Linux文件基础和mmap

Oct 26, 2017

脱360的壳时需要给mmap函数下断点，总结IPC的共享内存时也需要用到mmap，所以本文就先总结一下mmap相关内容。

一、Linux文件基础

1、fd

fd（File Descriptor）是内核为了管理已打开的文件所创建的索引，是一个非负整数，在open时产生。所有执行I/O操作的系统调用都会通过文件描述符。

文件描述符	用途	POSIX名称	stdio流
0	标准输入	STDIN_FILENO	stdin
1	标准输出	STDOUT_FILENO	stdout
2	标准错误	STDERR_FILENO	stderr

如果打开一个新的文件，它的fd会是3。

POSIX（Portable Operating System Interface of UNIX，可移植操作系统接口）标准规定每次打开文件时必须使用当前进程中最小可用的文件描述符号码。

1.1 限制

理论上系统内存有多大就可以打开多少的fd，但实际上内核允许最大的打开文件数是系统内存的10%（KB计算），如果超出这个数，就会提示“Too many open files”。可以使用sysctl -a | grep fs.file-max命令查看系统的最大打开文件数。这也就是系统级限制。

此外，为了防止一个进程消耗掉所有的文件资源，单个进程最大打开的文件数有限制，一般为1024，可通过ulimit -n查看。这就是用户级限制。

2、 fd和文件之间的关系

- 每个fd会与一个打开的文件对应；

- 不同的fd也可能会指向同一个文件

- 相同的文件可以被不同的进程打开，也可以在同一个进程中被多次打开

  系统为每个进程都维护了一个文件描述符表，该表的值都是从0开始， 进程凭借fd的值能通过文件描述符表快速查询对应的文件。

所以在不同的进程中会看到相同的fd，但它们可能指向同一个文件，也可能指向不同的文件。

3、内核维护的3个数据结构

- 进程级的文件描述符表
- 系统级的打开文件描述符表
- 文件系统的i-node表

3.1 inode表

inode是什么？

- 文件存储在硬盘上，硬盘的最小存储单位叫扇区（Sector），每个山区存储512字节（0.5KB）
- 操作系统在读取硬盘的时候，为了效率，会一次性连续读取多个扇区，也就是块（block）。块也就是文件存取的最小单位，它的大小最常见的为4kb，即8个连续的扇区组成一个块
- 因此文件的数据存储在块中，而存储文件的元信息（比如文件的创建者、创建时间、创建的大小等）的区域就叫做inode(index node， 索引节点)

inode的内容

- 文件的字节数
- 文件拥有者的User ID
- 文件的Group ID
- 文件的读、写、执行权限
- 文件的时间戳：ctime（inode上一次变动的时间）、mtime（文件内容上一次变动的时间）、atime（文件上一次打开的时间）
- 链接数，即有多少个文件名指向这个inode
- 文件数据block的位置

inode号码

每个inode都有一个号码，操作系统用inode号码来识别不同的文件。

其实在Unix/Linux系统中在识别文件时，不使用文件名而是使用inode号码。
对于系统来说，文件名只是inode号码便于识别的别称或绰号。

当用户通过文件名打开文件时，系统内部分为3个流程

1. 系统找到这个文件名对应的inode号码
2. 通过inode号码获取inode信息
3. 根据inode信息找到文件数据block的位置，读出数据

目录文件
因为Linux中万物皆文件，因此目录（directory）也是一种文件。
目录的结构其实就是一系列目录项（dirent）的列表。
每个目录项的内容为：

- 所包含文件的文件名
- 该文件名对应的inode号码

硬链接（**hard link**）
一般情况下，文件名和inode号码一一对应，每个inode号码对应一个文件名。

但Unix/Linux系统也允许多个文件名指向同一个inode号码，即可以用不同的文件名访问同样的内容。

对文件内容进行修改，会影响到所有文件名。但删除一个文件名，并不影响另一个文件名的访问。

软链接（**soft link**）
有一种特殊情况。
文件A和B的inode号码虽然不一样，但是文件A的内容是文件B的路径。即读取文件A时，系统会自动将访问者导向B。
因此，不论打开哪一个文件，最终读取的都是文件B。这时候文件A就称为文件B的软链接或符号链接（Symbolic link）。

ς是的文件A依赖于文件B存在，如果删除文件B，那么打开文件A就会报错：『No such file or

directory』。

这也就是软链接与硬链接最大的不同：文件A指向文件B，而不是文件B的inode号码，文件B的inode链接数不会因此发生变化

3.2 进程级的文件描述符表

该表的每一条目均记录了单个文件描述符的相关信

1. 控制文件描述符操作的一组标志（如close-on-exec）
2. 对打开文件句柄的引用

3.3 系统级的打开文件描述符表

内核对所有打开的文件维护有一个系统级的描述符表格（**open file description table**），也被称为打开文件表（**open file table**），并将表格中各条目称为打开文件句柄（**open file handle**）。一个打开文件句柄存储了与一个打开文件相关的全部信息

