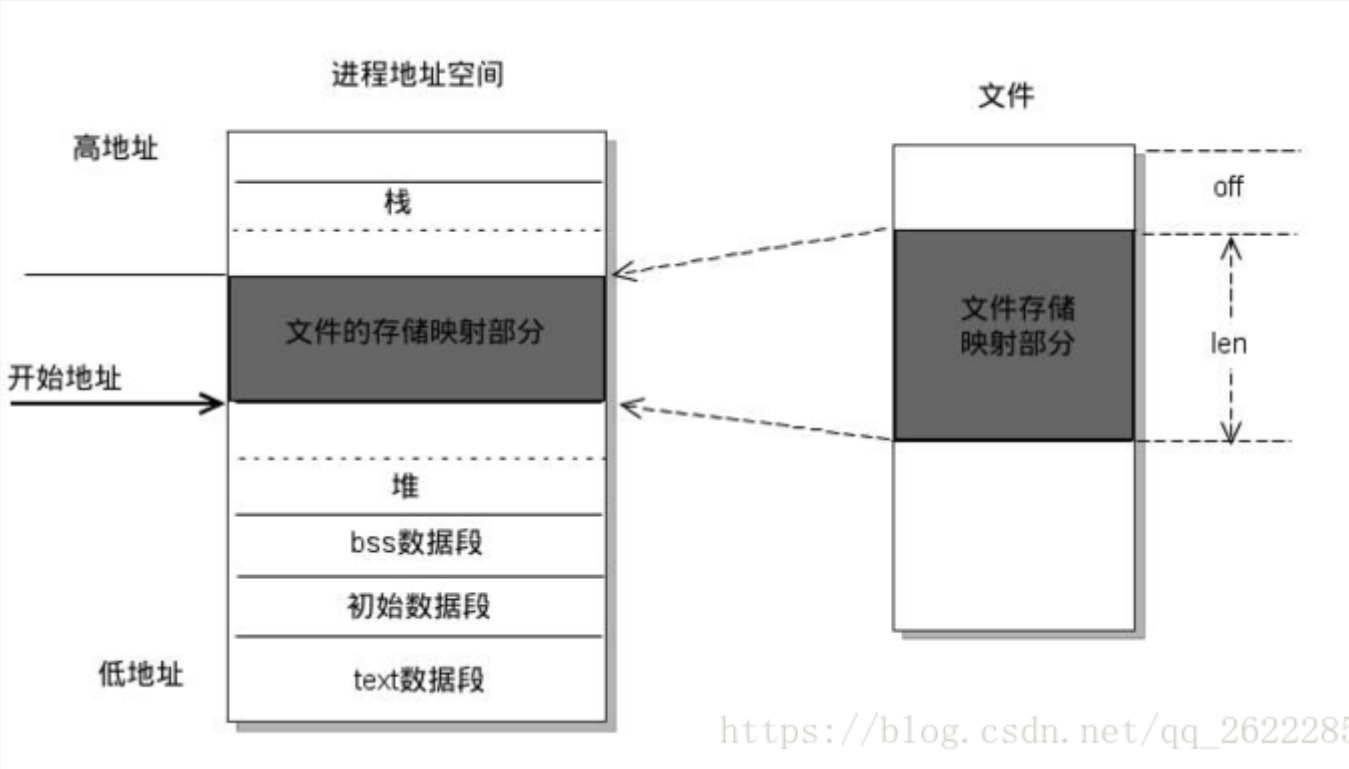
Linux mmap

mmap 基础概念

1.mmap 内存映射原理

mmap 是一种内存映射文件的方法，即将一个文件或者其他对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对应关系；实现这样的映射关系后，进程就可以采用指针的方式读写操作这一块内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必调用 read，write 等系统调用函数，相反，内核空间堆这段区域的修改也直接反应到用户空间，从而可以实现不同进程间的文件共享。

