Linux共享内存详解（上）

共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

Linux的2.2.x内核支持多种共享内存方式，如mmap()系统调用，Posix共享内存，以及系统V共享内存。linux发行版本如Redhat 8.0支持mmap()系统调用及系统V共享内存，但还没实现Posix共享内存，本文将主要介绍mmap()系统调用及系统V共享内存API的原理及应用。

一、内核怎样保证各个进程寻址到同一个共享内存区域的内存页面

1、page cache及swap cache中页面的区分：一个被访问文件的物理页面都驻留在page cache或swap cache中，一个页面的所有信息由struct page来描述。struct page中有一个域为指针mapping，它指向一个struct address_space类型结构。page cache或swap cache中的所有页面就是根据address_space结构以及一个偏移量来区分的。

2、文件与address_space结构的对应：一个具体的文件在打开后，内核会在内存中为之建立一个struct inode结构，其中的i_mapping域指向一个address_space结构。这样，一个文件就对应一个address_space结构，一个address_space与一个偏移量能够确定一个page cache 或swap cache中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。

3、进程调用mmap()时，只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。

用户操作

[留言] [发消息] [加为好友]

订阅我的博客

0 位读者 POWERED BY FEEDSKY

订阅 订阅到鲜果 订阅到Google 订阅到抓虾

niurou9527的公告

文章分类

C/C++

Linux相关

pro c

存档

2009年06月(2)

2009年05月(1)

2009年03月(2)

2008年12月(6)

2008年11月(2)

2008年10月(7)

4、对于共享内存映射情况，缺页异常处理程序首先在swap cache中寻找目标页（符合address_space以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区(swap area)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到page cache中。进程最终将更新进程页表。
注：对于映射普通文件情况（非共享映射），缺页异常处理程序首先会在page cache中根据address_space以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新。

5、所有进程在映射同一个共享内存区域时，情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。
注：一个共享内存区域可以看作是特殊文件系统shm中的一个文件，shm的安装点在交换区上。

上面涉及到了一些数据结构，围绕数据结构理解问题会容易一些。

二、mmap()及其相关系统调用

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，进程可以向访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。

注：实际上，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的共享内存IPC则纯粹用于共享目的，当然mmap()实现共享内存也是其主要应用之一。

1、mmap()系统调用形式如下：

void* mmap (void * addr , size_t len , int prot , int flags , int fd , off_t offset)
参数fd为即将映射到进程空间的文件描述字，一般由open()返回，同时，fd可以指定为-1，此时须指定flags参数中的MAP_ANON，表明进行的是匿名映射（不涉及具体的文件名，避免了文件的创建及打开，很显然只能用于具有亲缘关系的进程间通信）。len是映射到调用进程地址空间的字节数，它从被映射文件开头offset个字节开始算起。prot 参数指定共享内存的访问权限。可取如下几个值的或：PROT_READ（可读），PROT_WRITE （可写），PROT_EXEC （可执行），PROT_NONE（不可访问）。flags由以下几个常值指定：MAP_SHARED，MAP_PRIVATE，MAP_FIXED，其中，MAP_SHARED，MAP_PRIVATE必选其一，而MAP_FIXED则不推荐使用。offset参数一般设为0，表示从文件头开始映射。参数addr指定文件应被映射到进程空间的起始地址，一般被指定一个空指针，此时选择起始地址的任务留给内核来完成。函数的返回值为最后文件映射到进程空间的地址，进程可直接操作起始地址为该值的有效地址。这里不再详细介绍mmap()的参数，读者可参考mmap()手册页获得进一步的信息。

2、系统调用mmap()用于共享内存的两种方式：

（1）使用普通文件提供的内存映射：适用于任何进程之间；此时，需要打开或创建一个文件，然后再调用mmap(

)；典型调用代码如下：

```
 fd=open(name, flag, mode);
if(fd<0)
 ...
```

ptr=mmap(NULL, len ，PROT_READ|PROT_WRITE, MAP_SHARED ，fd ，0)；通过mmap()实现共享内存的通信方式有许多特点和要注意的地方，我们将在范例中进行具体说明。

（2）使用特殊文件提供匿名内存映射：适用于具有亲缘关系的进程之间；由于父子进程特殊的亲缘关系，在父进程中先调用mmap()，然后调用fork()。那么在调用fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。注意，这里不是一般的继承关系。一般来说，子进程单独维护从父进程继承下来的一些变量。而mmap()返回的地址，却由父子进程共同维护。
对于具有亲缘关系的进程实现共享内存最好的方式应该是采用匿名内存映射的方式。此时，不必指定具体的文件，只要设置相应的标志即可，参见范例2。

