计算机科学2002Vol.29№.4

Linux 内存管理机制的分析与研究*)

The Analysis and Research of Memory Management

李小群　孙玉芳

(中国科学院软件研究所　北京中科红旗软件技术有限公司　北京100080)

Abstract　The memory management of Linux is very complicated. This paper mostly discusses several aspects as followed: Virtual address to physical address mapping; The organization of Linux memory structure and the relationship of them; Swapping space and Page in&out management; Lastly, Linux kernel2.4's big change in demand paging is discussed in detail.

Keywords　Memory management, Memory mapping, Virtual memory, Demand Paging , Page in&out

1. 引言

内存管理在操作系统中不仅非常重要,而且很复杂。利用虚拟存贮技术,Linux 使得一个只有有限内存资源的计算机可以为每个进程提供多达4GB 的虚拟内存空间。其基本实现思路是通过进程映象和分页机制在内存和二级存储——对换空间之间传送数据,充分利用宝贵的内存资源。另外,Linux 虚拟内存管理机制把用户空间和核心空间分开,这样不仅有效地保护了核心空间,各个进程之间也互不影响。

2. 地址映射机制

由 INTEL 公司推出的32位80386芯片的工作模式包括实地址模式和虚地址模式。实地址模式与8086完全兼容,它的寻址范围是1MB 的地址空间,分段功能受到限制,不能区分特权级,当然分页机制也不能启用。在虚地址模式下,分段机制得到加强,段最大可达4GB,并且提供段内分页管理机制,为 Linux 虚拟内存管理机制提供了支持。

80386的虚拟地址模式使用了如下分段和分页两级地址转换机制来实现虚拟地址向物理地址的转换。

2.1　虚拟地址向线性地址的转换

用户进程要访问的虚拟地址包括一个16位的段选择器和一个32位的段内偏移,80386的分段机制将段寄存器中所装的段选择器和32位段内偏移量相加,得到32位的线性地址,如图1所示,16位的段选择器最低两位表示请求者的特权级,那么最多可以有16k 个段,每段的最大尺寸为4GB。但是它们都必须被映射到4GB 的线性地址空间。

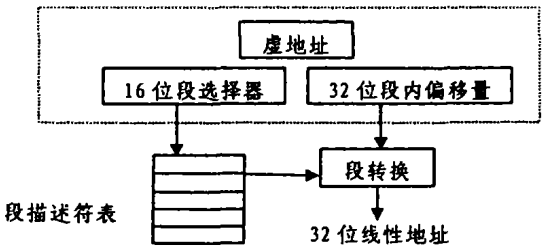

图1　保护模式下利用段机制虚拟地址向线性地址的转换

2.2　线性地址向物理地址的转换

Linux 的每个用户进程都可以访问4GB 的线性地址空间,而实际的物理内存可能远远少于4GB,采用分页机制,Linux 仅把可执行映像的一小部分装入物理内存,当需要访问未装入的页面时,系统产生一个缺页中断,把需要的页读入物理内存。

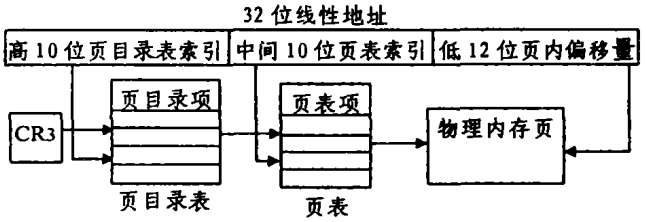

图2　保护模式下利用分页机制线性地址向物理地址的转换

Linux 采用两级页表结构——页目录表和页表实现地址映射,当前正在运行进程的页目录表的地址被保存在控制寄存器 CR3中。

由上面转换机制所得到的线性地址可以分为3部分,高10位是 DIR 域——页目录表的索引值,它与 CR3中的地址一起计算得到页表的物理地址,中间10位保存相对于页表的索引值,通过它得到所需的物理页号。物理页号与低12位页内偏移组合得到物理地址。其结构如图2所示。

