[Linux进程间通信——使用共享内存](http://blog.csdn.net/ljianhui/article/details/10253345)

一、什么是共享内存

顾名思义，共享内存就是允许两个不相关的进程访问同一个逻辑内存。共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式。不同进程之间共享的内存通常安排为同一段物理内存。进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用C语言函数malloc分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

特别提醒：共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。所以我们通常需要用其他的机制来同步对共享内存的访问，例如前面说到的信号量。有关信号量的更多内容，可以查阅我的另一篇文章：[Linux进程间通信——使用信号量](http://blog.csdn.net/ljianhui/article/details/10243617)

二、共享内存的使得

与信号量一样，在Linux中也提供了一组函数接口用于使用共享内存，而且使用共享共存的接口还与信号量的非常相似，而且比使用信号量的接口来得简单。它们声明在头文件 sys/shm.h中。

1、shmget函数

该函数用来创建共享内存，它的原型为：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. **int** shmget(key\_t key, **size\_t** size, **int** shmflg);

第一个参数，与信号量的semget函数一样，程序需要提供一个参数key（非0整数），它有效地为共享内存段命名，shmget函数成功时返回一个与key相关的共享内存标识符（非负整数），用于后续的共享内存函数。调用失败返回-1.

不相关的进程可以通过该函数的返回值访问同一共享内存，它代表程序可能要使用的某个资源，程序对所有共享内存的访问都是间接的，程序先通过调用shmget函数并提供一个键，再由系统生成一个相应的共享内存标识符（shmget函数的返回值），只有shmget函数才直接使用信号量键，所有其他的信号量函数使用由semget函数返回的信号量标识符。

第二个参数，size以字节为单位指定需要共享的内存容量

第三个参数，shmflg是权限标志，它的作用与open函数的mode参数一样，如果要想在key标识的共享内存不存在时，创建它的话，可以与IPC\_CREAT做或操作。共享内存的权限标志与文件的读写权限一样，举例来说，0644,它表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。

2、shmat函数

第一次创建完共享内存时，它还不能被任何进程访问，shmat函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间。它的原型如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. **void** \*shmat(**int** shm\_id, **const** **void** \*shm\_addr, **int** shmflg);

第一个参数，shm\_id是由shmget函数返回的共享内存标识。

第二个参数，shm\_addr指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。

第三个参数，shm\_flg是一组标志位，通常为0。

调用成功时返回一个指向共享内存第一个字节的指针，如果调用失败返回-1.

3、shmdt函数

该函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。它的原型如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. **int** shmdt(**const** **void** \*shmaddr);

参数shmaddr是shmat函数返回的地址指针，调用成功时返回0，失败时返回-1.

4、shmctl函数

与信号量的semctl函数一样，用来控制共享内存，它的原型如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. **int** shmctl(**int** shm\_id, **int** command, **struct** shmid\_ds \*buf);

第一个参数，shm\_id是shmget函数返回的共享内存标识符。

第二个参数，command是要采取的操作，它可以取下面的三个值 ：

    IPC\_STAT：把shmid\_ds结构中的数据设置为共享内存的当前关联值，即用共享内存的当前关联值覆盖shmid\_ds的值。

    IPC\_SET：如果进程有足够的权限，就把共享内存的当前关联值设置为shmid\_ds结构中给出的值

    IPC\_RMID：删除共享内存段

第三个参数，buf是一个结构指针，它指向共享内存模式和访问权限的结构。

shmid\_ds结构至少包括以下成员：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. **struct** shmid\_ds
2. {
3. uid\_t shm\_perm.uid;
4. uid\_t shm\_perm.gid;
5. mode\_t shm\_perm.mode;
6. };

