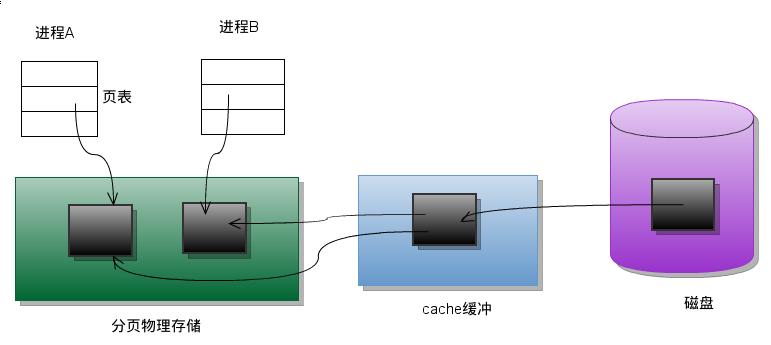
共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式, 因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据: 一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

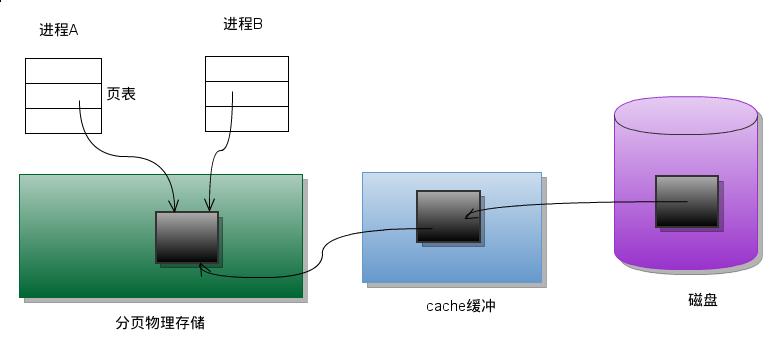
**一. 传统文件访问**

UNIX访问文件的传统方法是用open打开它们, 如果有多个进程访问同一个文件, 则每一个进程在自己的地址空间都包含有该文件的副本,这不必要地浪费了存储空间. 下图说明了两个进程同时读一个文件的同一页的情形. 系统要将该页从磁盘读到高速缓冲区中, 每个进程再执行一个存储器内的复制操作将数据从高速缓冲区读到自己的地址空间.



**二. 共享存储映射**

现在考虑另一种处理方法:　进程A和进程B都将该页映射到自己的地址空间, 当进程A第一次访问该页中的数据时, 它生成一个缺页中断. 内核此时读入这一页到内存并更新页表使之指向它.以后, 当进程B访问同一页面而出现缺页中断时, 该页已经在内存, 内核只需要将进程B的页表登记项指向次页即可. 如下图所示:



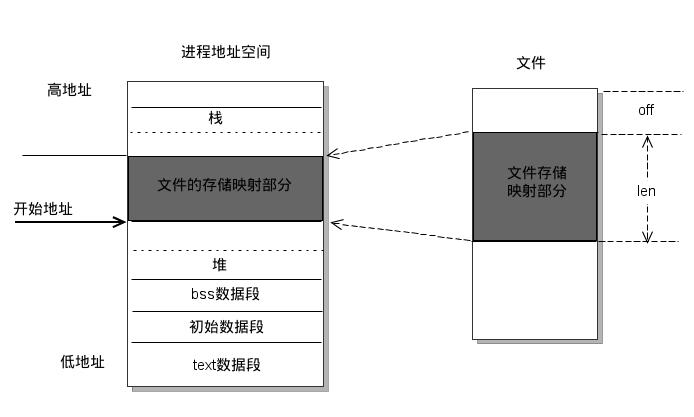
**三、mmap()及其相关系统调用**

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。

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 )

mmap的作用是映射文件描述符fd指定文件的 [off,off + len]区域至调用进程的[addr, addr + len]的内存区域, 如下图所示:



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。

**四. mmap的两个例子**

范例中使用的测试文件 data.txt:

