# vmalloc与mmap

mmap()系统调用是在用户进程与内核之间共享内存区域的常用方法。我们最近有个程序，需要应用进程能够读取内核驱动获取的数据，经过简单的调研，决定采用mmap方式。实现起来不难，在驱动中注册一个字符设备，实现该设备的mmap()方法即可。但这其中有一点小曲折。

在实现设备的mmap()方法时，需要将物理内存映射到应用程序通过mmap()系统调用传下来的vma中。vma代表的是进程的一段虚拟地址空间。在第一版里，考虑的不全面，利用alloc\_pages()将整个内存段申请为一段连续的物理地址空间。然后通过remap\_pfn\_range()函数将这段连续的物理内存映射到vma中。经过长时间的测试，没有发现问题。直到今天，在部署一个老集群时，遇到了问题。这个集群中有很多老机器，内存只有十多个G，而且长时间运行后产生了大量的内存碎片。从而导致，我们无法获得足够的连续物理内存。没办法，只好重新调整驱动中分配内存的方式，改用vmalloc获取地址空间。

在kernel里，通常有3种申请内存的方式：vmalloc, kmalloc, alloc\_pages。kmalloc与alloc\_pages类似，均是申请连续的地址空间。而vmalloc则可以申请一段不连续的物理地址空间，并将其映射到连续的线性地址上。每次vmalloc之后，内核会创建一个**vm\_struct**，用以映射分配到的不连续的内存区域。vm\_struct类似**vma**，但是又不是一回事。vma是将物理内存映射到进程的虚拟地址空间。而vm\_struct是将物理内存映射到内核的线性地址空间。

既然vmalloc拿到的不是连续的物理内存，那么将这些内存映射到vma时，就不能直接利用remap\_pfn\_range()了。

此时可以采用两种方法，一种是实现vm\_operations\_struct的fault()方法，用以在缺页时再映射需要的页。此方法操作起来较为麻烦。

另一种方法是直接使用remap\_vmalloc\_range()函数。该函数的原型为：

int remap\_vmalloc\_range(struct vm\_area\_struct \*vma, void \*addr, unsigned long pgoff)

其中参数vma是mmap使用调用传下来的，addr即为vmalloc()所分配内存的起始地址。而pgoff则为mmap()系统调用里的偏移参数，可以通过vma->vm\_pgoff获得。该函数成功执行后，返回值为0。如果返回值为负数，则说明出错了。通常是由于所传的参数不正确。

需要注意的是，需要映射到用户空间的内存段，不能直接利用vmalloc()分配，而应该使用**vmalloc\_user()**函数。该函数除了分配内存之外，还会将相应的vm\_struct结构标记为VM\_USERMAP。否则，remap\_vmalloc\_range将返回错误。

在这个项目中碰到的教训是，永远不要假设系统中一定会有超过一个页的连续物理内存。

不过较新的内核具有compact机制，可以整理内存碎片。但是，目前至少有一大部分机器不支持，或未开启此机制。

# linux内存映射mmap原理分析

一直都对内存映射文件这个概念很模糊，不知道它和虚拟内存有什么区别，而且映射这个词也很让人迷茫，今天终于搞清楚了。。。下面，我先解释一下我对映射这个词的理解，再区分一下几个容易混淆的概念，之后，什么是内存映射就很明朗了。

## 原理

首先，“映射”这个词，就和数学课上说的“一一映射”是一个意思，就是建立一种一一对应关系，在这里主要是只 硬盘上文件 的位置与进程 逻辑地址空间 中一块大小相同的区域之间的一一对应，如图1中过程1所示。这种对应关系纯属是逻辑上的概念，物理上是不存在的，原因是进程的逻辑地址空间本身就是不存在的。在内存映射的过程中，并没有实际的数据拷贝，文件没有被载入内存，只是逻辑上被放入了内存，具体到代码，就是建立并初始化了相关的数据结构（struct address\_space），这个过程有系统调用mmap()实现，所以建立内存映射的效率很高。

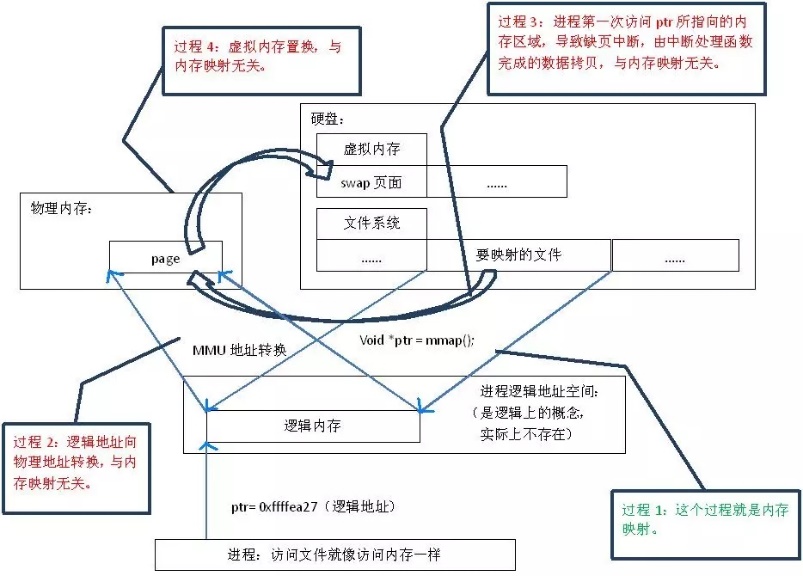


图1.内存映射原理

既然建立内存映射没有进行实际的数据拷贝，那么进程又怎么能最终直接通过内存操作访问到硬盘上的文件呢？那就要看内存映射之后的几个相关的过程了。

mmap()会返回一个指针ptr，它指向进程逻辑地址空间中的一个地址，这样以后，进程无需再调用read或write对文件进行读写，而只需要通过ptr就能够操作文件。但是ptr所指向的是一个逻辑地址，要操作其中的数据，必须通过MMU将逻辑地址转换成物理地址，如图1中过程2所示。这个过程与内存映射无关。

