**mmap是什么**

mmap是一种内存映射文件的方法，它将一个文件或者其它对象映射到进程的地址空间，从而实现了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映关系。实现了按需加载分页，不直接从磁盘读取文件内容，并且初始化时根本不使用物理 RAM。在访问特定位置后，以 lazy 方式从磁盘读取数据。

**mmap好处**

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段内存，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再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。

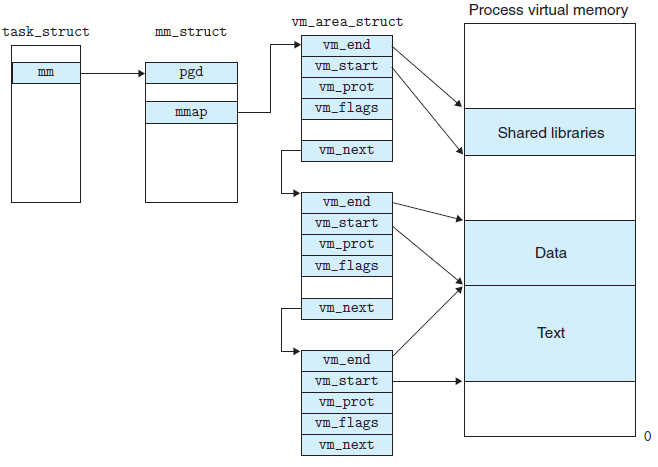
**mmap数据结构**

如下图所示：



由上图可以看出，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。上图中所示的text数据段（代码段）、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。而为内存映射服务的地址空间处在堆栈之间的空余部分。

linux内核使用vm\_area\_struct结构来表示一个独立的虚拟内存区域，由于每个不同的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个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结构使用链表或者树形结构链接，方便进程快速访问，如下图所示：



vm\_area\_struct结构中包含区域起始和终止地址以及其他相关信息，同时也包含一个vm\_ops指针，其内部可引出所有针对这个区域可以使用的系统调用函数。这样，进程对某一虚拟内存区域的任何操作需要用要的信息，都可以从vm\_area\_struct中获得。mmap函数就是要创建一个新的vm\_area\_struct结构，并将其与文件的物理磁盘地址相连。

# mmap内存映射原理

mmap内存映射的实现过程，总的来说可以分为三个阶段：

**（一）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**

1、进程在用户空间调用库函数mmap，原型：void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);

2、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

3、为此虚拟区分配一个vm\_area\_struct结构，接着对这个结构的各个域进行了初始化

4、将新建的虚拟区结构（vm\_area\_struct）插入进程的虚拟地址区域链表或树中

**（二）调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系**

5、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，链接到内核“已打开文件集”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相关各项信息。

6、通过该文件的文件结构体，链接到file\_operations模块，调用内核函数mmap，其原型为：int mmap(struct file \*filp, struct vm\_area\_struct \*vma)，不同于用户空间库函数。

7、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。

8、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关系。此时，这片虚拟地址并没有任何数据关联到主存中。

**（三）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（主存）的拷贝**

注：前两个阶段仅在于创建虚拟区间并完成地址映射，但是并没有将任何文件数据的拷贝至主存。真正的文件读取是当进程发起读或写操作时。

9、进程的读或写操作访问虚拟地址空间这一段映射地址，通过查询页表，发现这一段地址并不在物理页面上。因为目前只建立了地址映射，真正的硬盘数据还没有拷贝到内存中，因此引发缺页异常。

10、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。

11、调页过程先在交换缓存空间（swap cache）中寻找需要访问的内存页，如果没有则调用nopage函数把所缺的页从磁盘装入到主存中。

12、之后进程即可对这片主存进行读或者写的操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。

注：修改过的脏页面并不会立即更新回文件中，而是有一段时间的延迟，可以调用msync()来强制同步, 这样所写的内容就能立即保存到文件里了。

# mmap和常规文件操作的区别

我们首先简单的回顾一下常规文件系统操作（调用read/fread等类函数）中，函数的调用过程：

1、进程发起读文件请求。

2、内核通过查找进程文件符表，定位到内核已打开文件集上的文件信息，从而找到此文件的inode。

3、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。

4、如果不存在，则通过inode定位到文件磁盘地址，将数据从磁盘复制到页缓存。之后再次发起读页面过程，进而将页缓存中的数据发给用户进程。

总结来说，常规文件操作为了提高读写效率和保护磁盘，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中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。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任务。写操作也是一样，待写入的buffer在内核空间不能直接访问，必须要先拷贝至内核空间对应的主存，再写回磁盘中（延迟写回），也是需要两次数据拷贝。