三、使用共享内存进行进程间通信

说了这么多，又到了实战的时候了。下面就以两个不相关的进程来说明进程间如何通过共享内存来进行通信。其中一个文件shmread.c创建共享内存，并读取其中的信息，另一个文件shmwrite.c向共享内存中写入数据。为了方便操作和数据结构的统一，为这两个文件定义了相同的数据结构，定义在文件shmdata.c中。结构shared\_use\_st中的written作为一个可读或可写的标志，非0：表示可读，0表示可写，text则是内存中的文件。

shmdata.h的源代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. #ifndef \_SHMDATA\_H\_HEADER
2. #define \_SHMDATA\_H\_HEADER
4. #define TEXT\_SZ 2048
6. **struct** shared\_use\_st
7. {
8. **int** written;//作为一个标志，非0：表示可读，0表示可写
9. **char** text[TEXT\_SZ];//记录写入和读取的文本
10. };
12. #endif

源文件shmread.c的源代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. #include <unistd.h>
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <sys/shm.h>
5. #include "shmdata.h"
7. **int** main()
8. {
9. **int** running = 1;//程序是否继续运行的标志
10. **void** \*shm = NULL;//分配的共享内存的原始首地址
11. **struct** shared\_use\_st \*shared;//指向shm
12. **int** shmid;//共享内存标识符
13. //创建共享内存
14. shmid = shmget((key\_t)1234, **sizeof**(**struct** shared\_use\_st), 0666|IPC\_CREAT);
15. **if**(shmid == -1)
16. {
17. fprintf(stderr, "shmget failed\n");
18. exit(EXIT\_FAILURE);
19. }
20. //将共享内存连接到当前进程的地址空间
21. shm = shmat(shmid, 0, 0);
22. **if**(shm == (**void**\*)-1)
23. {
24. fprintf(stderr, "shmat failed\n");
25. exit(EXIT\_FAILURE);
26. }
27. printf("\nMemory attached at %X\n", (**int**)shm);
28. //设置共享内存
29. shared = (**struct** shared\_use\_st\*)shm;
30. shared->written = 0;
31. **while**(running)//读取共享内存中的数据
32. {
33. //没有进程向共享内存定数据有数据可读取
34. **if**(shared->written != 0)
35. {
36. printf("You wrote: %s", shared->text);
37. sleep(rand() % 3);
38. //读取完数据，设置written使共享内存段可写
39. shared->written = 0;
40. //输入了end，退出循环（程序）
41. **if**(strncmp(shared->text, "end", 3) == 0)
42. running = 0;
43. }
44. **else**//有其他进程在写数据，不能读取数据
45. sleep(1);
46. }
47. //把共享内存从当前进程中分离
48. **if**(shmdt(shm) == -1)
49. {
50. fprintf(stderr, "shmdt failed\n");
51. exit(EXIT\_FAILURE);
52. }
53. //删除共享内存
54. **if**(shmctl(shmid, IPC\_RMID, 0) == -1)
55. {
56. fprintf(stderr, "shmctl(IPC\_RMID) failed\n");
57. exit(EXIT\_FAILURE);
58. }
59. exit(EXIT\_SUCCESS);
60. }

源文件shmwrite.c的源代码如下：

**[cpp]** [view plain](http://blog.csdn.net/ljianhui/article/details/10253345) [copy](http://blog.csdn.net/ljianhui/article/details/10253345)

[print?](http://blog.csdn.net/ljianhui/article/details/10253345)[在CODE上查看代码片](https://code.csdn.net/snippets/252538)

