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。

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。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html?ca=drs-#ibm-pcon)

**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。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html?ca=drs-#ibm-pcon)

**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\_normalfile1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\_ANONYMOUS,-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" );  } |

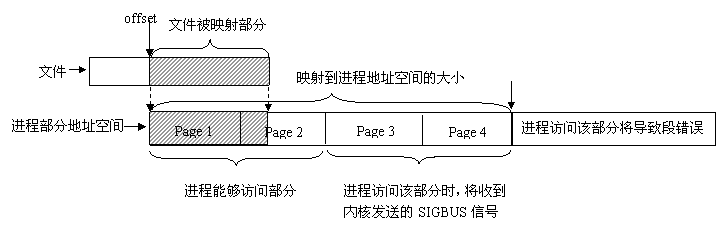
考察程序的输出结果，体会父子进程匿名共享内存：

|  |
| --- |
| 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 |

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html?ca=drs-#ibm-pcon)

**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。

**参考资料**

[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

系统调用mmap()通过映射一个普通文件实现共享内存。系统V则是通过映射特殊文件系统shm中的文件实现进程间的共享内存通信。也就是说，每个共享内存区域对应特殊文件系统shm中的一个文件（这是通过shmid\_kernel结构联系起来的），后面还将阐述。

**1、系统V共享内存原理**

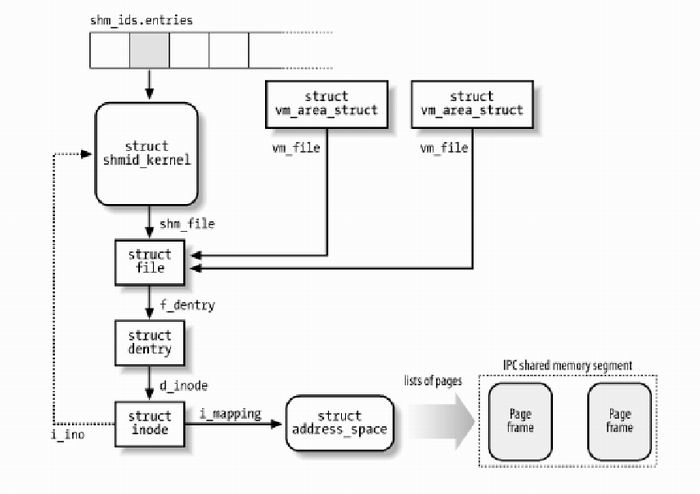
进程间需要共享的数据被放在一个叫做IPC共享内存区域的地方，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。系统V共享内存通过shmget获得或创建一个IPC共享内存区域，并返回相应的标识符。内核在保证shmget获得或创建一个共享内存区，初始化该共享内存区相应的shmid\_kernel结构注同时，还将在特殊文件系统shm中，创建并打开一个同名文件，并在内存中建立起该文件的相应dentry及inode结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享内存区）。所有这一切都是系统调用shmget完成的。

注：每一个共享内存区都有一个控制结构struct shmid\_kernel，shmid\_kernel是共享内存区域中非常重要的一个数据结构，它是存储管理和文件系统结合起来的桥梁，定义如下：

|  |
| --- |
| struct shmid\_kernel /\* private to the kernel \*/  {  struct kern\_ipc\_perm shm\_perm;  struct file \* shm\_file;  int id;  unsigned long shm\_nattch;  unsigned long shm\_segsz;  time\_t shm\_atim;  time\_t shm\_dtim;  time\_t shm\_ctim;  pid\_t shm\_cprid;  pid\_t shm\_lprid;  }; |

该结构中最重要的一个域应该是shm\_file，它存储了将被映射文件的地址。每个共享内存区对象都对应特殊文件系统shm中的一个文件，一般情况下，特殊文件系统shm中的文件是不能用read()、write()等方法访问的，当采取共享内存的方式把其中的文件映射到进程地址空间后，可直接采用访问内存的方式对其访问。

这里我们采用[1]中的图表给出与系统V共享内存相关数据结构：



正如消息队列和信号灯一样，内核通过数据结构struct ipc\_ids shm\_ids维护系统中的所有共享内存区域。上图中的shm\_ids.entries变量指向一个ipc\_id结构数组，而每个ipc\_id结构数组中有个指向kern\_ipc\_perm结构的指针。到这里读者应该很熟悉了，对于系统V共享内存区来说，kern\_ipc\_perm的宿主是shmid\_kernel结构，shmid\_kernel是用来描述一个共享内存区域的，这样内核就能够控制系统中所有的共享区域。同时，在shmid\_kernel结构的file类型指针shm\_file指向文件系统shm中相应的文件，这样，共享内存区域就与shm文件系统中的文件对应起来。

在创建了一个共享内存区域后，还要将它映射到进程地址空间，系统调用shmat()完成此项功能。由于在调用shmget()时，已经创建了文件系统shm中的一个同名文件与共享内存区域相对应，因此，调用shmat()的过程相当于映射文件系统shm中的同名文件过程，原理与mmap()大同小异。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html#ibm-pcon)