3、系统调用munmap()

int munmap(void * addr, size_t len)
该调用在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射区的大小。当映射关系解除后，对原来映射地址的访问将导致段错误发生。

4、系统调用msync()

int msync (void * addr , size_t len, int flags)
一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后才执行该操作。可以通过调用msync()实现磁盘上文件内容与共享内存区的内容一致

三、mmap()范例

下面将给出使用mmap()的两个范例：范例1给出两个进程通过映射普通文件实现共享内存通信；范例2给出父子进程通过匿名映射实现共享内存。系统调用mmap()有许多有趣的地方，下面是通过mmap（）映射普通文件实现进程间的通信的范例，我们通过该范例来说明mmap()实现共享内存的特点及注意事项。

范例1：两个进程通过映射普通文件实现共享内存通信

范例1包含两个子程序：map_normalfile1.c及map_normalfile2.c。编译两个程序，可执行文件分别为map_n

ormalfile1及map_normalfile2。两个程序通过命令行参数指定同一个文件来实现共享内存方式的进程间通信。map_normalfile2试图打开命令行参数指定的一个普通文件，把该文件映射到进程的地址空间，并对映射后的地址空间进行写操作。map_normalfile1把命令行参数指定的文件映射到进程地址空间，然后对映射后的地址空间执行读操作。这样，两个进程通过命令行参数指定同一个文件来实现共享内存方式的进程间通信。

下面是两个程序代码:

```
/*-------------map_normalfile1.c-----------*/
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
typedef struct{
 char name[4];
 int  age;
}people;

main(int argc, char** argv) // map a normal file as shared mem:
{
 int fd,i;
 people *p_map;
 char temp;

 fd=open(argv[1],O_CREAT|O_RDWR|O_TRUNC,00777);
 lseek(fd,sizeof(people)*5-1,SEEK_SET);
 write(fd,"",1);

 p_map = (people*) mmap( NULL,sizeof(people)*10,PROT_READ|PROT_WRITE,MAP_SHARED,fd,
0 );
 close( fd );
 temp = 'a';
 for(i=0; i<10; i++)
 {
  temp += 1;
  memcpy( ( *(p_map+i) ).name, &temp,2 );
  ( *(p_map+i) ).age = 20+i;
 }
 printf(" initialize over \n ");
```

```
 sleep(10);

 munmap( p_map, sizeof(people)*10 );
 printf( "umap ok \n" );
}

/*-------------map_normalfile2.c-----------*/
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
typedef struct{
 char name[4];
 int  age;
}people;

main(int argc, char** argv) // map a normal file as shared mem:
{
 int fd,i;
 people *p_map;
 fd=open( argv[1],O_CREAT|O_RDWR,00777 );
 p_map = (people*)mmap(NULL,sizeof(people)*10,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0)
;
 for(i = 0;i<10;i++)
 {
 printf( "name: %s age %d;\n",(*(p_map+i)).name, (*(p_map+i)).age );

 }
 munmap( p_map,sizeof(people)*10 );
}
```

map_normalfile1.c首先定义了一个people数据结构，（在这里采用数据结构的方式是因为，共享内存区的数据往往是有固定格式的，这由通信的各个进程决定，采用结构的方式有普遍代表性）。map_normfile1首先打开或创建一个文件，并把文件的长度设置为5个people结构大小。然后从mmap()的返回地址开始，设置了10个people结构。然后，进程睡眠10秒钟，等待其他进程映射同一个文件，最后解除映射。

map_normfile2.c只是简单的映射一个文件，并以people数据结构的格式从mmap()返回的地址处读取10个people结构，并输出读取的值，然后解除映射。

分别把两个程序编译成可执行文件map_normalfile1和map_normalfile2后，在一个终端上先运行./map_normalfile2 /tmp/test_shm，程序输出结果如下：

```
initialize over
umap ok
```

在map_normalfile1输出initialize over 之后，输出umap ok之前，在另一个终端上运行map_normalfile2 /tmp/test_shm，将会产生如下输出(为了节省空间，输出结果为稍作整理后的结果)：

```
name: b age 20; name: c age 21; name: d age 22; name: e age 23; name: f age 24;
name: g age 25; name: h age 26; name: I age 27; name: j age 28; name: k age 29;
```

在map_normalfile1 输出umap ok后，运行map_normalfile2则输出如下结果：

```
name: b age 20; name: c age 21; name: d age 22; name: e age 23; name: f age 24;
name: age 0; name: age 0; name: age 0; name: age 0; name: age 0;
```