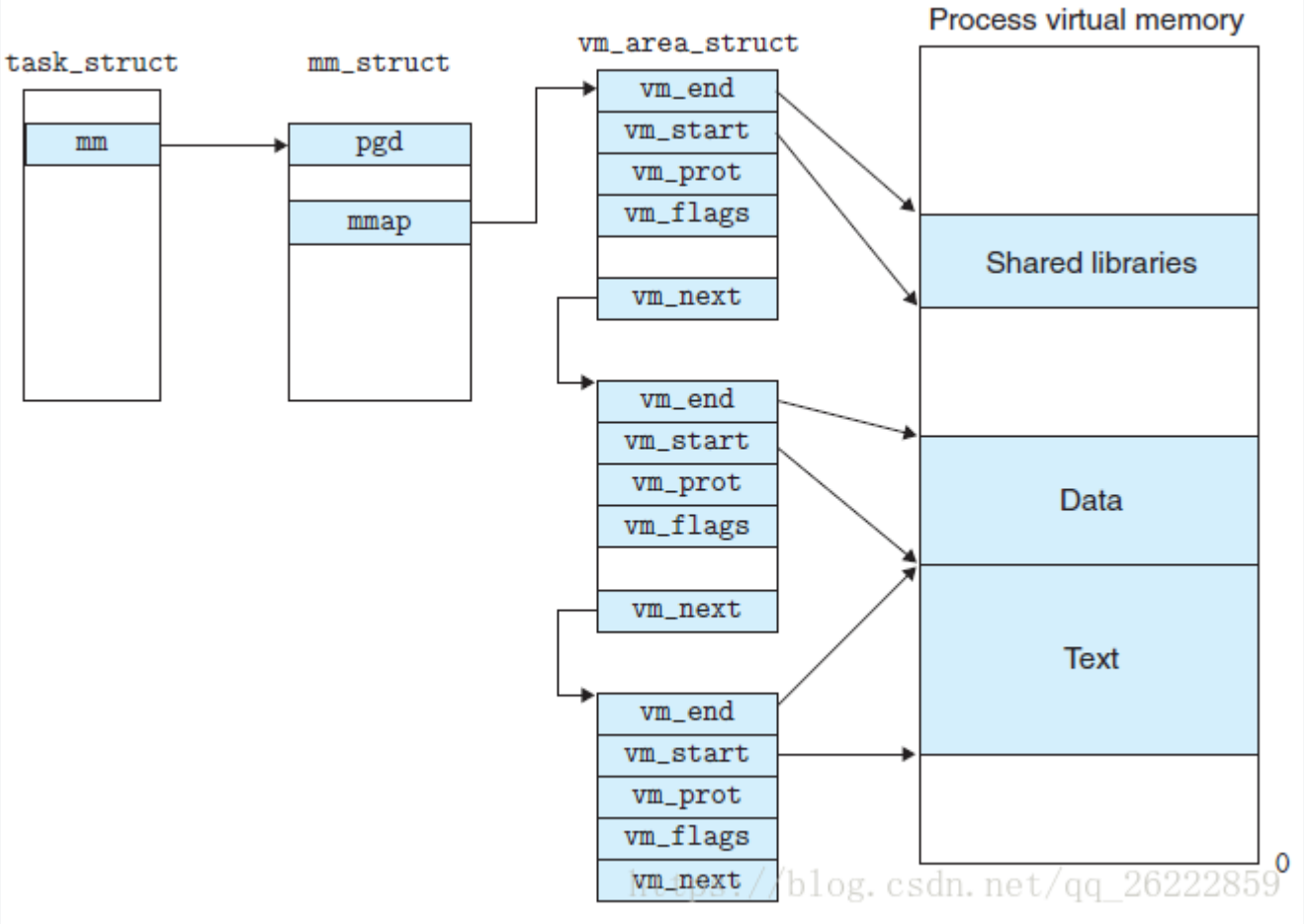


- 进程的虚拟地址空间，由多个虚拟内存区域构成。
- 虚拟内存区域是进程的虚拟地址空间中的一个同质区间，即具有同样特性的连续地址范围。
- text 数据段，初始数据段，BSS 数据段，堆，栈和内存映射都是一个独立的虚拟内存区域。
- 为内存映射服务的地址空间处于堆栈之外的空余部分。

