Linux内存管理之mmap详解

一. mmap系统调用

1. mmap系统调用

    mmap将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。munmap执行相反的操作，删除特定地址区域的对象映射。

当使用mmap映射文件到进程后,就可以直接操作这段虚拟地址进行文件的读写等操作,不必再调用read,write等系统调用.但需注意,直接对该段内存写时不会写入超过当前文件大小的内容.

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：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。

    基 于文件的映射，在mmap和munmap执行过程的任何时刻，被映射文件的st\_atime可能被更新。如果st\_atime字段在前述的情况下没有得到 更新，首次对映射区的第一个页索引时会更新该字段的值。用PROT\_WRITE 和 MAP\_SHARED标志建立起来的文件映射，其 st\_ctime 和 st\_mtime在对映射区写入之后，但在msync()通过MS\_SYNC 和 MS\_ASYNC两个标志调用之前会被更新。

用法：

#include

void \*mmap(void \*start, size\_t length, int prot, int flags,

int fd, off\_t offset);

int munmap(void \*start, size\_t length);

返回说明：

成功执行时，mmap()返回被映射区的指针，munmap()返回0。失败时，mmap()返回MAP\_FAILED[其值为(void \*)-1]，munmap返回-1。errno被设为以下的某个值

EACCES：访问出错

EAGAIN：文件已被锁定，或者太多的内存已被锁定

EBADF：fd不是有效的文件描述词

EINVAL：一个或者多个参数无效

ENFILE：已达到系统对打开文件的限制

ENODEV：指定文件所在的文件系统不支持内存映射

ENOMEM：内存不足，或者进程已超出最大内存映射数量

EPERM：权能不足，操作不允许

ETXTBSY：已写的方式打开文件，同时指定MAP\_DENYWRITE标志

SIGSEGV：试着向只读区写入

SIGBUS：试着访问不属于进程的内存区

参数：

start：映射区的开始地址。

length：映射区的长度。

prot：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过or运算合理地组合在一起

PROT\_EXEC //页内容可以被执行

PROT\_READ //页内容可以被读取

PROT\_WRITE //页可以被写入

PROT\_NONE //页不可访问

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

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。

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。

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。

MAP\_DENYWRITE //这个标志被忽略。

MAP\_EXECUTABLE //同上

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。

MAP\_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。

MAP\_GROWSDOWN //用于堆栈，告诉内核VM系统，映射区可以向下扩展。

MAP\_ANONYMOUS //匿名映射，映射区不与任何文件关联。

MAP\_ANON //MAP\_ANONYMOUS的别称，不再被使用。

MAP\_FILE //兼容标志，被忽略。

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。

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。

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。

fd：有效的文件描述词。如果MAP\_ANONYMOUS被设定，为了兼容问题，其值应为-1。

offset：被映射对象内容的起点。

2. 系统调用munmap()

#include

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。

3. 系统调用msync()

#include

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。

二. 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： 

1. fd=open(name, flag, mode);
2. if(fd<0)
3. ...
4. 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

（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.

三. mmap进行内存映射的原理

     mmap系统调用的最终目的是将,设备或文件映射到用户进程的虚拟地址空间,实现用户进程对文件的直接读写,这个任务可以分为以下三步:

1.在用户虚拟地址空间中寻找空闲的满足要求的一段连续的虚拟地址空间,为映射做准备(由内核mmap系统调用完成)

       每个进程拥有3G字节的用户虚存空间。但是，这并不意味着用户进程在这3G的范围内可以任意使用，因为虚存空间最终得映射到某个物理存储空间（内存或磁盘空间），才真正可以使用。

       那 么，内核怎样管理每个进程3G的虚存空间呢？概括地说，用户进程经过编译、链接后形成的映象文件有一个代码段和数据段（包括data段和bss段），其中 代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，即全局变量和所有申明为static的局部变量，这些空间是进程所必需的基本要求，这些 空间是在建立一个进程的运行映像时就分配好的。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的，如图3.1所示：

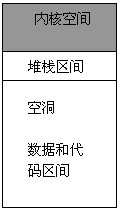
[](http://blog.chinaunix.net/attachment/201202/19/26669729_1329661302VzvN.png)

 图3.1  进程虚拟空间的划分

      在内核中,这样每个区域用一个结构struct vm\_area\_struct 来表示.它描述的是一段连续的、具有相同访问属性的虚存空间，该虚存空间的大小为物理内存页面的整数倍。可以使用 cat /proc//maps来查看一个进程的内存使用情况,pid是进程号.其中显示的每一行对应进程的一个vm\_area\_struct结构.

下面是struct vm\_area\_struct结构体的定义：

1. #include <linux/mm\_types.h>
2. /\* This struct defines a memory VMM memory area. \*/
3. struct vm\_area\_struct {
4. struct mm\_struct \* vm\_mm; /\* VM area parameters \*/
5. unsigned long vm\_start;
6. unsigned long vm\_end;
7. /\* linked list of VM areas per task, sorted by address \*/
8. struct vm\_area\_struct \*vm\_next;
9. pgprot\_t vm\_page\_prot;
10. unsigned long vm\_flags;
11. /\* AVL tree of VM areas per task, sorted by address \*/
12. short vm\_avl\_height;
13. struct vm\_area\_struct \* vm\_avl\_left;
14. struct vm\_area\_struct \* vm\_avl\_right;
15. /\* For areas with an address space and backing store,
16. vm\_area\_struct \*vm\_next\_share;
17. struct vm\_area\_struct \*\*vm\_pprev\_share;
18. struct vm\_operations\_struct \* vm\_ops;
19. unsigned long vm\_pgoff; /\* offset in PAGE\_SIZE units, \*not\* PAGE\_CACHE\_SIZE \*/
20. struct file \* vm\_file;
21. unsigned long vm\_raend;
22. void \* vm\_private\_data; /\* was vm\_pte (shared mem) \*/
23. };