**Xml代码  [收藏代码](javascript:void())**

1. aaaaaaaaa
2. bbbbbbbbb
3. ccccccccc
4. ddddddddd

1 通过共享映射的方式修改文件

**C代码  [收藏代码](javascript:void())**

1. #include <sys/mman.h>
2. #include <sys/stat.h>
3. #include <fcntl.h>
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <unistd.h>
7. #include <error.h>
9. #define BUF\_SIZE 100
11. **int** main(**int** argc, **char** \*\*argv)
12. {
13. **int** fd, nread, i;
14. **struct** stat sb;
15. **char** \*mapped, buf[BUF\_SIZE];
17. **for** (i = 0; i < BUF\_SIZE; i++) {
18. buf[i] = '#';
19. }
21. /\* 打开文件 \*/
22. **if** ((fd = open(argv[1], O\_RDWR)) < 0) {
23. perror("open");
24. }
26. /\* 获取文件的属性 \*/
27. **if** ((fstat(fd, &sb)) == -1) {
28. perror("fstat");
29. }
31. /\* 将文件映射至进程的地址空间 \*/
32. **if** ((mapped = (**char** \*)mmap(NULL, sb.st\_size, PROT\_READ |
33. PROT\_WRITE, MAP\_SHARED, fd, 0)) == (**void** \*)-1) {
34. perror("mmap");
35. }
37. /\* 映射完后, 关闭文件也可以操纵内存 \*/
38. close(fd);
40. printf("%s", mapped);
42. /\* 修改一个字符,同步到磁盘文件 \*/
43. mapped[20] = '9';
44. **if** ((msync((**void** \*)mapped, sb.st\_size, MS\_SYNC)) == -1) {
45. perror("msync");
46. }
48. /\* 释放存储映射区 \*/
49. **if** ((munmap((**void** \*)mapped, sb.st\_size)) == -1) {
50. perror("munmap");
51. }
53. **return** 0;
54. }

2 私有映射无法修改文件

/\* 将文件映射至进程的地址空间 \*/

if ((mapped = (char \*)mmap(NULL, sb.st\_size, PROT\_READ |

                    PROT\_WRITE, MAP\_PRIVATE, fd, 0)) == (void \*)-1) {

    perror("mmap");

}

**五. 使用共享映射实现两个进程之间的通信**

两个程序映射同一个文件到自己的地址空间, 进程A先运行, 每隔两秒读取映射区域, 看是否发生变化.

进程B后运行, 它修改映射区域, 然后推出, 此时进程A能够观察到存储映射区的变化

进程A的代码:

**C代码  [收藏代码](javascript:void())**

1. #include <sys/mman.h>
2. #include <sys/stat.h>
3. #include <fcntl.h>
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <unistd.h>
7. #include <error.h>
9. #define BUF\_SIZE 100
11. **int** main(**int** argc, **char** \*\*argv)
12. {
13. **int** fd, nread, i;
14. **struct** stat sb;
15. **char** \*mapped, buf[BUF\_SIZE];
17. **for** (i = 0; i < BUF\_SIZE; i++) {
18. buf[i] = '#';
19. }
21. /\* 打开文件 \*/
22. **if** ((fd = open(argv[1], O\_RDWR)) < 0) {
23. perror("open");
24. }
26. /\* 获取文件的属性 \*/
27. **if** ((fstat(fd, &sb)) == -1) {
28. perror("fstat");
29. }
31. /\* 将文件映射至进程的地址空间 \*/
32. **if** ((mapped = (**char** \*)mmap(NULL, sb.st\_size, PROT\_READ |
33. PROT\_WRITE, MAP\_SHARED, fd, 0)) == (**void** \*)-1) {
34. perror("mmap");
35. }
37. /\* 文件已在内存, 关闭文件也可以操纵内存 \*/
38. close(fd);
40. /\* 每隔两秒查看存储映射区是否被修改 \*/
41. **while** (1) {
42. printf("%s\n", mapped);
43. sleep(2);
44. }
46. **return** 0;
47. }

进程B的代码:

