Linux 虚拟内存 和 VFS 研究报告

1831604 张尹嘉

1. VFS 中 inode 和 address_space

1.1 inode

Linux中，操作系统处理一个文件所需要的所有信息都被存储在一个叫做inode的数据结构中。一个文件在其存在期间只对应唯一的一个inode对象。《Understanding the Linux Kernel, 3rd Edition》的 12.2节中列举了inode结构体所有的fields。 以下只列出其中一些属性：

Type	Field	Description
struct list_head	i_list	描述inode状态的链表头
struct list_head	i_dentry	一个引用这个inode的目录项链表头
unsigned long	i_ino	inode id
atomic_t	i_count	被使用计数
umode_t	i_mode	文件类型和权限
unsigned int	i_nlink	hard link的数量
uid_t	i_uid	所属用户id
gid_t	i_gid	所属用户组id
dev_t	i_rdev	实际对应的设备id
loff_t	i_size	文件大小(byte)
unsigned int	i_blkbits	块大小(bits)
unsigned long	i_blksize	块大小(bytes)
unsigned long	i_blocks	文件块数量
unsigned short	i_size	最后一个文件块中数据的大小(bytes)
unsigned char	i_sock	如果该文件是一个socket，该值不是0
struct semaphore	i_sem	inode对象的信号量
struct rw_semaphore	i_alloc_sem	对文件直接I/O操作时防止竞争条件的信号量
struct inode_operations*	i_op	指向与该inode对象有关的操作的指针
struct file_operations*	i_fop	指向与该文件有关的操作的指针
struct address_space*	i_mapping	指向一个address_space对象的指针
struct address_space	i_data	文件的address_space对象

Type	Field	Description
atomic_t	i_writecount	写进程的计数
...	...	...

inode对象中存储了一个文件的元信息以及文件所在磁盘块的信息。操作系统读取文件时，先读取inode，然后通过inode中的信息，读取磁盘上对应的块。 同时，inode对象中还记录了这个文件对应的address_space信息。

1.2 address_space

address_space 是一个嵌入在页所属inode对象中的数据结构，是连接文件系统和页缓存的关键数据结构。缓存中许多页可能指向同一个拥有者，因此这些页都会与同一个address_space 对象关联。address_space用一棵radix tree来保存和管理页缓存中与其对应的页， 还会保存一个指向与该address_space对应的inode对象的指针。address_space中一些关键成员如下：

Type	Field	Description
struct inode *	host	指向对应inode对象的指针
struct radix_tree_root	page_tree	指向管理有关页号的radix-tree的根结点
spinlock_t	tree_lock	保护radix-tree的自旋锁
unsigned long	nrpages	所有者在缓存中的总页数
...	...	...

page_tree的使用场景如下:

- 假设读文件A 5000字节
  - 找到A的address_space结构的radix树(page_tree)
  - 5000/4096 = 1 -> 读文件的#1逻辑页
  - 用 #1 作为索引，去查询radix树上与 #1 对应的叶子节点
    - 如果 #1 叶子节点非0, 假设数值为2356, 说明 #2356 页框中有要访问的数据且存在于缓存中
    - 如果 #1 叶子节点是0, 为文件A分配一个主存页框（假设为9876）， 把9876填入radix树#1叶子节点， 然后IO磁盘。IO命令是读A文件的#1号逻辑页装入9876页框(通过DMA命令)
  - 取物理页框（2356/9876）中的904个字节

2. fork() 和 exec()

2.1 fork()

Linux 系统中， 轻量级的进程是由clone()函数创建的。clone()函数接收一些参数，包括：

1. fn: 新进程要执行的函数
2. arg: 传递给fn的参数
3. flags: 由一系列CLONE_*定义的信息
4. chid_stack: 要传递给子进程的esp寄存器的用户栈的指针

以及其他与flag有关的参数。 而fork()函数就是通过调用clone()函数实现的。fork()函数在调用clone()时将flag设置为SIGCHLD以及所有清零的CLONE_*标志。

