6.6 示例：Linux 存储管理

一、Linux 虚拟存储技术

Linux 操作系统采用虚拟内存管理技术，使得每个进程都有各自互不干涉的进程地址空间。该空间是块大小为 4G 的线性虚拟空间，用户所看到和接触到的都是该虚拟地址，无法看到实际的物理内存地址。利用这种虚拟地址不但能起到保护操作系统的效果(用户不能直接访问物理内存)，而且更重要的是，用户程序可使用比实际物理内存更大的地址空间。在 Linux 中，每一个用户进程都可以访问 4GB 的线性虚拟内存空间。

Linux 使用按需调页将可执行映像加载到进程的虚拟内存中。当命令执行时，可执行的命令文件被打开，同时其内容被映射到进程的虚拟内存中。这些操作是通过修改描述进程内存映像的数据结构来完成的，此过程称为内存映射。然而只有映像的起始部分被调入物理内存，其余部分仍然留在磁盘上。当映像执行时，将会产生页面错误，Linux 会决定把磁盘上哪些部分调入内存继续执行。Linux 支持三层页式存储管理策略，但考虑到 CPU 的限制，将第二层的页式管理（pmd）与第一层的页式管理（pgd）合并，因此真正发挥作用的是以页目录和页表为中心的数据结构和函数。

Linux 关于存储管理的绝大部分功能，都是利用 vm_area_struct 结构实现。利用 vm_area_struct 结构及其由它构成的链表，可以表达和维护被进程使用的虚拟空间。

此外，Linux 利用一种变型的 Buddy System 机制来管理空闲的物理页面：包括 bitmap 表和 free_area 数组。利用它们进行页面的分配和回收。

下面将通过 Linux 内核代码分析，介绍 Linux 内核对虚拟内存、虚存段、分页式内存管理、按需调页的实现机制。

1. Linux 的分页管理

LINUX 采用“按需调页”（Demand Paging）技术管理虚拟内存。标准 Linux 的虚存页表应为三级页表，依次为页目录(PGD，Page Directory)、中间页目录(PMD，Page Middle Directory)和页表（PTE，Page Table）。如下图 6-21 所示：

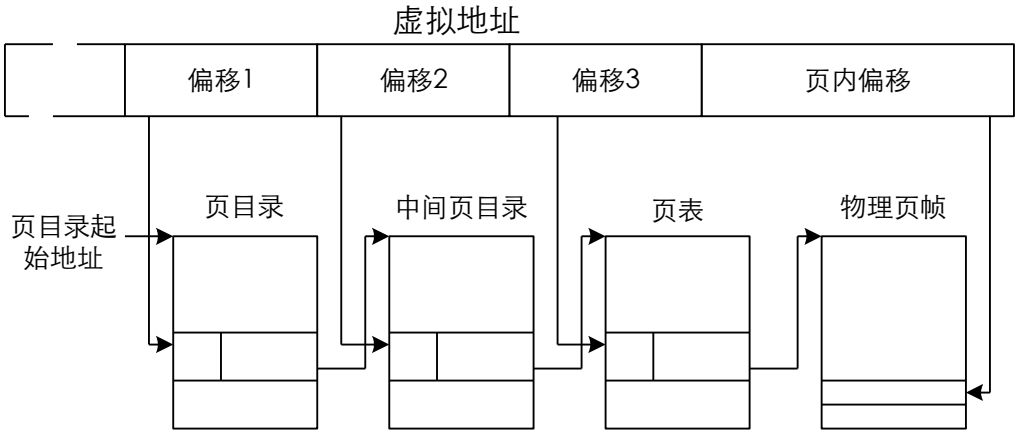

图 6-21 Linux 的三级页表结构

每一级页表通过虚拟地址的一个域来访问。图 6-21 说明虚拟地址是如何分割成多个域的。其中有三个域分别提供了在三级页表内的偏移，最后一个域提供了页内偏移。为了将虚拟地址转换成物理地址，处理器必须依次得到这几个域中包含的偏移值，还需要有页目录在物理内存中的起始地址，该地址保存在寄存器中。处理器首先根据页目录在物理内存中的