1.2 vm_area_struct

vm_area_struct ：虚拟内存管理的最基本单元，描述的是一段连续的，具有相同访问属性的虚拟空间，该空间的大小为物理内存页面的整数倍。

linux 内核实用 vm_area_struct 来表示一个独立的虚拟内存区域，由于每个不同质的虚拟内存区域功能和内部机制不同，因此一个进程实用多个 vm_area_struct 结构来分别表示不同类型的虚拟内存区域。各个 vm_area_struct 实用链表或者树形结构连接，方便进程快速访问。



task_struct:进程控制模块；

mm_struct：进程所拥有的内存描述符；

2.mmap 内存映射原理

mmap 内存映射的实现过程，可分为三个阶段

2.1 进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域

- 进程在用户空间调用 mmap 函数
- 在当前进程的虚拟地址空间中，寻找一段空闲的满足要求的连续的虚拟地址
- 为此虚拟去分配一个 vm_area_struct 结构，并对该结构各个域进行初始化
- 将新建的 vm_area_struct 插入到进程的虚拟地址区域链表或树中

2.2 调用内核空间的系统调用函数 mmap（不同于用户空间），实现文件的物理地址和进程的虚拟地址的一一映射关系。

- 为映射分配了新的虚拟地址区域后，通过待映射的文件指针，在文件描述符表中找到对应的文件描述符，通过文件描述符，连接到内核“已打开文集”中该文件的文件结构体 struct file，每个文件结构体维护着和这个已打开文件的相关信息。
- 通过该文件的文件结构体，连接到 file_operations 模块，调用 内核函数 mmap，int mmap(struct file *filep,struct vm_area_struct *vma)
- 内核 mmap 函数通过虚拟文件系统 inode 模块定位到文件磁盘物理地址。
- 通过 remap_pfn_range 函数建立页表，即实现了文件地址和虚拟地址区域的映射，此时这片虚拟地址区域没有任何数据关联到主存中

2.3 进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存的拷贝。

- 进程的读写操作访问虚拟地址空间的这一段映射地址，通过查询页表，发现这一段地址不在物理页面上，因为只是建立了地址映射，真正的磁盘数据还没有拷贝到内存中，因此引发缺页异常
- 缺页异常进行一系列判断，确定无非法操作后，内核发起请求调页过程
- 调页过程先在交换缓存空间中寻找需要访问的内存页，如果没有则调用 nopage 函数把所缺的页面从磁盘装入主存中
- 之后进程可对这片主存进行读或写操作，如果写操作改变了内容，一定时间后系统会自动回写脏页面到对应的磁盘地址，也就是完成了写入到文件的过程
- 修改过的脏页面不会立即更新到文件中，而是有一段时间的延迟，可以调用 msync 来强制同步，这样所写的内容就立即保存到文件里了。