**C代码  [收藏代码](javascript:void())**

1. #include <sys/mman.h>
2. #include <sys/stat.h>
3. #include <fcntl.h>
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <unistd.h>
7. #include <error.h>
9. #define BUF\_SIZE 100
11. **int** main(**int** argc, **char** \*\*argv)
12. {
13. **int** fd, nread, i;
14. **struct** stat sb;
15. **char** \*mapped, buf[BUF\_SIZE];
17. **for** (i = 0; i < BUF\_SIZE; i++) {
18. buf[i] = '#';
19. }
21. /\* 打开文件 \*/
22. **if** ((fd = open(argv[1], O\_RDWR)) < 0) {
23. perror("open");
24. }
26. /\* 获取文件的属性 \*/
27. **if** ((fstat(fd, &sb)) == -1) {
28. perror("fstat");
29. }
31. /\* 私有文件映射将无法修改文件 \*/
32. **if** ((mapped = (**char** \*)mmap(NULL, sb.st\_size, PROT\_READ |
33. PROT\_WRITE, MAP\_PRIVATE, fd, 0)) == (**void** \*)-1) {
34. perror("mmap");
35. }
37. /\* 映射完后, 关闭文件也可以操纵内存 \*/
38. close(fd);
40. /\* 修改一个字符 \*/
41. mapped[20] = '9';
43. **return** 0;
44. }

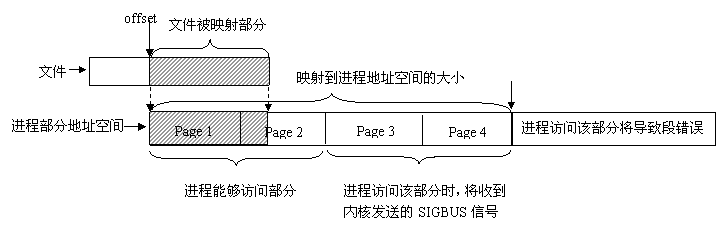
**六. 通过匿名映射实现父子进程通信**

**C代码**

1. #include <sys/mman.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <unistd.h>
6. #define BUF\_SIZE 100
8. **int** main(**int** argc, **char**\*\* argv)
9. {
10. **char**    \*p\_map;
12. /\* 匿名映射,创建一块内存供父子进程通信 \*/
13. p\_map = (**char** \*)mmap(NULL, BUF\_SIZE, PROT\_READ | PROT\_WRITE,
14. MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);
16. **if**(fork() == 0) {
17. sleep(1);
18. printf("child got a message: %s\n", p\_map);
19. sprintf(p\_map, "%s", "hi, dad, this is son");
20. munmap(p\_map, BUF\_SIZE); //实际上，进程终止时，会自动解除映射。
21. exit(0);
22. }
24. sprintf(p\_map, "%s", "hi, this is father");
25. sleep(2);
26. printf("parent got a message: %s\n", p\_map);
28. **return** 0;
29. }

**七. 对mmap()返回地址的访问**

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：



总结一下就是, 文件大小, mmap的参数 len 都不能决定进程能访问的大小, 而是容纳文件被映射部分的最小页面数决定

进程能访问的大小. 下面看一个实例:

**C代码**

1. #include <sys/mman.h>
2. #include <sys/types.h>
3. #include <sys/stat.h>
4. #include <fcntl.h>
5. #include <unistd.h>
6. #include <stdio.h>
8. **int** main(**int** argc, **char**\*\* argv)
9. {
10. **int** fd,i;
11. **int** pagesize,offset;
12. **char** \*p\_map;
13. **struct** stat sb;
15. /\* 取得page size \*/
16. pagesize = sysconf(\_SC\_PAGESIZE);
17. printf("pagesize is %d\n",pagesize);
19. /\* 打开文件 \*/
20. fd = open(argv[1], O\_RDWR, 00777);
21. fstat(fd, &sb);
22. printf("file size is %zd\n", (**size\_t**)sb.st\_size);
24. offset = 0;
25. p\_map = (**char** \*)mmap(NULL, pagesize \* 2, PROT\_READ|PROT\_WRITE,
26. MAP\_SHARED, fd, offset);
27. close(fd);
29. p\_map[sb.st\_size] = '9';  /\* 导致总线错误 \*/
30. p\_map[pagesize] = '9';    /\* 导致段错误 \*/
32. munmap(p\_map, pagesize \* 2);
34. **return** 0;
35. }