起始地址和第一个偏移值，访问页目录，得出中间页目录的起始地址；然后根据中间页目录的起始地址和第二个偏移值，访问中间页目录，得出页表的起始地址；再然后根据页表的起始地址和第三个偏移值，访问页表，得出页帧号；最后根据页帧号和页内偏移得出物理地址。

在 Intel x86 体系的微机上，Linux 的页表结构实际上为两级。其中页目录就是 PGD，页表就是 PTE，而 PMD 和 PGD 实际上是合二为一的。所有有关 PMD 的操作实际上是对 PGD 的操作。所以源代码中形如*_pgd_*()和*_pmd_*()的函数所实现的功能是一样的。有关的宏定义如下：

/include/asm-i386/pgtable-2level-defs.h

```
1 #ifndef _I386_PGTABLE_2LEVEL_DEFS_H
2 #define _I386_PGTABLE_2LEVEL_DEFS_H
3
4 #define HAVE_SHARED_KERNEL_PMD 0
5
6 /*
7  * traditional i386 two-level paging structure:
8  */
9
10 #define PGDIR_SHIFT     22
11 #define PTRS_PER_PGD    1024
12
13 /*
14  * the i386 is two-level, so we don't really have any
15  * PMD directory physically.
16  */
17
18 #define PTRS_PER_PTE    1024
19
20 #endif /* _I386_PGTABLE_2LEVEL_DEFS_H */
```

从上面的宏定义可以清楚地看到 i386 体系结构中 PMD 实际上是不存在的（#define HAVE_SHARED_KERNEL_PMD 0），实际上这一级是退化了。页目录 PGD 和页表 PTE 都含有 1024 个项。

每当启动一个新进程，Linux 都为其分配一个 task_struct 结构体，内含 ldt（local descriptor table）、tss(task state segment)、mm 等内存管理信息。其中，task_struct 结构体内含了指向 mm_struct 结构体的指针，mm_struct 结构体包含了用户进程中与内存管理有关的信息。

include/linux/sched.h

```
299 struct mm_struct {
300         struct vm_area_struct * mmap;         /* list of VMAs */
301         rb_root_t mm_rb;
302         struct vm_area_struct * mmap_cache;     /* last find_vma result */
                ……
314         pgd_t * pgd;
```

```
315     atomic_t mm_users;              /* How many users with user space? */
316     atomic_t mm_count;              /*  How many references to "struct mm_struct"
                                            *  (users count as 1) */
317     int map_count;                  /* number of VMAs */
318     struct rw_semaphore mmap_sem;
319     spinlock_t page_table_lock;         /* Protects task page tables and mm->rss */
320
321     struct list_head mmlist;            /*  List of all active mm's.  These are globally
322                                 *  together off init_mm.mmlist,
323                                 *  and are protected by mmlist_lock
324                                 */
            ……
335     unsigned long total_vm, locked_vm, shared_vm, exec_vm;
336     unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
337     unsigned long start_code, end_code, start_data, end_data;
338     unsigned long start_brk, brk, start_stack;
339     unsigned long arg_start, arg_end, env_start, env_end;
            ……
343     unsigned dumpable:2;
344     unsigned long cpu_vm_mask;
            ……
346     /* Architecture-specific MM context */
347     mm_context_t context;
348
349     /* Token based thrashing protection. */
350     unsigned long swap_token_time;
351     char recent_pagein;
352
353     /* coredumping support */
354     int core_waiters;
355     struct completion *core_startup_done, core_done;
356
357     /* aio bits */
358     rwlock_t    ioctx_list_lock;
359     struct kioctx  *ioctx_list;
360 };
```

---

300 mmap 指向 vma 段双向链表的指针。

301 mm_rb 指向 vma 段红黑树的指针。

302 mmap_cache 存储上一次对 vma 块的查找操作的结果。

314 pgd 进程页目录的起始地址。

315 mm_users 记录了目前正在使用此 mm_struct 结构的用户数。

316 mm_count 由于系统中所有进程页表的内核部分都是一样的，内核线程和普通进程相比无需 mm_struct 结构。普通进程切换到内核线程时，内核线程可以直接借用进程的的页表，无需重新加载独立的页表。内核线程用 active_mm 指针指向所借用进程的 mm_struct 结构，而每次被 active_mm 引用都要将这个 mm_count 域加 1。另外注意对于 atomic_t 类型的变量只能通过 atomic_read，atomic_inc，atomic_set 等进