3.mmap 和常规文件操作的区别

3.1 常规的文件读写操作的过程

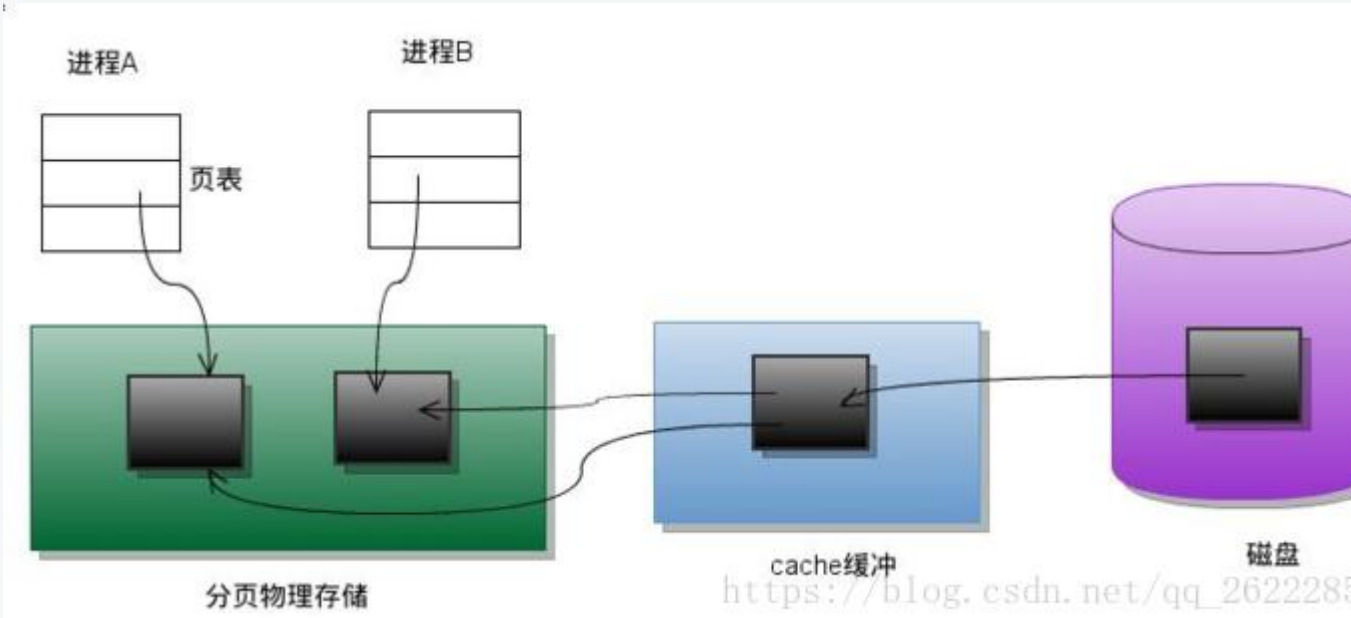
- 进程发起读文件请求
- 内核通过查找进程文件符表，定位内核已经打开文件集上的文件信息，从而找到文件的 inode
- inode 在 address_space 上查找要请求的文件页是否已经缓存在页缓存中，如果存在，则直接返回这片文件页的内容
- 如果不存在，就通过 inode 定位到文件磁盘地址，将数据从磁盘复制到页缓存，之后再发起读页面的过程，进而将也缓存中的数据发给用户进程

总的来说，**常规文件操作为了提高写效率和保护磁盘，使用了页缓存机制，这样造成读文件时需要先将文件页从磁盘拷贝到页缓存，由于页缓存处于内核空间，不能被用户进程直接寻址，所以还需要将也缓存中数据页再次拷贝到内存对应的用户空间。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取。**

写操作也一样，待写入的 **buffer** 在内核空间不能直接访问，必须先拷贝到内核空间对应的主存，再回写到磁盘中，也是需要两次拷贝。

如果多个进程访问同一个文件，则每个进程都在自己的地址空间都含有该文件的副本。

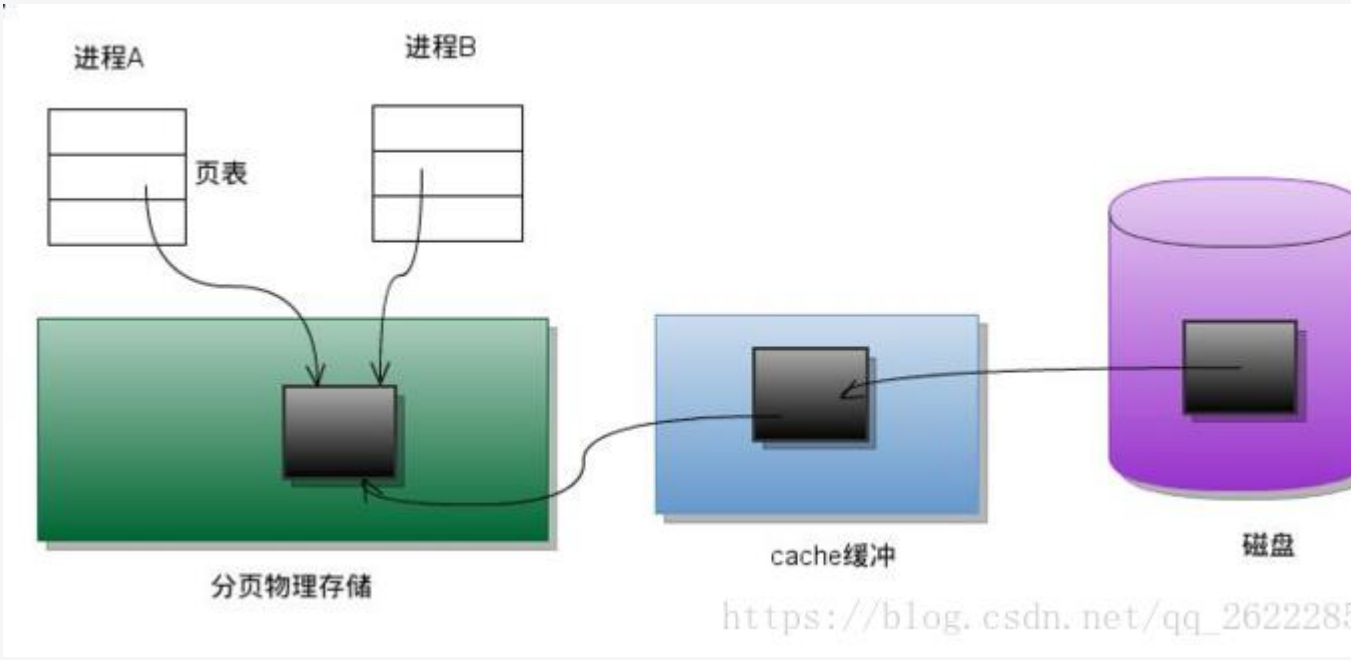
也就是系统将该页从磁盘读到页缓存，每个进程再执行一个存储器内的复制操作将数据从页缓存拷贝的自己的地址空间中