3. 虚存的组织和管理

每个用户进程都可以有4GB 的虚存空间,为了更好地管理这部分虚存空间,Linux 主要定义了如下三个数据结构:

struct vm_area_struct, struct vm_operations_struct 和 struct mm_struct

虚存段(vm_area_struct),简称 vma 是某个进程的一段连续的虚存空间,一个进程通常占用几个 vma 段,例如代码段、数据段、堆栈段等。vma 不仅可以代表一段内存区间,也可以对应于一个文件、共享内存或者对换设备。

每一个进程的所有 vma 由一个双向链表管理。为了提高对 vma 的查询、插入、删除等操作的效率,Linux 把系统中所

*)本文得到国家自然科学基金项目60073022、国家863项目8633-306-2D12－14-2和中科院知识创新重大项目 KGCX1-09的资助。李小群　博士研究生,研究方向:系统软件。孙玉芳　博士生导师,研究员,研究方向:系统软件,中文信息处理,大型数据库与网络工程。

有进程的 vma 组成了一棵 AVL 树。这是一棵平衡二叉树，当 vma 数量特别大时，利用这棵 AVL 树查找 vma 的效率得到明显提高。

不同的 vma 可能需要不同的操作处理方式，但同时考虑到系统接口的统一性，Linux 采用 vm_operations_struct 结构和面向对象的思想来定义操作方式，一个 vm_operations_struct 结构体是一组函数指针，对于不同的 vma，它可能指向不同的处理函数，例如当发生缺页错误时，共享内存和代码段的 readpage 所指向的页面读入函数可能就不同。

内存管理中另外一个非常重要的数据结构是 struct mm_struct 结构体，进程的 task_struct 中的 mm 成员指向它，当前运行进程的整个虚拟空间都由它来管理和描述，它不仅包含该进程的映像信息，而且它的 mmap 成员项指向该进程所有 vma 组成的链表，它的 mmap_avl 成员项指向整个系统的 AVL 树。

这三个数据结构之间相互关联，共同管理虚拟内存，它们之间的关系如图3所示。

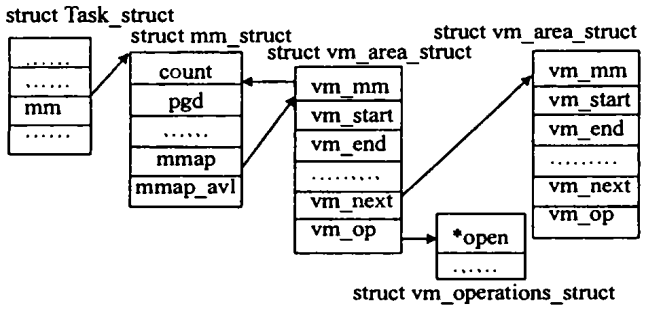

图3　进程虚拟地址管理的主要数据结构图

这部分相关的系统调用主要有如下两个：

```
do_mmap(struct file * file,unsigned long addr,unsigned long len,
    unsigned long prot,unsigned long flags,unsigned long off)
find_vma(struct mm_struct * mm,unsigned long addr);
```

do_mmap 函数实现了内存映射。find_vma 函数的功能是找到包含参数 addr 指定的虚拟地址所属的 vma。

当要运行一个可执行映像时，调用 do_mmap 将其装入到该进程的虚拟地址空间，并且产生一组 vma 结构，如前所述该进程的整个虚拟空间由 mm_struct 结构管理，但是此时可执行文件仅仅被连接到进程的虚拟空间中，只有一小部分页面被装入到物理内存，其余大部分并没有被真正装入到物理内存，在进程的运行过程中，产生缺页错误，操作系统首先调用 find_vma，找到该虚拟地址所在的 vma，然后根据该 vma 的成员变量 vm_ops 指向的 vm_operations_struct 结构中的缺页操作函数，把页装入物理内存。

4. 对换空间

Linux 的每个进程可以有4GB 的虚拟内存空间，而且系统中还要同时存在多个进程，但是，事实上大多数计算机都没有这么多物理内存空间，当系统中的物理内存紧缺时，就需要利用对换空间把一部分未来可能不用的页面从物理内存中移到对换设备或对换文件中。

Linux 采用两种方式保存换出的页面。一种是利用整个块设备，如硬盘的一个分区，即对换设备，另一种是利用文件系统中固定长度的文件，即对换文件。它们统称为对换空间。这两种方式的相同之处是它们的内部格式一致，但是在执行效率方面，对换设备要好一些，这是因为对换设备上同一页面的数据块是连续存放的，故而可以顺序存取，而在对换文件中，同一页面的数据块实际的物理位置可能是不连续的，需要通过对换文件的 inode 检索，这就降低了存取效率。