1. 当前文件的偏移量（调用read()和write()时更新，或使用lseek()直接修改）
2. 打开文件时所使用的状态标识（如open()的flags参数）
3. 文件访问模式（调用open时的只读、只写或读写模式）
4. 与信号驱动相关的设置
5. 对该文件inode对象的引用
6. 文件类型（如常规文件、套接字、FIFO）和访问权限
7. 指向该文件所持有的锁列表的指针
8. 文件的各种属性，包括文件大小、不同类型操作相关的时间戳


文件描述符、打开的文件句柄以及i-node之间的关系

图为：文件描述符、打开的文件句柄以及**i-node**之间的关系

在上图的进程A中，文件描述符1和30都指向了同一个打开的文件句柄（标号23）。这可能是通过调用dup()、dup2()、fcntl()或者对同一个文件多次调用了open()函数而形成的。

进程A的文件描述符2和进程B的文件描述符2都指向了同一个打开的文件句柄（标号73）。这种情形可能是在调用fork()后出现的（即，进程A、B是父子进程关系），或者当某进程通过UNIX域套接字将一个打开的文件描述符传递给另一个进程时，也会发生。再者是不同的进程独自去调用open函数打

开了同一个文件，此时进程内部的描述符正好分配到与其他进程打开该文件的描述符一样。

此外，进程A的描述符0和进程B的描述符3分别指向不同的打开文件句柄，但这些句柄均指向i-node表的相同条目（1976），换言之，指向同一个文件。发生这种情况是因为每个进程各自对同一个文件发起了open()调用。同一个进程两次打开同一个文件，也会发生类似情况。

4、fd与filp

fd只是一个整数，起索引的作用，进程通过PCB（Process Control Block）中的文件描述符表找到该fd所指向的文件指针filp（file pointer），filp返回strcut file*结构指针。

二、I/O缓冲区

1、 概念

与高速缓存（cache）产生的原理类似，在I/O过程中，读取磁盘的速度相对内存读取速度要慢的多。因此为了能够加快处理数据的速度，需要将读取过的数据缓存在内存里。而这些缓存在内存里的数据就是高速缓冲区（buffer cache），下面简称为“buffer”。

具体来说，buffer（缓冲区）是一个用于存储速度不同步的设备或优先级不同的设备之间传输数据的区域。一方面，通过缓冲区，可以使进程之间的相互等待变少，从而使从速度慢的设备读入数据时，速度快的设备的操作进程不发生间断。另一方面，可以保护硬盘或减少网络传输的次数。

2、Buffer和Cache

buffer和cache是两个不同的概念：cache是高速缓存，用于CPU和内存之间的缓冲；buffer是I/O缓存，用于内存和硬盘的缓冲；
简单的说，cache是加速“读”，而buffer是缓冲“写”，前者解决读的问题，保存从磁盘上读出的数据，后者是解决写的问题，保存即将要写入到磁盘上的数据。

3、Buffer Cache和 Page Cache

uffer cache和page cache都是为了处理设备和内存交互时高速访问的问题。buffer cache可称为块缓冲器，page cache可称为页缓冲器

缓存备，page cache可称为页缓存备。

在linux不支持虚拟内存机制之前，还没有页的概念，因此缓冲区以块为单位对设备进行。在linux采用虚拟内存的机制来管理内存后，页是虚拟内存管理的最小单位，开始采用页缓冲的机制来缓冲内存。

Linux2.6之后内核将这两个缓存整合，页和块可以相互映射，同时，页缓存page cache面向的是虚拟内存，块I/O缓存Buffer cache是面向块设备。需要强调的是，页缓存和块缓存对进程来说就是一个存储系统，进程不需要关注底层的设备的读写。

buffer cache和page cache两者最大的区别是缓存的粒度。buffer cache面向的是文件系统的块。而内核的内存管理组件采用了比文件系统的块更高级别的抽象：页page，其处理的性能更高。因此和内存管理交互的缓存组件，都使用页缓存。

4、页缓存Page Cache

页缓存是面向文件，面向内存的。通俗来说，它位于内存和文件之间缓冲区，文件IO操作实际上只和page cache交互，不直接和内存交互。page cache可以用在所有以文件为单元的场景下，比如网络文件系统等等。

5、文件读写基本流程

5.1 读文件

1. 进程调用库函数向内核发起读文件请求；
2. 内核通过检查进程的文件描述符定位到虚拟文件系统的已打开文件列表表项；
3. 调用该文件可用的系统调用函数read()
4. read()函数通过文件表项链接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的inod
5. 在inode中，通过文件内容偏移量计算出要读取的页；
6. 通过inode找到文件对应的address_space；
7. 在address_space中访问该文件的页缓存树，查找对应的页缓存结点：
   （1）如果页缓存命中，那么直接返回文件内容；
   （2）如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过inode找到文件该页的磁盘地
8. 文件内容读取成功。