而使用mmap操作文件中，创建新的虚拟内存区域和建立文件磁盘地址和虚拟内存区域映射这两步，没有任何文件拷贝操作。而之后访问数据时发现内存中并无数据而发起的缺页异常过程，可以通过已经建立好的映射关系，只使用一次数据拷贝，就从磁盘中将数据传入内存的用户空间中，供进程使用。

**总而言之，常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝。而mmap操控文件，只需要从磁盘到用户主存的一次数据拷贝过程。**说白了，mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。

# mmap优点总结

由上文讨论可知，mmap优点共有一下几点：

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，用内存读写取代I/O读写，提高了文件读取效率。

2、实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉。

3、提供进程间共享内存及相互通信的方式。不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件或匿名映射到同一片区域。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的。

     同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经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。内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过mmap映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，mmap都可以发挥其功效。

# mmap相关函数

**函数原型**

void \*mmap(void \*start, size\_t length, int prot, int flags,int fd, off\_t offset);

**返回说明**

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

1 EACCES：访问出错

2 EAGAIN：文件已被锁定，或者太多的内存已被锁定

3 EBADF：fd不是有效的文件描述词

4 EINVAL：一个或者多个参数无效

5 ENFILE：已达到系统对打开文件的限制

6 ENODEV：指定文件所在的文件系统不支持内存映射

7 ENOMEM：内存不足，或者进程已超出最大内存映射数量

8 EPERM：权能不足，操作不允许

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

10 SIGSEGV：试着向只读区写入

11 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

1 PROT\_EXEC ：页内容可以被执行

2 PROT\_READ ：页内容可以被读取

3 PROT\_WRITE ：页可以被写入

4 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

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 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 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 MAP\_DENYWRITE //这个标志被忽略。

5 MAP\_EXECUTABLE //同上

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 MAP\_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。

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

9 MAP\_ANONYMOUS //匿名映射，映射区不与任何文件关联。

10 MAP\_ANON //MAP\_ANONYMOUS的别称，不再被使用。

11 MAP\_FILE //兼容标志，被忽略。

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 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 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

**相关函数**

int munmap( void \* addr, size\_t len )

成功执行时，munmap()返回0。失败时，munmap返回-1，error返回标志和mmap一致；

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。

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。

# mmap使用细节

1、使用mmap需要注意的一个关键点是，mmap映射区域大小必须是物理页大小(page\_size)的整倍数（32位系统中通常是4k字节）。原因是，内存的最小粒度是页，而进程虚拟地址空间和内存的映射也是以页为单位。为了匹配内存的操作，mmap从磁盘到虚拟地址空间的映射也必须是页。

2、内核可以跟踪被内存映射的底层对象（文件）的大小，进程可以合法的访问在当前文件大小以内又在内存映射区以内的那些字节。也就是说，如果文件的大小一直在扩张，只要在映射区域范围内的数据，进程都可以合法得到，这和映射建立时文件的大小无关。具体情形参见“情形三”。

3、映射建立之后，即使文件关闭，映射依然存在。因为映射的是磁盘的地址，不是文件本身，和文件句柄无关。同时可用于进程间通信的有效地址空间不完全受限于被映射文件的大小，因为是按页映射。

**mmap 实际应用**

前面介绍了 mmap 的高效率、不丢失的特点，所以非常适合用在写日志中，参考后文的 xlog，微信就是通过 mmap 来解决大量日志造成高 CPU、高 I/O，大量使用内存易丢日志的问题。

另外还可以用来解决图片缓存的问题。

在做启动优化时，有一个很重要的部分就是启动 I/O 优化，在启动 I/O 优化过程中通过 Instrument 可以看到大量的图片 I/O 操作，这时就可以将启动过程中用到的图片 I/O 转换为一个大文件保存，然后启动时通过 mmap 内存映射加载，将多个文件 I/O 合并成一个大的文件 I/O（可以子线程预加载），并节省图片文件到用户空间过程中由于页缓存的消耗，并且通过 mmap 在大文件中截取各个图片对应的位置，并恢复成图片可以节省大量的 I/O。

