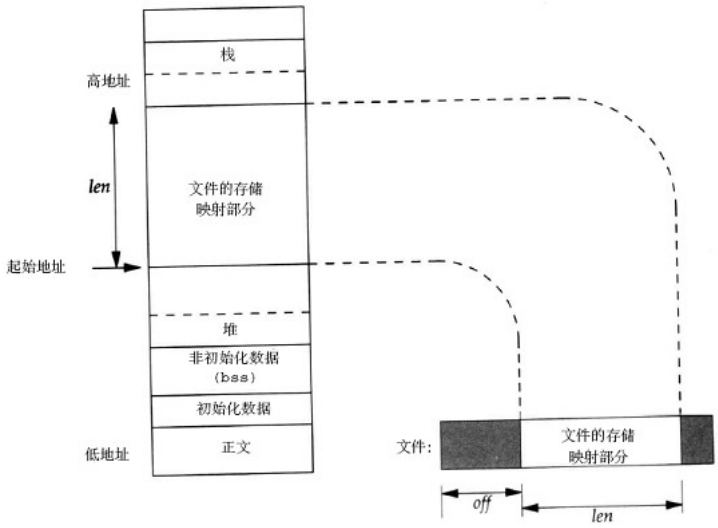
linux内存映射mmap原理分析

内存映射，简而言之就是将用户空间的一段内存区域映射到内核空间，映射成功后，用户对这段内存区域的修改可以直接反映到内核空间，同样，内核空间对这段区域的修改也直接反映用户空间。那么对于内核空间<---->用户空间两者之间需要大量数据传输等操作的话效率是非常高的。

以下是一个把普遍文件映射到用户空间的内存区域的示意图。

图一：



二、基本函数

mmap函数是unix/linux下的系统调用，详细内容可参考《Unix Netword programming》卷二12.2节。

mmap系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。而Posix或系统V的纯粹用于共享目的，当然mmap()实现共享内存也是其主要应用之一。

mmap系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调write（）等操作。mmap并不分配空间，只是将文件映射到调用进程的地址空间里（但是会占掉你的 virutal memory），然后你就可以用memcpy等操作写文件，而不用write。内存中的内容并不会立即更新到文件中，而是有一段时间的延迟，你可以调用msync()来显式同步一下，这样你所写的内容就能立即保存到文件里了.这点应该和驱动相关。不来写文件这种方式没办法增加文件的长度，因为要映射的长度在调用mmap()的时候就决定了.如果想取消内存映射，可以调用munmap()来取消内存映射

```
void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset)
```

mmap用于把文件映射到内存空间中，简单说mmap就是把一个文件的内容在内存里面做一个映像。映射成功后，用户对这段内存区域的修改可以直接反映到内核空间，同这段区域的修改也直接反映用户空间。那么对于内核空间<---->用户空间两者之间需要大量数据传输等操作的话效率是非常高的。