从程序的运行结果中可以得出的结论

1、 最终被映射文件的内容的长度不会超过文件本身的初始大小，即映射不能改变文件的大小；

2、 可以用于进程通信的有效地址空间大小大体上受限于被映射文件的大小，但不完全受限于文件大小。打开文件被截短为5个people结构大小，而在map_normalfile1中初始化了10个people数据结构，在恰当时候（map_normalfile1输出initialize over 之后，输出umap ok之前）调用map_normalfile2会发现map_normalfile2将输出全部10个people结构的值，后面将给出详细讨论。
注：在linux中，内存的保护是以页为基本单位的，即使被映射文件只有一个字节大小，内核也会为映射分配一个页面大小的内存。当被映射文件小于一个页面大小时，进程可以对从mmap()返回地址开始的一个页面大小进行访问，而不会出错；但是，如果对一个页面以外的地址空间进行访问，则导致错误发生，后面将进一步描述。因此，可用于进程间通信的有效地址空间大小不会超过文件大小及一个页面大小的和。

3、 文件一旦被映射后，调用mmap()的进程对返回地址的访问是对某一内存区域的访问，暂时脱离了磁盘上文件的影响。所有对mmap()返回地址空间的操作只在内存中有意义，只有在调用了munmap()后或者msync()时，才把内存中的相应内容写回磁盘文件，所写内容仍然不能超过文件的大小。

范例2：父子进程通过匿名映射实现共享内存

```
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
typedef struct{
```

```
 char name[4];
 int  age;
}people;
main(int argc, char** argv)
{
 int i;
 people *p_map;
 char temp;
 p_map=(people*)mmap(NULL,sizeof(people)*10,PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANO
NYMOUS,-1,0);
 if(fork() == 0)
 {
  sleep(2);
  for(i = 0;i<5;i++)
   printf("child read: the %d people's age is %d\n",i+1,(*(p_map+i)).age);
  (*p_map).age = 100;
  munmap(p_map,sizeof(people)*10); //实际上，进程终止时，会自动解除映射。
  exit();
 }
 temp = 'a';
 for(i = 0;i<5;i++)
 {
  temp += 1;
  memcpy((*(p_map+i)).name, &temp,2);
  (*(p_map+i)).age=20+i;
 }

 sleep(5);
 printf( "parent read: the first people,s age is %d\n",(*p_map).age );
 printf("umap\n");
 munmap( p_map,sizeof(people)*10 );
 printf( "umap ok\n" );
}
```

考察程序的输出结果，体会父子进程匿名共享内存：

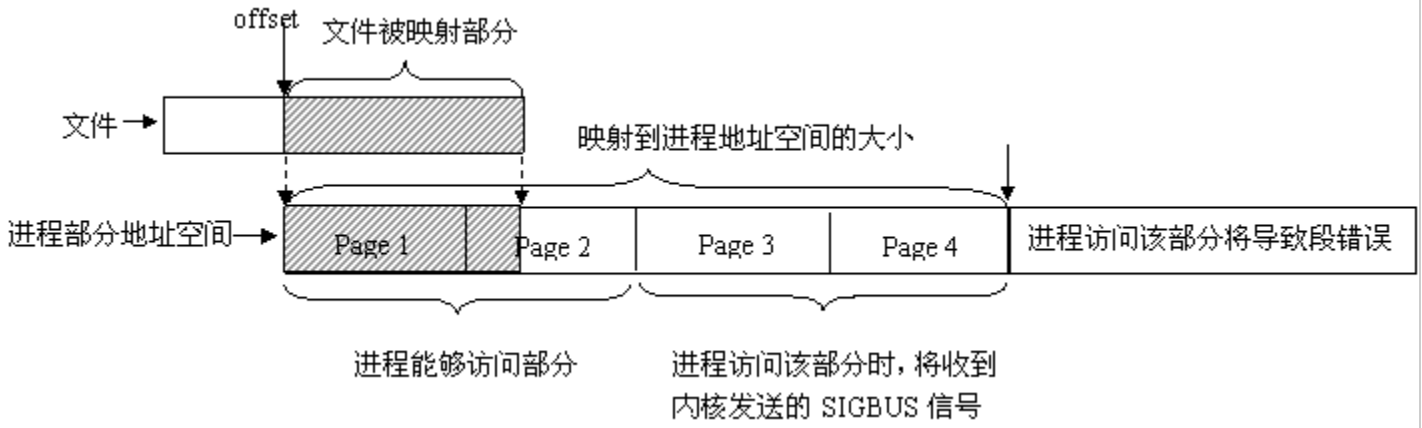
```
child read: the 1 people's age is 20
child read: the 2 people's age is 21
```

```
child read: the 3 people's age is 22
child read: the 4 people's age is 23
child read: the 5 people's age is 24

parent read: the first people,s age is 100
umap
umap ok
```