.2 写文件

前5步和读文件一致，在address_space中查询对应页的页缓存是否存在：

6、如果页缓存命中，直接把文件内容修改更新在页缓存的页中。写文件就结束了。这时候文件修改位于页缓存，并没有写回到磁盘文件中去。

7、 如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过inode找到文件该页的磁盘地址，读取相应的页填充该缓存页。此时缓存页命中，进行第6步。

8、一个页缓存中的页如果被修改，那么会被标记成脏页。脏页需要写回到磁盘中的文件块。有两种方式可以把脏页写回磁盘：
（1）手动调用sync()或者fsync()系统调用把脏页写回
（2）pdflush进程会定时把脏页写回到磁盘

同时注意，脏页不能被置换出内存，如果脏页正在被写回，那么会被设置写回标记，这时候该页就被上锁，其他写请求被阻塞直到锁释放。

三、内核空间和用户空间

Linux的虚拟地址与线性地址总是一致，Linux的虚拟地址空间跟线性地址一样，为0～4G。Linux内核将这4G字节的空间分为两部分。将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为“内核空间”。而将较低的3G字节（从虚拟地址 0x00000000到0xBFFFFFFF），供各个进程使用，称为“用户空间）。

每个进程可以通过系统调用进入内核，因此，Linux内核由系统内的所有进程共享。

故，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟空间。

内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。

其实说白了也就是：两个的权限不一样，处理的事务不一样。内核空间能调系统调用，用户空[illegible]只能调用户调用

四、mmap

mmap是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映关系。


进程虚拟地址空间

从上图可以了解，进程的虚拟地址空间由多个虚拟内存区域构成。虚拟内存区域是虚拟内存空间中具有同样特性的连续地址范围。

上方的text数据段、初始数据段、bss数据段、堆、内存映射、栈，每个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在堆在之间的空余部分。

vm_area_struct

Linux内核使用vm_area_struct结构表示一个独立的虚拟内存区域，每个vm_area_struct结构使用链表或者树形结构连接，方便进程快速访问。它包含区域其实和终止地址以及其他相关信息，同时也包括一个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。

因此进程对某一虚拟内存区域的任何操作需要用到的信息都可以直接从vm_area_struct中获得。

mmap也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相连。

1、mmap内存映射的流程

###（一）进程在虚拟地址空间中创建虚拟映射区域

1. 进程在用户空间调用库函数mmap
2. 在当前进程句柄的虚拟地址空间中，寻找一段空闲且满足要求的连续虚拟地址
3. 为此虚拟区分配一个vm_area_struct结构并初始化它
4. 将新建的vm_area_struct结构插入到进程的虚拟地址区域链表或树中

```
oid mmap(void start, size_t length, int prot, int flags, int fd, off_t offset);
```

> *start*：映射区的开始位置
> *length*：映射区的长度
> *port*：期望的内存保护标志，不能与文件的打开模式冲突。可以通过*or*运算组合。
> *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参数的组合
> *fd:* 文件描述符
> *offset*：被映射对象内容的起点

port的值可以为：

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

flag的参数类型：

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
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。
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
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。
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
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

###（二）调用内核空间的系统调用函数mmap，实现文件物理地址和进程虚拟地址的映射关系

1. 通过待映射的文件指针filp，在文件描述符表中查询对应的fd，通过fd，链接到内核已打开文件表（Open fil
2. 通过该文件结构体，连接到file_operations模块，调用内核函数mmap

3. 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
4. 通过remap_pfn_range函数建立页表，即实现文件地址和虚拟地址的映射关系。

另外，此时的系统调用函数mmap与库函数mmap不同。它的定义如下：

int mmap(struct file *filp, struct vm_area_struct* vma)

里面两个参数的定义就是文件指针filp和虚拟空间结构体

总结：其实也就是通过一系列的手段获取文件磁盘的物理地址，从而与虚拟地址映射

###（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

在前两步完成之后，只完成了地址映射，但没有将任何文件数据拷贝到主存。

当进程发起读或写操作时才会执行文件读取

1. 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
2. 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。
3. 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
4. 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

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。

2 mmap和常规文件操作的区别

2.1 常规文件:

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，从而找到该文件的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，直接返回
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，然后再发起读页面的过程，从而

实际上呢，页缓存存在于内核空间，用户空间是不能对它直接寻址的。所以还需要将页缓存中的数据再次拷贝到内存对应的用户空间去才行。

所以也就会有两次拷贝。

2.2 mmap

它实现了用户空间和内核空间的直接交互，就省去了额外拷贝的过程，所以就只会有一次拷贝。

优点：

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
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
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

TAGGED IN

汇编 漏洞