1. #include <unistd.h>
2. #include <stdlib.h>
3. #include <stdio.h>
4. #include <string.h>
5. #include <sys/shm.h>
6. #include "shmdata.h"
8. **int** main()
9. {
10. **int** running = 1;
11. **void** \*shm = NULL;
12. **struct** shared\_use\_st \*shared = NULL;
13. **char** buffer[BUFSIZ + 1];//用于保存输入的文本
14. **int** shmid;
15. //创建共享内存
16. shmid = shmget((key\_t)1234, **sizeof**(**struct** shared\_use\_st), 0666|IPC\_CREAT);
17. **if**(shmid == -1)
18. {
19. fprintf(stderr, "shmget failed\n");
20. exit(EXIT\_FAILURE);
21. }
22. //将共享内存连接到当前进程的地址空间
23. shm = shmat(shmid, (**void**\*)0, 0);
24. **if**(shm == (**void**\*)-1)
25. {
26. fprintf(stderr, "shmat failed\n");
27. exit(EXIT\_FAILURE);
28. }
29. printf("Memory attached at %X\n", (**int**)shm);
30. //设置共享内存
31. shared = (**struct** shared\_use\_st\*)shm;
32. **while**(running)//向共享内存中写数据
33. {
34. //数据还没有被读取，则等待数据被读取,不能向共享内存中写入文本
35. **while**(shared->written == 1)
36. {
37. sleep(1);
38. printf("Waiting...\n");
39. }
40. //向共享内存中写入数据
41. printf("Enter some text: ");
42. fgets(buffer, BUFSIZ, stdin);
43. strncpy(shared->text, buffer, TEXT\_SZ);
44. //写完数据，设置written使共享内存段可读
45. shared->written = 1;
46. //输入了end，退出循环（程序）
47. **if**(strncmp(buffer, "end", 3) == 0)
48. running = 0;
49. }
50. //把共享内存从当前进程中分离
51. **if**(shmdt(shm) == -1)
52. {
53. fprintf(stderr, "shmdt failed\n");
54. exit(EXIT\_FAILURE);
55. }
56. sleep(2);
57. exit(EXIT\_SUCCESS);
58. }

再来看看运行的结果：



分析：

1、程序shmread创建共享内存，然后将它连接到自己的地址空间。在共享内存的开始处使用了一个结构struct\_use\_st。该结构中有个标志written，当共享内存中有其他进程向它写入数据时，共享内存中的written被设置为0，程序等待。当它不为0时，表示没有进程对共享内存写入数据，程序就从共享内存中读取数据并输出，然后重置设置共享内存中的written为0，即让其可被shmwrite进程写入数据。

2、程序shmwrite取得共享内存并连接到自己的地址空间中。检查共享内存中的written，是否为0，若不是，表示共享内存中的数据还没有被完，则等待其他进程读取完成，并提示用户等待。若共享内存的written为0，表示没有其他进程对共享内存进行读取，则提示用户输入文本，并再次设置共享内存中的written为1，表示写完成，其他进程可对共享内存进行读操作。

四、关于前面的例子的安全性讨论

这个程序是不安全的，当有多个程序同时向共享内存中读写数据时，问题就会出现。可能你会认为，可以改变一下written的使用方式，例如，只有当written为0时进程才可以向共享内存写入数据，而当一个进程只有在written不为0时才能对其进行读取，同时把written进行加1操作，读取完后进行减1操作。这就有点像文件锁中的读写锁的功能。咋看之下，它似乎能行得通。但是这都不是原子操作，所以这种做法是行不能的。试想当written为0时，如果有两个进程同时访问共享内存，它们就会发现written为0，于是两个进程都对其进行写操作，显然不行。当written为1时，有两个进程同时对共享内存进行读操作时也是如些，当这两个进程都读取完是，written就变成了-1.

要想让程序安全地执行，就要有一种进程同步的进制，保证在进入临界区的操作是原子操作。例如，可以使用前面所讲的信号量来进行进程的同步。因为信号量的操作都是原子性的。

五、使用共享内存的优缺点

1、优点：我们可以看到使用共享内存进行进程间的通信真的是非常方便，而且函数的接口也简单，数据的共享还使进程间的数据不用传送，而是直接访问内存，也加快了程序的效率。同时，它也不像匿名管道那样要求通信的进程有一定的父子关系。

2、缺点：共享内存没有提供同步的机制，这使得我们在使用共享内存进行进程间通信时，往往要借助其他的手段来进行进程间的同步工作。