3.2 共享内存的方式

进程 a 和进程 b 豆浆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，当进程 B 访问

同一页而出现缺页中断时，该页已经存在在内存中，内核只需要将进程 B 的页表指向即可。



3.2 使用 mmap 操作文件

> 1.创建新的虚拟内存区域
> 2.建立文件磁盘地址和虚拟内存区域映射

上面的两个过程并没有任何文件拷贝操作，而之后访问数据时发现内存中并没有数据而引发的缺页异常过程，可以通过已经建立好的映射关系只进行一次数据拷贝，就将磁盘中的数据拷贝到用户空间中，供进程使用。

3.3 总结

常规文件操作从磁盘到页缓存再到用户主存，要两次数据拷贝；
mmap 操作文件，只需要从磁盘到用户主存的一次数据拷贝过程。
mmap 的关键点是：实现了用户空间和内核空间数据的直接交换。

4.mmap 优点总结

> 1.对文件的读取操作跨过了页缓存，减少了数据的拷贝次数。
> 2.实现了用户空间和内核空间的高效交互方式，
> 3.提供进程间共享内存以及相互通信的方式。两个进程将自身的用户空间映射到同一片区域，从而达到通信或者共享的目的。（进程 A 和进程 B 都映射了区

> 域 C，当 A 第一次读取 C 时通过缺页异常从磁盘复制文件页到内存中，但当 B 再去读取 C 的时候，虽然会产生缺页异常，但是不需要再从磁盘复制文件，而是可以直接使用保存在内存中的文件数据）

4.可用于实现高效的大规模数据传输。也即是说需要磁盘代替内存空间的时候，可以使用 mmap

5.mmap 相关函数

函数原型

```
1. void *mmap(void *start, size_t length, int prot, int flags, int fd,
   off_t offset);
2.
```

返回说明

成功执行时：mmap 返回被映射区域的指针

失败时：返回 MAP_FAILED

参数

start：映射区开始地址

length：映射区的长度

prot：期望的内存保护标志，不能与文件打开模式冲突

1. 1 PROT_EXEC ：页内容可以被执行
2. 2 PROT_READ ：页内容可以被读取
3. 3 PROT_WRITE ：页可以被写入
4. 4 PROT_NONE ：页不可访问

fd:有效的文件描述符

offset：被映射内容的起点。

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

1. 1 MAP_FIXED //使用指定的映射起始地址，如果由 start 和 len 参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。
2. 2 MAP_SHARED //与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。直到 msync()或者 munmap()被调用，文件实际上不会被更新。
3. 3 MAP_PRIVATE //建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。
4. 4 MAP_DENYWRITE //这个标志被忽略。

5. 5 MAP_EXECUTABLE //同上
6. 6 MAP_NORESERVE //不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。
7. 7 MAP_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。
8. 8 MAP_GROWSDOWN //用于堆栈，告诉内核 VM 系统，映射区可以向下扩展。
9. 9 MAP_ANONYMOUS //匿名映射，映射区不与任何文件关联。
10. 10 MAP_ANON //MAP_ANONYMOUS 的别称，不再被使用。
11. 11 MAP_FILE //兼容标志，被忽略。
12. 12 MAP_32BIT //将映射区放在进程地址空间的低 2GB，MAP_FIXED 指定时会被忽略。当前这个标志只在 x86-64 平台上得到支持。
13. 13 MAP_POPULATE //为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。
14. 14 MAP_NONBLOCK //仅和 MAP_POPULATE 一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。
15.