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。

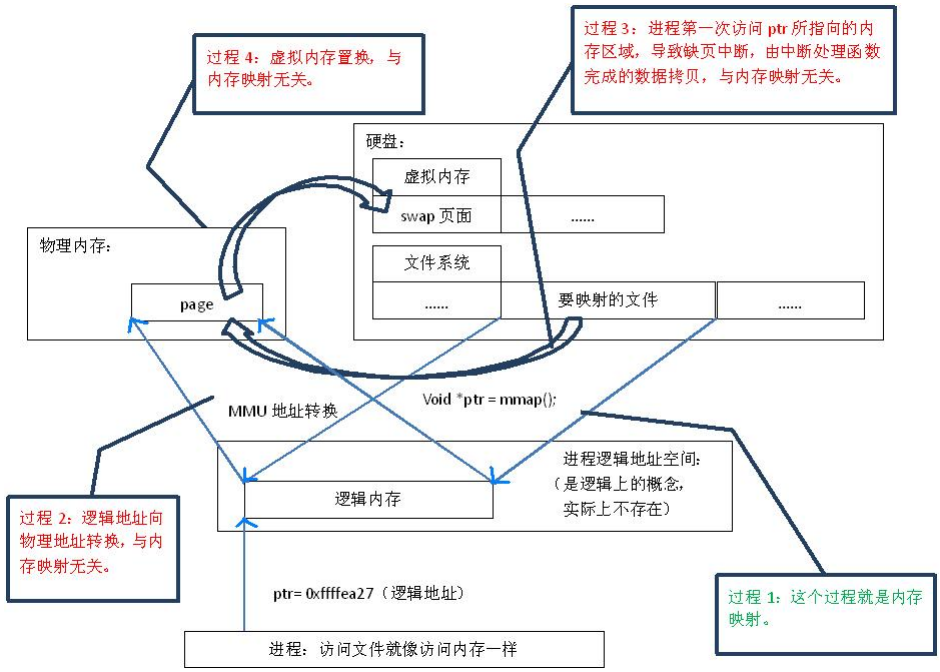


图1.内存映射原理

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

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。

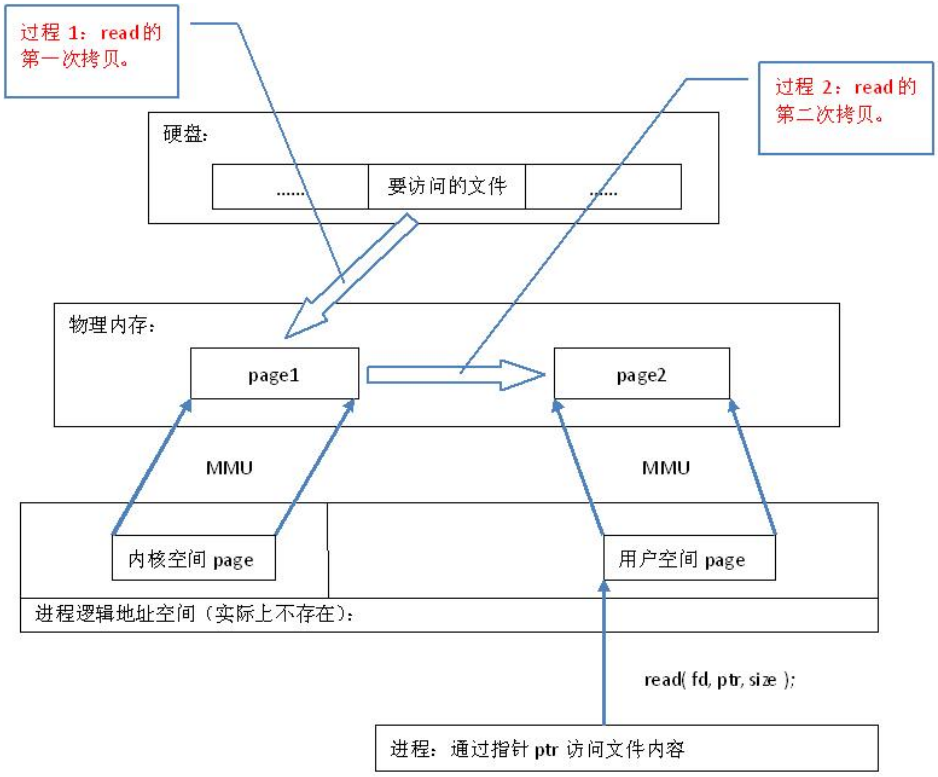


图2.read系统调用原理

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
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;

```

```
33 struct timeval tv1, tv2;
34
35 int *array = (int *)malloc( sizeof(int)*MAX );
36
37
38
39 /*read*/
40
41
42
43 gettimeofday( &tv1, NULL );
44
45 fd = open( "mmap_test", O_RDWR );
46
47 if( sizeof(int)*MAX != read( fd, (void *)array, sizeof(int)*MAX ) )
48
49 {
50
51 printf( "Reading data failed.../n" );
52
53 return -1;
54
55 }
56
57 for( i=0; i<MAX; ++i )
58
59
60
61 ++array[ i ];
62
63 if( sizeof(int)*MAX != write( fd, (void *)array, sizeof(int)*MAX ) )
64
65 {
66
67 printf( "Writing data failed.../n" );
68
69 return -1;
70
71 }
72
73 free( array );
74
75 close( fd );
76
77 gettimeofday( &tv2, NULL );
78
79 printf( "Time of read/write: %dms/n", tv2.tv_usec-tv1.tv_usec );
80
81
82
83 /*mmap*/
84
85
86
87 gettimeofday( &tv1, NULL );
88
89 fd = open( "mmap_test", O_RDWR );
90
91 array = mmap( NULL, sizeof(int)*MAX, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0 );
92
93 for( i=0; i<MAX; ++i )
94
95
96
97 ++array[ i ];
98
99 munmap( array, sizeof(int)*MAX );
100
```

```
101 msync( array, sizeof(int)*MAX, MS_SYNC );
102
103 free( array );
104
105 close( fd );
106
107 gettimeofday( &tv2, NULL );
108
109 printf( "Time of mmap: %dms/n", tv2.tv_usec-tv1.tv_usec );
110
111
112
113 return 0;
114
115 }
```

输出结果：

```
Time of read/write: 154ms

Time of mmap: 68ms
```