      通常，进程所使用到的虚存空间不连续，且各部分虚存空间的访问属性也可能不同。所以一个进程的虚存空间需要多个vm\_area\_struct结构来描述。 在vm\_area\_struct结构的数目较少的时候，各个vm\_area\_struct按照升序排序，以单链表的形式组织数据（通过vm\_next指针 指向下一个vm\_area\_struct结构）。但是当vm\_area\_struct结构的数据较多的时候，仍然采用链表组织的化，势必会影响到它的搜索 速度。针对这个问题，vm\_area\_struct还添加了vm\_avl\_hight（树高）、vm\_avl\_left（左子节点）、 vm\_avl\_right（右子节点）三个成员来实现AVL树，以提高vm\_area\_struct的搜索速度。

　　假如该vm\_area\_struct描述的是一个文件映射的虚存空间，成员vm\_file便指向被映射的文件的file结构，vm\_pgoff是该虚存空间起始地址在vm\_file文件里面的文件偏移，单位为物理页面。

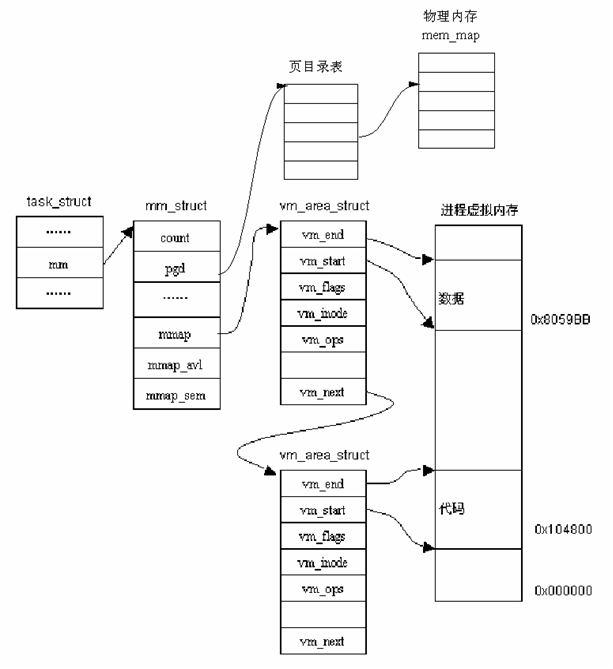
[](http://blog.chinaunix.net/attachment/201202/19/26669729_1329661050h7BX.png)

图3.2  进程虚拟地址示意图

因此,mmap系统调用所完成的工作就是准备这样一段虚存空间,并建立vm\_area\_struct结构体,将其传给具体的设备驱动程序.

2. 建立虚拟地址空间和文件或设备的物理地址之间的映射(设备驱动完成)

  建立文件映射的第二步就是建立虚拟地址和具体的物理地址之间的映射,这是通过修改进程页表来实现的.mmap方法是file\_opeartions结构的成员:

  int (\*mmap)(struct file \*,struct vm\_area\_struct \*);

linux有2个方法建立页表:

(1) 使用remap\_pfn\_range一次建立所有页表.

   int remap\_pfn\_range(struct vm\_area\_struct \*vma, unsigned long virt\_addr, unsigned long pfn, unsigned long size, pgprot\_t prot);

返回值:

成功返回 0, 失败返回一个负的错误值  
参数说明:

vma 用户进程创建一个vma区域

virt\_addr 重新映射应当开始的用户虚拟地址. 这个函数建立页表为这个虚拟地址范围从 virt\_addr 到 virt\_addr\_size.

pfn 页帧号, 对应虚拟地址应当被映射的物理地址. 这个页帧号简单地是物理地址右移 PAGE\_SHIFT 位. 对大部分使用, VMA 结构的 vm\_paoff 成员正好包含你需要的值. 这个函数影响物理地址从 (pfn<<page\_shift) 到 (pfn<<page\_shift)+size.< span=""></page\_shift) 到 (pfn<<page\_shift)+size.<>

size 正在被重新映射的区的大小, 以字节.

prot 给新 VMA 要求的"protection". 驱动可(并且应当)使用在vma->vm\_page\_prot 中找到的值.

(2) 使用nopage VMA方法每次建立一个页表项.

   struct page \*(\*nopage)(struct vm\_area\_struct \*vma, unsigned long address, int \*type);

返回值:

成功则返回一个有效映射页,失败返回NULL.

参数说明:

address 代表从用户空间传过来的用户空间虚拟地址.

返回一个有效映射页.

(3) 使用方面的限制：

remap\_pfn\_range 不能映射常规内存，只存取保留页和在物理内存顶之上的物理地址。因为保留页和在物理内存顶之上的物理地址内存管理系统的各个子模块管理不到。 640 KB 和 1MB 是保留页可能映射，设备I/O内存也可以映射。如果想把kmalloc()申请的内存映射到用户空间，则可以通过 mem\_map\_reserve()把相应的内存设置为保留后就可以。

3. 当实际访问新映射的页面时的操作(由缺页中断完成)

(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。