6.实例

6.1 通过 mmap 修改文件

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <error.h>

#define BUF_SIZE 100

int main(int argc, char **argv)
{
    int fd, nread, i;
    struct stat sb;
    char *mapped, buf[BUF_SIZE];

    for (i = 0; i < BUF_SIZE; i++) {
        buf[i] = '#';
    }

    /* 打开文件 */
```

```
    if ((fd = open(argv[1], O_RDWR)) < 0) {
        perror("open");
    }

    /* 获取文件的属性 */
    if ((fstat(fd, &sb)) == -1) {
        perror("fstat");
    }

    /* 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_SHARED, fd, 0)) == (void *)-1) {
        perror("mmap");
    }

    /* 映射完后, 关闭文件也可以操纵内存 */
    close(fd);

    printf("%s", mapped);

    /* 修改一个字符,同步到磁盘文件 */
    mapped[20] = '9';
    if ((msync((void *)mapped, sb.st_size, MS_SYNC)) == -1) {
        perror("msync");
    }

    /* 释放存储映射区 */
    if ((munmap((void *)mapped, sb.st_size)) == -1) {
        perror("munmap");
    }

    return 0;
}

```

6.2mmap 实现进程间通信

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
```

```
6. #include <unistd.h>
7. #include <error.h>
8.
9. #define BUF_SIZE 100
10.
11.int main(int argc, char **argv)
12.{
13.    int fd, nread, i;
14.    struct stat sb;
15.    char *mapped, buf[BUF_SIZE];
16.
17.    for (i = 0; i < BUF_SIZE; i++) {
18.        buf[i] = '#';
19.    }
20.
21.    /* 打开文件 */
22.    if ((fd = open(argv[1], O_RDWR)) < 0) {
23.        perror("open");
24.    }
25.
26.    /* 获取文件的属性 */
27.    if ((fstat(fd, &sb)) == -1) {
28.        perror("fstat");
29.    }
30.
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.    }
36.
37.    /* 文件已在内存，关闭文件也可以操纵内存 */
38.    close(fd);
39.
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.    }
45.
46.    return 0;
47.}
```

48.

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <error.h>

#define BUF_SIZE 100

int main(int argc, char **argv)
{
    int fd, nread, i;
    struct stat sb;
    char *mapped, buf[BUF_SIZE];

    for (i = 0; i < BUF_SIZE; i++) {
        buf[i] = '#';
    }

    /* 打开文件 */
    if ((fd = open(argv[1], O_RDWR)) < 0) {
        perror("open");
    }

    /* 获取文件的属性 */
    if ((fstat(fd, &sb)) == -1) {
        perror("fstat");
    }

    /* 私有文件映射将无法修改文件 */
    if ((mapped = (char *)mmap(NULL, sb.st_size, PROT_READ |
                    PROT_WRITE, MAP_PRIVATE, fd, 0)) == (void *)-1) {
        perror("mmap");
    }

    /* 映射完后, 关闭文件也可以操纵内存 */
    close(fd);

    /* 修改一个字符 */
    mapped[20] = '9';
```

```
42.
43.    return 0;
44.}
45.
```

6.3 匿名映射实现父子进程通信

```
1. #include <sys/mman.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <unistd.h>
5.
6. #define BUF_SIZE 100
7.
8. int main(int argc, char** argv)
9. {
10.    char    *p_map;
11.
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);
15.
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.    }
23.
24.    sprintf(p_map, "%s", "hi, this is father");
25.    sleep(2);
26.    printf("parent got a message: %s\n", p_map);
27.
28.    return 0;
29.}
30.
31.
```