**2、系统V共享内存API**

对于系统V共享内存，主要有以下几个API：shmget()、shmat()、shmdt()及shmctl()。

|  |
| --- |
| #include <sys/ipc.h>  #include <sys/shm.h> |

shmget（）用来获得共享内存区域的ID，如果不存在指定的共享区域就创建相应的区域。shmat()把共享内存区域映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区域进行访问操作。shmdt()调用用来解除进程对共享内存区域的映射。shmctl实现对共享内存区域的控制操作。这里我们不对这些系统调用作具体的介绍，读者可参考相应的手册页面，后面的范例中将给出它们的调用方法。

注：shmget的内部实现包含了许多重要的系统V共享内存机制；shmat在把共享内存区域映射到进程空间时，并不真正改变进程的页表。当进程第一次访问内存映射区域访问时，会因为没有物理页表的分配而导致一个缺页异常，然后内核再根据相应的存储管理机制为共享内存映射区域分配相应的页表。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html#ibm-pcon)

**3、系统V共享内存限制**

在/proc/sys/kernel/目录下，记录着系统V共享内存的一下限制，如一个共享内存区的最大字节数shmmax，系统范围内最大共享内存区标识符数shmmni等，可以手工对其调整，但不推荐这样做。

在[2]中，给出了这些限制的测试方法，不再赘述。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html#ibm-pcon)

**4、系统V共享内存范例**

本部分将给出系统V共享内存API的使用方法，并对比分析系统V共享内存机制与mmap()映射普通文件实现共享内存之间的差异，首先给出两个进程通过系统V共享内存通信的范例：

|  |
| --- |
| /\*\*\*\*\* testwrite.c \*\*\*\*\*\*\*/  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/types.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  } people;  main(int argc, char\*\* argv)  {  int shm\_id,i;  key\_t key;  char temp;  people \*p\_map;  char\* name = "/dev/shm/myshm2";  key = ftok(name,0);  if(key==-1)  perror("ftok error");  shm\_id=shmget(key,4096,IPC\_CREAT);  if(shm\_id==-1)  {  perror("shmget error");  return;  }  p\_map=(people\*)shmat(shm\_id,NULL,0);  temp='a';  for(i = 0;i<10;i++)  {  temp+=1;  memcpy((\*(p\_map+i)).name,&temp,1);  (\*(p\_map+i)).age=20+i;  }  if(shmdt(p\_map)==-1)  perror(" detach error ");  }  /\*\*\*\*\*\*\*\*\*\* testread.c \*\*\*\*\*\*\*\*\*\*\*\*/  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/types.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  } people;  main(int argc, char\*\* argv)  {  int shm\_id,i;  key\_t key;  people \*p\_map;  char\* name = "/dev/shm/myshm2";  key = ftok(name,0);  if(key == -1)  perror("ftok error");  shm\_id = shmget(key,4096,IPC\_CREAT);  if(shm\_id == -1)  {  perror("shmget error");  return;  }  p\_map = (people\*)shmat(shm\_id,NULL,0);  for(i = 0;i<10;i++)  {  printf( "name:%s\n",(\*(p\_map+i)).name );  printf( "age %d\n",(\*(p\_map+i)).age );  }  if(shmdt(p\_map) == -1)  perror(" detach error ");  } |

testwrite.c创建一个系统V共享内存区，并在其中写入格式化数据；testread.c访问同一个系统V共享内存区，读出其中的格式化数据。分别把两个程序编译为testwrite及testread，先后执行./testwrite及./testread 则./testread输出结果如下：

|  |
| --- |
| 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; |

通过对试验结果分析，对比系统V与mmap()映射普通文件实现共享内存通信，可以得出如下结论：

1、 系统V共享内存中的数据，从来不写入到实际磁盘文件中去；而通过mmap()映射普通文件实现的共享内存通信可以指定何时将数据写入磁盘文件中。 注：前面讲到，系统V共享内存机制实际是通过映射特殊文件系统shm中的文件实现的，文件系统shm的安装点在交换分区上，系统重新引导后，所有的内容都丢失。

2、 系统V共享内存是随内核持续的，即使所有访问共享内存的进程都已经正常终止，共享内存区仍然存在（除非显式删除共享内存），在内核重新引导之前，对该共享内存区域的任何改写操作都将一直保留。

3、 通过调用mmap()映射普通文件进行进程间通信时，一定要注意考虑进程何时终止对通信的影响。而通过系统V共享内存实现通信的进程则不然。 注：这里没有给出shmctl的使用范例，原理与消息队列大同小异。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index2.html#ibm-pcon)

**结论：**

共享内存允许两个或多个进程共享一给定的存储区，因为数据不需要来回复制，所以是最快的一种进程间通信机制。共享内存可以通过mmap()映射普通文件（特殊情况下还可以采用匿名映射）机制实现，也可以通过系统V共享内存机制实现。应用接口和原理很简单，内部机制复杂。为了实现更安全通信，往往还与信号灯等同步机制共同使用。

共享内存涉及到了存储管理以及文件系统等方面的知识，深入理解其内部机制有一定的难度，关键还要紧紧抓住内核使用的重要数据结构。系统V共享内存是以文件的形式组织在特殊文件系统shm中的。通过shmget可以创建或获得共享内存的标识符。取得共享内存标识符后，要通过shmat将这个内存区映射到本进程的虚拟地址空间。

**参考资料**

[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]shmget、shmat、shmctl、shmdt手册