clone()函数底层是由do_fork()系统调用执行的。当通过fork()调用do_fork()时，执行步骤如下：

1. 分配新的PID
2. 检查父进程的ptrace字段，如果ptrace!=0， 说明另一个进程正在跟踪父进程，因此要检查debugger程序时候想跟踪子进程。如果想跟踪并且子进程不是内核态的，那么就会设置CLONE_PTRACE标志为1.
3. 调用copy_process()复制进程描述符。copy_process()函数返回刚刚创建的task_struct的地址。在复制的过程中， 分配一个新的task_struct空间以及task_struct包含的指针代表的空间(例如tty, fs, files, signal, mm)等。并将父进程task_struct中的值复制到子进程task_struct中对应空间。特别的，父进程在将PTE复制给子进程前，不会在缓存中复制逻辑页面，而是将RW改成RO并将PTE的count++后复制给子进程。后续无论父进程还是子进程谁先执行，都可以利用Copy-on-Write机制复制缓存页框。
4. 如果设置了CLONE_STOPED标志或者子进程需要被跟踪，那么设置子进程的的相应状态和信号量。
5. 如果没有设置CLONE_STOPED，调用wake_up_new_task()函数，执行下列操作:
    1. 调整父进程和子进程的调度参数
    2. 如果父进程和子进程运行在同一个CPU上，且父子进程不能共享同一组页表，就把子进程插入父进程运行队列并恰好在父进程之前。(为了在子进程刷新其地址空间并被创建之后执行新程序的情况下获得更好的性能。如果父进程先执行，会调用Copy-on-Write，但是子进程的地址空间并不会被利用，而是进行了刷新。)
    3. 否则把子进程插入父进程运行队列的队尾。
6. 进行跟踪相关的操作
7. 返回子进程PID

2.2 exec()

调用exec()函数后，会将现有进程的进程映像替换为新的进程映像，相当于将当前执行的程序替换成一个新的程序，并且会重新初始化代码段，数据段以及堆栈段。为了避免原有进程被杀死，可以先调用fork()，然后在子进程中调用exec().

调用exec()函数后，会对当前运行进程执行以下操作:

1. 清空mmap
2. 清空用户态页表，原来页表映射的所有页框count--
3. 从文件读取可执行文件的文件头，读取libc.so程序头，读取ld程序头，读取动态链接库程序头，构造mmap链表。
4. 申请两个主存页框作为初始用户栈，写页表中负责映射用户空间最后两个逻辑页面的PTE。写入环境变量和命令行参数。
5. exec 返回。

随后开始执行新的程序。假设执行main函数：

1. 当执行第一条指令时，一定会发生缺页，此时会引发代码段的缺页调用将代码页面填好。
2. 当访问全局变量时，如果发生了缺页，假设是第i号页面：
    1. 如果PTE[i]==0, 说明是第一次访问该变量， 就从对应的address_space中的page_tree中读取文件中的初值
    2. 如果PTE[i]!=0, 说明变量被换出，因此读取交换区，swap-in有关变量。
3. BSS段全部刷成0, 具体见第三节。

3. 缺页中断

当进程给出了逻辑地址VA， MMU(内存管理单元)会将VA转化为逻辑页号，随后用这个逻辑页号查询现运行进程页表，如果无法进行地址映射，则发生缺页中断。

缺页中断的处理:

1. 检查task_struct中的mm字段指向的mm_struct中的mmap, 查看VA是否落在合法的内存区域
    1. 如果找到和VA对应的内存区域，说明合法；进行后续步骤
    2. 如果没有找到，但是VA访问的是当前active的堆栈的栈顶附近，那么也是合法的。这种情况下是栈的push操作引发的缺页中断。(附近是指 esp-VA<32bytes)。此时找到堆栈对应的在mmap中的vm_area_struct, size+=4k, vm_start -= 4k(扩充堆栈)，并分配新的主存页框并写PTE。这种缺页没有进行磁盘的I/O，叫做minor page fault。
2. 如果所缺的页是代码页(假设vm_area_struct是B):
    1. X = VA - B.vm_start + vm_pgoff 是CPU中下一条指令在文件中的地址
    2. 将floor(X/4K)作为逻辑页号，在address_space的radix-tree中进行查询：如果radix-tree中没有相应的叶节点，说明还没有其他进程运行过该程序，进行I/O操作；如果有读影叶子节点，说明其他进程也执行了该程序；因为代码段在内存中只有唯一的一份。查询后将radix-tree中对应的叶节点的值添加到PTE，并把代码文件中的prot填入PTE。
3. 如果缺页的是BSS段，Linux会返回ZERO_PAGE(全局变量)，PTE[i].Base = ZERO_PAGE;只有在写操作的时候才会分配主存页框(写时分配)。
4. 如果缺页的是数据段：
    1. PTE[i]!=0, 说明fork()之前父进程访问过该页面。此时需要检查PTE[i].Base对应页框的count
        1. if count > 1: 说明fork后还没有其他进程访问过该页框；此时分配新的页框，挂在自己的页表，复制原来页框到新页框，old page count --, new page RO->RW；
        2. if count == 1: 说明其他进程已经复制过该页框，可以直接将RO改成RW。
        3. 上述过程就是就是Copy-on-Write;
    2. PTE[i]==0(例如exec后立即fork): 父子进程分别从可执行文件中读取数据，分配页框，将页号挂在radix-tree中并写PTE。