前面讲过，建立内存映射并没有实际拷贝数据，这时，MMU在地址映射表中是无法找到与ptr相对应的物理地址的，也就是MMU失败，将产生一个缺页中断，缺页中断的中断响应函数会在swap中寻找相对应的页面，如果找不到（也就是该文件从来没有被读入内存的情况），则会通过mmap()建立的映射关系，从硬盘上将文件读取到物理内存中，如图1中过程3所示。这个过程与内存映射无关。

如果在拷贝数据时，发现物理内存不够用，则会通过虚拟内存机制（swap）将暂时不用的物理页面交换到硬盘上，如图1中过程4所示。这个过程也与内存映射无关。

## 效率

从代码层面上看，从硬盘上将文件读入内存，都要经过文件系统进行数据拷贝，并且数据拷贝操作是由文件系统和硬件驱动实现的，理论上来说，拷贝数据的效率是一样的。但是通过内存映射的方法访问硬盘上的文件，效率要比read和write系统调用高，这是为什么呢？原因是read()是系统调用，其中进行了数据拷贝，它首先将文件内容从硬盘拷贝到内核空间的一个缓冲区，如图2中过程1，然后再将这些数据拷贝到用户空间，如图2中过程2，在这个过程中，实际上完成了两次数据拷贝；而mmap()也是系统调用，如前所述，mmap()中没有进行数据拷贝，真正的数据拷贝是在缺页中断处理时进行的，由于mmap()将文件直接映射到用户空间，所以中断处理函数根据这个映射关系，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝。因此，内存映射的效率要比read/write效率高。

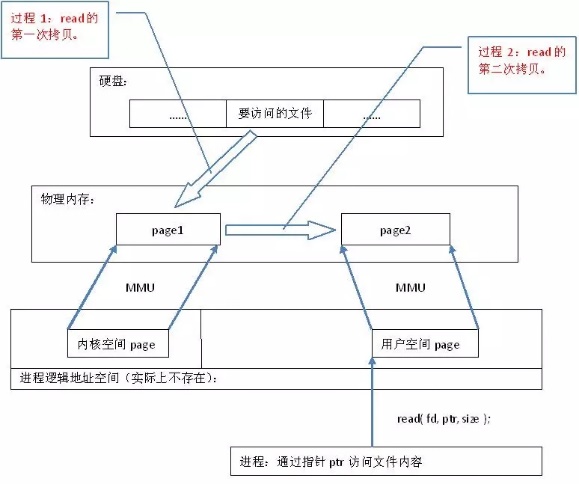


图2.read系统调用原理

下面这个程序，通过read和mmap两种方法分别对硬盘上一个名为“mmap\_test”的文件进行操作，文件中存有10000个整数，程序两次使用不同的方法将它们读出，加1，再写回硬盘。通过对比可以看出，read消耗的时间将近是mmap的两到三倍。

|  |
| --- |
| #include<unistd.h>  #include<stdio.h>  #include<stdlib.h>  #include<string.h>  #include<sys/types.h>  #include<sys/stat.h>  #include<sys/time.h>  #include<fcntl.h>  #include<sys/mman.h>    #define MAX 10000    int main()  {  int i=0;  int count=0, fd=0;  struct timeval tv1, tv2;  int \*array = (int \*)malloc( sizeof(int)\*MAX );    /\*read\*/  gettimeofday( &tv1, NULL );  fd = open( "mmap\_test", O\_RDWR );  if( sizeof(int)\*MAX != read( fd, (void \*)array, sizeof(int)\*MAX ) )  {  printf( "Reading data failed.../n" );  return -1;  }    for( i=0; i<MAX; ++i )  ++array[ i ];    if( sizeof(int)\*MAX != write( fd, (void \*)array, sizeof(int)\*MAX ) )  {  printf( "Writing data failed.../n" );  return -1;  }  free( array );  close(fd);  gettimeofday( &tv2, NULL );  printf( "Time of read/write: %dms/n", tv2.tv\_usec-tv1.tv\_usec );    /\*mmap\*/  gettimeofday( &tv1, NULL );  fd = open( "mmap\_test", O\_RDWR );  array = mmap( NULL, sizeof(int)\*MAX, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0 );  for( i=0; i<MAX; ++i )  ++array[ i ];  munmap( array, sizeof(int)\*MAX );    msync( array, sizeof(int)\*MAX, MS\_SYNC );  free( array );  close( fd );  gettimeofday( &tv2, NULL );  printf( "Time of mmap: %dms/n", tv2.tv\_usec-tv1.tv\_usec );  return 0;  } |

输出结果：

Time of read/write: 154ms

Time of mmap: 68ms