每个对换文件或对换设备由 struct swap_info_struct 结构来描述。

有关对换设备的函数主要是 get_swap_page(…)，当内存中的页面需要被换出时，调用 get_swap_page 函数申请得到一个对换空间中的物理页面，如果成功，就返回一个非零代码，否则，返回0。

5. 分页管理机制

Linux 使用分页管理机制来更加有效地利用物理内存，当创建一个进程时，仅仅把当前进程的一小部分真正装入内存，其余部分需要访问时，处理器产生一个页故障，由缺页中断服务程序根据缺页虚拟地址和出错码调用写拷贝函数 do_wp_page、此地址所属的 vma 的 vm_ops 指向的 nopage、do_swap_page，swap_in 等函数将需要的页换入物理内存。

随着可执行映像的运行和页面的换入，系统中的内存有可能变得不足，这时 Linux 核心就必须调用 kswapd 守护进程释放部分物理内存。kswapd 在系统启动时由 init 进程建立。在系统的运行过程中，它被定期唤醒，检查系统中的空闲物理内存是否很少，如果是，则释放一部分内存，或者将一些页面换出到对换空间，然后继续睡眠。

5.1　缺页中断和页面换入

页面换入主要由缺页中断服务入口函数 do_page_fault 来实现，当系统中产生页面故障时，如果虚拟内存地址有效，则产生错误的原因有如下两种：

·虚拟内存地址对应的物理页不在内存中，那么它必然在磁盘或对换空间中，如果在磁盘上，那么我们调用 do_no_page 函数，而 do_no_page 调用 vma->vm_ops->nopage() 函数建立页面映射，从对换空间或磁盘中调入页面，或者通过 do_swap_page()函数调用 swap_in()来换入页面。

·该虚拟地址对应的物理页在内存，但是被写保护，如果这种情况发生在一个共享页面上，则需要“写拷贝“函数 do_wp_page 来换入页面，do_wp_page 函数首先调用_get_free_page 获得一新页面，然后调用 copy_cow_page 拷贝页面的内容，当然还要调用相应的刷新函数刷新 TLB 和缓存等。

5.2　页交换进程和页面换出

正如我们上面所描述的，系统使用 kswapd 守护进程来定期地换出页面，使系统中有足够的空闲物理内存页。kswapd 进程定期地检查系统中的空闲页面数，如果少于一定值，则按照以下三中途径获得空闲页面：①减少缓冲区和页面高速缓存的大小；②把共享内存占用的页面置换到对换空间；③换出或丢弃物理内存页。

6. Linux kernel 2.4所作的改进

Linux 2.4核心在内存管理这一部分的改动非常大，以下主要从页面的换入和换出等方面来讨论。

在讨论这部分之前，首先简单介绍 Linux 核心的页面缓存机制。为了提高页面的访问速度，内存中的页面管理采用了页缓存机制，所有读入的页都使用散列表结构保存在页缓存中，当系统要从内存映射文件中读取某页时，首先在页缓存中查找，如果发现该页保存在缓存中，则只需从页缓存中读取，

如果该页不在缓存中，则 Linux 内核首先分配物理页，然后从磁盘读入页面内容，并且 Linux 还会读入与此页相邻的其他页。

为了提高物理内存的利用率，尽量减少页面的换入和换出，Linux2.4核心对有关的数据结构和实现算法都作了很大的改动。改动后 struct page 的数据结构如下所示：

```
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    unsigned long age;
    wait_queue_head_t wait;
    struct page **pprev_hash;
    struct buffer_head *buffers;
    void *virtual; /* non-NULL if kmapped */
    struct zone_struct *zone;
    }
```

最重要的改动是加入了 struct address_space *mapping 成员项，它的结构如下：

```
struct address_space {
    struct list_head clean_pages; /* clean pages 链表 */
    struct list_head dirty_pages; /* dirty pages 链表 */
    struct list_head locked_pages; /* locked pages 链表 */
    unsigned long nrpages; /* 所有的页面数 */
    struct address_space_operations *a_ops; /* 操作函数 */
    struct inode *host; /* inode 和 block_device 的所有者 */
    struct vm_area_struct *i_mmap; /* 私有映射列表 */
    struct vm_area_struct *i_mmap_shared; /* 共享映射列表 */
    spinlock_t i_shared_lock;
    }
```

