Linux 内核中 Demand paging 与 swap 机制的源码分析

王森
msn: kernel_senix@163.com

1 Swap partition 与 file-backed memory

首先对于每个文件，内核里都有一个 **struct address_space** 结构来管理,他有两个作用
1 该文件所有被读进内存的页,都被加进 struct address_space 结构
2 该结构提供读写例程，内核对该文件所有的读写都通过这些例程完成。这些例程向下跟文件系统连接起来。

其次，所谓 **swap partition** 可以是个普通文件、也可是个设备文件，对其管理也是当成文件来处理，自然就有对应的 struct address_space：
struct address_space swapper_space。用户进程的 stack/heap 内存都会通过 swapper_space 写出去。

所谓 **file-backed memory** 就是把一块连续的虚拟内存与文件映射起来，对虚拟内存的首次访问会产生 demand panging，内核分配物理内存，并把对应的文件内容读进物理内存，而这些物理内存以 page 为单位放在文件的 struct address_space 里。

可见 **swap partition** 机制和 **file-backed memory** 机制的区别仅仅在于换出目的不同。

另外 demand paging 产生时会内核动作分为 3 步
1）分配物理内存
2）将存放在 swap partition 或 backed 在文件系统里的内容读入内存
3）修复页表项

2 Swap out 机制分析

linux 内核会管理所有的用户进程使用物理内存（包括上文提到的 stack/heap 内存，和 file-backed memory），单位为 page（4k），这些 page 被按最近使用的频率的穿起来，如果内存紧张了，就把很久没有用的 page 释放掉、或者 swap out 出去。而根据使用 **swap partition** 机制还是 **file-backed memory** 机制，产生不同的目的地。

/mm/vmscan.c
//在进入这个函数前，内核已经把需要 swap out 或者释放掉的页放在 page_list 里了
static unsigned long shrink_page_list(struct list_head *page_list,…)
{
...
//这个 page_list 就是一串很久没有被用到的页，逐个从这个 page_list 取出页，做以下操作：

//如果定义了系统支持 swap partition 下面这部分橙色代码会被编译进来,而对于 android 系统,下边这部分灰底代码是无效的，**内核 swap partition 和 file-backed memory 的区别仅仅就在这里**。这部分代码就是为依靠 swap partition 机制换出的匿名内存绑定目的地：swapper_space

```
#ifdef CONFIG_SWAP
        if (PageAnon(page) && !PageSwapCache(page)) {
//这个 page 是匿名的(动态分配的 stack/heap,没有对应的文件),并且没有和 swap cache 联系起来，所谓 swap cache，就是 swap partition 的  swapper_space
            if (!add_to_swap(page, GFP_ATOMIC))//将这个匿名页加入到 swap cache
        …
        }
#endif /* CONFIG_SWAP */
…
        mapping = page_mapping(page);//找到这个 page 对应的 struct address_space
//依靠 swap partition 机制换出的目的地，在应用程序调用 map（）时绑定，即为对应的 file，这里就是找到这个 file 的 struct address_space
//对于依靠 swap  partition 的内存，其换出目的地在灰底代码部分绑定，这里是也将其找出来
    …
        if (PageDirty(page)) {//对应的 page 是被修改的,该条件成立

            /* Page is dirty, try to write it out here */
            pageout(page, mapping, sync_writeback)//把修改的页写出去
//mapping 就是换出目的地
//可见无论通过 Swap partition 还是 file-backed memory 做 swap out 其本质都是一样的，不同的//仅仅是 swap out 的目的地不一样。一个是 swap partition，一个是文件系统上的普通文件。
                ...
                    ...
//如果这个 page 是 clean 的内存，简单释放即可
        ...
}

/*
 * pageout is called by shrink_page_list() for each dirty page.
 * Calls ->writepage().
 */
static pageout_t pageout(struct page *page, struct address_space *mapping,
                         enum pageout_io sync_writeback)
{
    …
        res = mapping->a_ops->writepage(page, &wbc);//writepage 为文件系统提供例程，
    …
}
```

3 demand paging 和 swap in 分析

demand paging 的产生必定是对应页表项的解析出现异常，内核会转向对该页表异常的处理

```
handle_page_fault()->int handle_mm_fault()->
static inline int handle_pte_fault(struct mm_struct *mm,
        struct vm_area_struct *vma, unsigned long address,
        pte_t *pte, pmd_t *pmd, int write_access)
{

//对于每块建立起来的用户虚拟地址，内核有一个对应 struct vm_area_struct *vma 来管理她
…
if (!pte_present(entry)) {
        if (pte_none(entry)) {
            if (vma->vm_ops) {
//对于 file-backed memory 走到这里，因为这种内存使用前会建立起和文件系统的映射，所以 vma->vm_ops 不会为空，蓝色部分为相关代码
                if (likely(vma->vm_ops->fault))
                    return do_linear_fault(mm, vma, address,
                        pte, pmd, write_access, entry);
            }
//对于第一次 demand paging 的匿名 memory 走到这里，红色部分为相关代码
            return do_anonymous_page(mm, vma, address,
                        pte, pmd, write_access);
        }
        if (pte_file(entry))//这是对同一地址同时有多个映射，嵌入式系统不会用到
            return do_nonlinear_fault(mm, vma, address,
                    pte, pmd, write_access, entry);
//对于 swap 出去的匿名 memroy，绿色部分为相关代码
        return do_swap_page(mm, vma, address,
                    pte, pmd, write_access, entry);
    }


//COW 的处理，不用管它
    if (write_access) {
        if (!pte_write(entry))
            return do_wp_page(mm, vma, address,

...

}
```

```
//对于 file-backed memory  走到这里
static int do_linear_fault()-〉
static int __do_fault(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pmd_t *pmd,
        pgoff_t pgoff, unsigned int flags, pte_t orig_pte)
{
    ...

    ret = vma->vm_ops->fault(vma, &vmf);//这里对应的就是被 map 的文件系统处理函数:

//在一个虚拟内存被映射到文件的时候，内核会建立 struct vm_area_struct 来管理这块虚拟内存，
//而这块内存显然应该由对应的文件系统处理，这个处理函数就放在 struct vm_area_struct 结构的 vm_ops 里
//一般文件系统的处理例程如下：
//一般的文件系统都使用 generic_file_vm_ops
//struct vm_operations_struct generic_file_vm_ops = {
//    .fault          = filemap_fault,
//};

...
}

int filemap_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
...
page = find_lock_page(mapping, vmf->pgoff);//分配处物理内存,该页内存同时加入该文件的 struct address_space
...
error = mapping->a_ops->readpage(file, page);//启动文件读
...
}


//对于匿名页，如 stack/heap 的异常走到这里

static int do_anonymous_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        int write_access)
{
...
page = alloc_zeroed_user_highpage_movable(vma, address);//分配处物理内存
...
set_pte_at(mm, address, page_table, entry);//重置产生异常的页表项
```

```
}


//如果该页被 swap 出去又被用户进程重新访问，与匿名页的区别在于这是交换出去的匿名内存
static int do_swap_page(struct mm_struct *mm, struct vm_area_struct *vma,
        unsigned long address, pte_t *page_table, pmd_t *pmd,
        int write_access, pte_t orig_pte)
{

page = lookup_swap_cache(entry);//在上文提到的 swapper_space 尝试找到产生异常的页

if (!page) {//没有找到，说明这个匿名页被 swap 出去了
    ...
        page = swapin_readahead(entry,GFP_HIGHUSER_MOVABLE, vma, address);//把对应的页重新 swap in 近来
    ...
    }

set_pte_at(mm, address, page_table, pte);////重置产生异常的页表项
```