四、对mmap()返回地址的访问

前面对范例运行结构的讨论中已经提到，linux采用的是页式管理机制。对于用mmap()映射普通文件来说，进程会在自己的地址空间新增一块空间，空间大小由mmap()的len参数指定，注意，进程并不一定能够对全部新增空间都能进行有效访问。进程能够访问的有效地址大小取决于文件被映射部分的大小。简单的说，能够容纳文件被映射部分大小的最少页面个数决定了进程从mmap()返回的地址开始，能够有效访问的地址空间大小。超过这个空间大小，内核会根据超过的严重程度返回发送不同的信号给进程。可用如下图示说明：



注意：文件被映射部分而不是整个文件决定了进程能够访问的空间大小，另外，如果指定文件的偏移部分，一定要注意为页面大小的整数倍。下面是对进程映射地址空间的访问范例：

```
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
```

```
typedef struct{
 char name[4];
 int  age;
}people;

main(int argc, char** argv)
{
 int fd,i;
 int pagesize,offset;
 people *p_map;

 pagesize = sysconf(_SC_PAGESIZE);
 printf("pagesize is %d\n",pagesize);
 fd = open(argv[1],O_CREAT|O_RDWR|O_TRUNC,00777);
 lseek(fd,pagesize*2-100,SEEK_SET);
 write(fd,"",1);
 offset = 0; //此处offset = 0编译成版本1; offset = pagesize编译成版本2
 p_map = (people*)mmap(NULL,pagesize*3,PROT_READ|PROT_WRITE,MAP_SHARED,fd,offset);
 close(fd);

 for(i = 1; i<10; i++)
 {
  (*(p_map+pagesize/sizeof(people)*i-2)).age = 100;
  printf("access page %d over\n",i);
  (*(p_map+pagesize/sizeof(people)*i-1)).age = 100;
  printf("access page %d edge over, now begin to access page %d\n",i, i+1);
  (*(p_map+pagesize/sizeof(people)*i)).age = 100;
  printf("access page %d over\n",i+1);
 }
 munmap(p_map,sizeof(people)*10);
}
```

如程序中所注释的那样，把程序编译成两个版本，两个版本主要体现在文件被映射部分的大小不同。文件的大小介于一个页面与两个页面之间（大小为：pagesize*2-99），版本1的被映射部分是整个文件，版本2的文件被映射部分是文件大小减去一个页面后的剩余部分，不到一个页面大小(大小为：pagesize-99)。程序中试图访问每一个页面边界，两个版本都试图在进程空间中映射pagesize*3的字节数。

版本1的输出结果如下：

```
pagesize is 4096
access page 1 over
access page 1 edge over, now begin to access page 2
access page 2 over
access page 2 over
access page 2 edge over, now begin to access page 3
Bus error  //被映射文件在进程空间中覆盖了两个页面，此时，进程试图访问第三个页面
```

版本2的输出结果如下：

```
pagesize is 4096
access page 1 over
access page 1 edge over, now begin to access page 2
Bus error  //被映射文件在进程空间中覆盖了一个页面，此时，进程试图访问第二个页面
```

结论：采用系统调用mmap()实现进程间通信是很方便的，在应用层上接口非常简洁。内部实现机制区涉及到了linux存储管理以及文件系统等方面的内容，可以参考一下相关重要数据结构来加深理解。在本专题的后面部分，将介绍系统v共享内存的实现。

参考文献：

[1] Understanding the Linux Kernel, 2nd Edition, By Daniel P. Bovet, Marco Cesati ，对各主题阐述得重点突出，脉络清晰。

[2] UNIX网络编程第二卷：进程间通信，作者：W.Richard Stevens，译者：杨继张，清华大学出版社。对mmap()有详细阐述。

[3] Linux内核源代码情景分析（上），毛德操、胡希明著，浙江大学出版社，给出了mmap()相关的源代码分析。

[4]mmap()手册

关于作者：

郑彦兴，国防科大攻读博士学位。联系方式： mlinux@163.com

发表于 @ 2008年10月22日 10:32:00 | 评论(0) | 举报 | 收藏

旧一篇:守护进程的编写 | 新一篇:Linux共享内存详解（下）

发表评论　“评论王争夺赛”活动，第4期开始啦！

表 情:

评论内容:

用 户 名: huapuyu7　匿名评论

发表评论