核心2.4所做的这些改动主要是为了实现页面缓存机制和减少对页面的换入换出，struct address_space 结构中的 struct address_space_operations 成员定义了可以对该页面执行的操作，其具体结构如下：

```
struct address_space_operations {
    int (*writepage)(struct page *);
    int (*readpage)(struct file *, struct page *);
    int (*sync_page)(struct page *);
    int (*prepare_write)(struct file *, struct page *, unsigned, unsigned);
    int (*commit_write)(struct file *, struct page *, unsigned, unsigned);
    int (*bmap)(struct address_space *, long);
}
```

我们可以看到，Linux 在这里采用了面向对象的方法，针对不同对象，可以使用不同的函数，以下通过页面的换入来简要介绍其功能。

当发生缺页错误时，核心的处理流程如下：

首先在哈希表中查找该页

```
hash=page_hash(inode->i_mapping, pgoff);
page=__find_get_page(inode->i_mapping, pgoff, hash);
```

如果找到，则返回该页，否则执行下面操作，

```
page=page_cache_alloc(); /* 为该页分配一物理页面 */
__add_to_page_cache(page, mapping, offset, hash); /* 把该页加入哈希表 */
error = mapping->a_ops->readpage(file, page); /* 从文件中读入内容到该物理页 */
```

随着可执行映象的读取和执行，页缓存中的内容会增多，物理内存有可能出现不足，Linux 内存管理子系统需要调用 kswaped 守护进程释放部分物理内存页，kernel2.4在页面换出方面也做了非常大的改动，这些改变有效地提高了系统的效率，降低了页面换入换出的频率。

Kernel2.4的物理内存页面主要由以下几个链表管理：

active_list；inactive_dirty_list；inactive_clean_list 和 free_list.

页面换出守护进程 kswaped 的主要执行流程如下：

·调用 page_launder(gfp_mask, user)，它的主要功能是将 active_list 中的干净页移到 inactive_clean_list 中，将 inactive_dirty_list 中脏页的内容写入磁盘并将其移入到 inactive_clean_list 链表。

·如果此时的物理内存仍然不够分配，则调用 refill_inactive_scan (unsigned int priority, int oneshot)，将 page->age 为0的页(即最老的页)从 active_list 移至 inactive_dirty_list 链表。

分配空闲物理页函数_alloc_pages (zonelist_t *zonelist, unsigned long order)的实现流程如下：

·如果系统中有足够的内存，则调用 rmqueue(z,order)从 free_list 链表中分配一空闲的物理页。

·如果系统中没有足够的空闲内存，则唤醒 kreclaimd_wait 进程，该进程将 inactive_clean_list 中的一页移入 free_list 链表。

小结 通过这种段页式管理方式和对换空间机制，每个进程可以访问多达4GB 的虚地址空间，但是，一般情况下，实际物理内存都非常有限，远远少于4GB。当系统中挂起的进程比较多时，内存资源就显得尤为紧缺，页面就会被频繁地换入和换出，从而使得系统的效率严重下降。为了尽量减少页面换入换出的频率，核心2.4使用多个链表队列来管理已分配物理内存页面，从而有效地提高了内存资源的利用率和系统的效率。

参考文献

1 Bach M J. The Design of the Unix Operation System. Prentice-Hall, 1990

2 Maxwell S. Linux Core Kernel Commentory. Coriolis Group, 2000

3 Vahalia U. Unix Internals-The New Frontiers. Prentice-Hall, 1996

4 Tanenbaum A S, Woodhull A S. Operation Systems Design and Implementation Second Edition. Prentice-Hall, 1997

5 Stevens W R. Advanced Programming in the Unix Environment. Addison-Wesley, 1992

6 Tanenbaum A S. Modern Operating Systems. Prentice-Hall, 1999

7 李善平，郑扣根. 操作系统及试验及教程. 机械工业出版社，1999

8 樊建平，等. Unix System V 操作系统内核代码剖析. 海洋出版社，1992