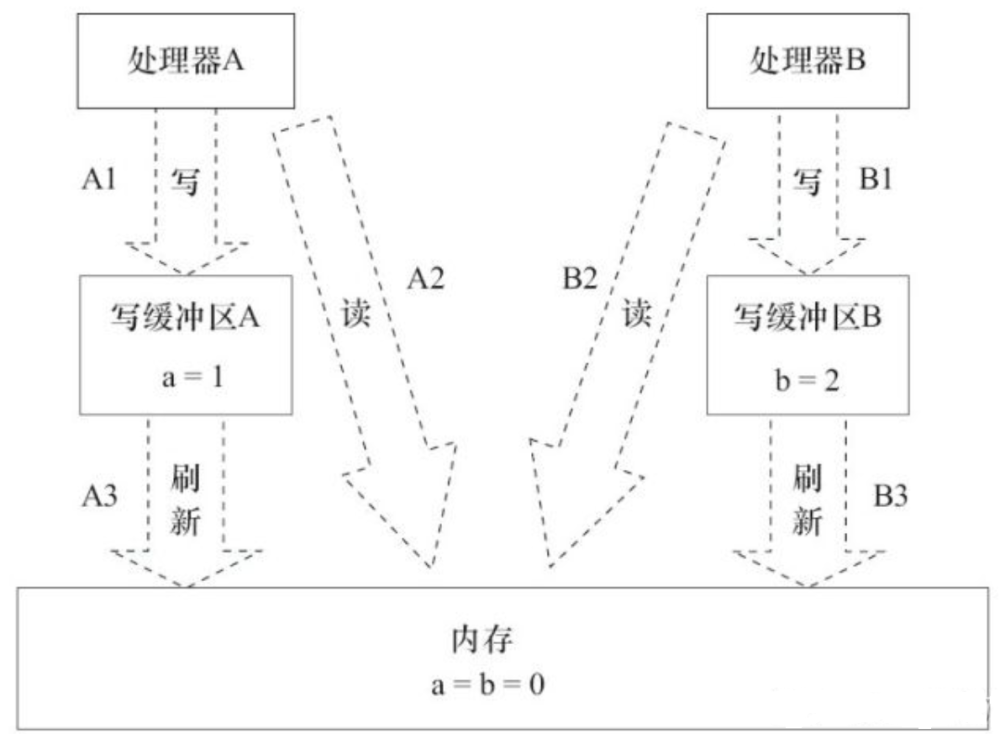
* 2、**指令级并行的重排序**。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序；

* 3、**内存的重排序**。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行的。

先来看一则例子：

多核处理器实际执行，如下：





**图3.7 处理器可能的执行过程**

写缓存区没有及时刷新，使得处理器执行的读写操作与内存上**顺序不一致**。

处理器A读b=0，处理器B读a=0。A1写a=1先写到处理器A的写缓存区中，此时内存中a=0。如果这时处理器B从内存中读a，读到的将是0。也可能会出现x,y都是0。

另一个可能内核的例子：

x = 0;

isOK = 0;

// thread 1

if (isOK) {

y=x;// 期望y=2；

}

// thread 2

x = 2;

**mb();**

isOK = 1;

一般来讲，CPU重排序和cache访问排序发生在内核态(mb,wmb,rmb)，用户态感知不到，对于用户态可以做的是处理编译器的优化：加锁和volatile;

可参见：<https://www.cnblogs.com/xiayong123/archive/2012/03/31/3717124.html>

***【参考资料】***

[*https://yuque.antfin.com/weijjia.wj/tidex/memory*](https://yuque.antfin.com/weijjia.wj/tidex/memory)

[*https://yuque.antfin.com/uwk7mr/wqwyc9/tygisq*](https://yuque.antfin.com/uwk7mr/wqwyc9/tygisq)

[*https://yuque.antfin.com/uwk7mr/molt9b/tbaeyp*](https://yuque.antfin.com/uwk7mr/molt9b/tbaeyp)

[*https://zhuanlan.zhihu.com/p/149581303*](https://zhuanlan.zhihu.com/p/149581303)

[*https://blog.csdn.net/hit\_shaoqi/article/details/78516508*](https://blog.csdn.net/hit_shaoqi/article/details/78516508)

[*https://blog.csdn.net/qq\_39191122/article/details/79734128*](https://blog.csdn.net/qq_39191122/article/details/79734128)

*===============================The End==================================*