# 页高速缓存和页回写

从访问速度来看，内存访问速度高于磁盘访问，而缓存访问速度又高于内存。

缓存手段

页高速缓存是由内存中的物理页面组成，对应的是磁盘上的物理块，可以动态扩大缩小。当内核开始读操作时，会先检查数据是否在缓存中，不命中采取磁盘找。

写缓存

Linux在调用写操作时，使用的策略是——回写。这种策略是直接写到缓存中，而存储不是立刻更新，而是将页高速缓存中被写入的页面标记成脏页，表示缓存与磁盘不同步。延迟写磁盘，能方便后续对该页面的操作。

Linux页高速缓存

由于页高速缓存中的页可能包含多个不连续的物理磁盘块。Linux页高速缓存使用一个新的结构来管理缓存项和页面io操作。

struct address\_space {  
 struct inode \*host; /\* owning inode \*/  
 struct radix\_tree\_root page\_tree; /\* radix tree of all pages \*/  
 spinlock\_t tree\_lock; /\* page\_tree lock \*/  
 unsigned int i\_mmap\_writable; /\* VM\_SHARED ma count \*/  
 struct prio\_tree\_root i\_mmap; /\* list of all mappings \*/  
 struct list\_head i\_mmap\_nonlinear; /\* VM\_NONLINEAR ma list \*/  
 spinlock\_t i\_mmap\_lock; /\* i\_mmap lock \*/  
 atomic\_t truncate\_count; /\* truncate re count \*/  
 unsigned long nrpages; /\* total number of pages \*/  
 pgoff\_t writeback\_index; /\* writeback start offset \*/  
 struct address\_space\_operations \*a\_ops; /\* operations table \*/  
 unsigned long flags; /\* gfp\_mask and error flags \*/  
 struct backing\_dev\_info \*backing\_dev\_info; /\* read-ahead information \*/  
 spinlock\_t private\_lock; /\* private lock \*/  
 struct list\_head private\_list; /\* private list \*/  
 struct address\_space \*assoc\_mapping; /\* associated buffers \*/  
};

其中i\_mmap字段是一个优先搜索树，它包含了在该结构体中所有共享的与私有的映射页面。

**address\_space 操作**

a\_ops域指向地址空间对象中的操作函数表，由adress\_space\_operations结构体表示：

struct address\_space\_operations {  
 int (\*writepage)(struct page \*, struct writeback\_control \*);  
 int (\*readpage) (struct file \*, struct page \*);  
 int (\*sync\_page) (struct page \*);  
 int (\*writepages) (struct address\_space \*, struct writeback\_control \*);  
 int (\*set\_page\_dirty) (struct page \*);  
 int (\*readpages) (struct file \*, struct address\_space \*,struct list\_head \*, unsigned);  
 int (\*prepare\_write) (struct file \*, struct page \*, unsigned, unsigned);  
 int (\*commit\_write) (struct file \*, struct page \*, unsigned, unsigned);  
 sector\_t (\*bmap)(struct address\_space \*, sector\_t);  
 int (\*invalidatepage) (struct page \*, unsigned long);  
 int (\*releasepage) (struct page \*, int);  
 int (\*direct\_IO) (int, struct kiocb \*, const struct iovec \*,loff\_t, unsigned long);  
};

这里面最重要的是读写——readpage()和writepage()。

对于readpage，Linux内核试图在页高速缓存中使用**find\_get\_page(mapping, index)**找到需要的数据，将一个adress\_space对象和一个偏移量传给该方法。如果找不到会返回一个NULL，并且内核新分配一个页面，并之前搜索的页面加入到页高速缓存。

对于writepage，首先在cache中搜索需要的页，如果需要的页不在，则新分配一个空闲项；下一步，内核会创建一个写请求，接着数据被从用户空间拷贝到内核缓冲，最后写入磁盘。

mmap本质上从内核中 进程中申请了vm\_area\_struct虚拟内存后，把虚拟内存保存到 file结构体address\_space结构体中的i\_mmap中。当需要访问这段虚拟地址，就会产生却也中断，通过伙伴系统申请出物理内存绑定。如果是文件映射还会预读取文件内容，之后读取可以命中后续的缓存。项目中也有频繁进行写入，可以快速的同步到内存中。