行互斥性的操作。

317 map_count 此进程所使用的 VMA 块的个数。

318 mmap_sem 对 mmap 操作的互斥信号量。

319 page_table_lock 对此进程的页表操作时所需要的自旋锁。

321 mmlist task_struct 中的 active_mm 域的链表。对于普通进程，active_mm 等于 mm，对于内核线程，它等于上一次用户进程的 mm。

337 start_code、end_code 进程代码段的起始地址和结束地址。
start_data、end_data 进程数据段的起始地址和结束地址。

338 start_brk、brk 进程未初始化的数据段的起始地址和结束地址。

339 arg_start、arg_end 调用参数区的起始地址和结束地址。
env_start、env_end 进程环境区的起始地址和结束地址。

347 context 这个域存放了当前进程使用的段起始地址。

2. 虚存段（vma）的组织和管理

程序执行时，可执行映像的内容将被调入进程虚拟地址空间中,可执行映像使用的共享库也同样如此。然而可执行文件实际上并没有被调入到物理内存中，而是仅仅连接到进程的虚拟内存。当程序的其他部分运行需要引用到这部分内容时才把它们从磁盘上调入内存。

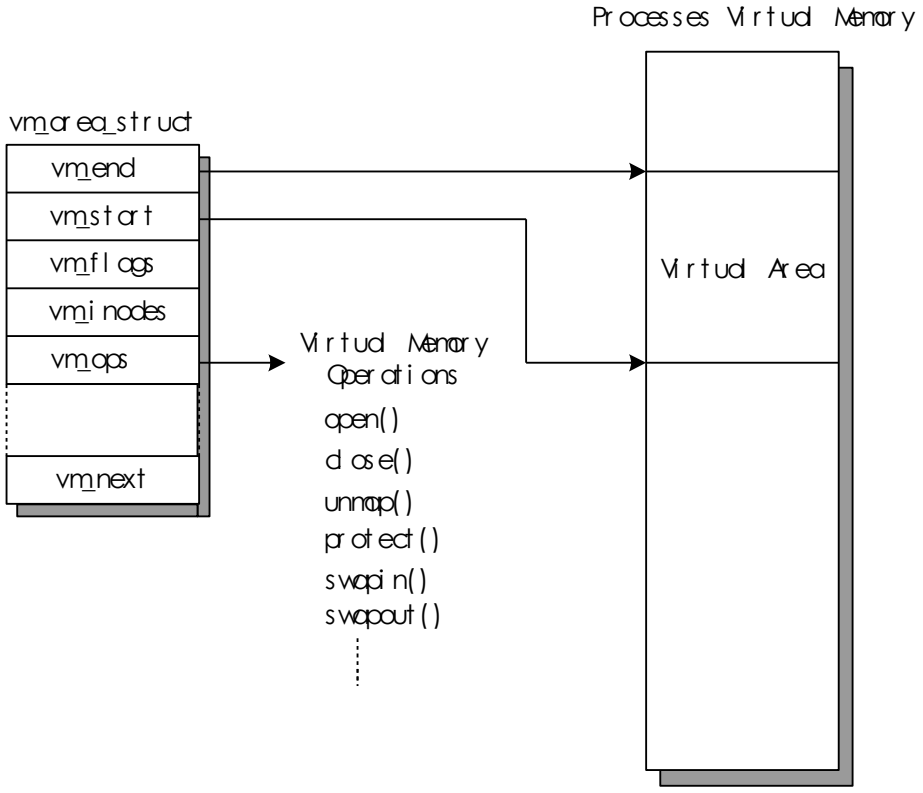

图 6-22 虚拟内存区域

每个进程的虚拟内存用一个 mm_struct 来管理。它包含一些指向 vm_area_struct 的指针。如图 6-22，每个 vm_area_struct 数据结构描述了虚拟内存段的起始与结束位置，进程对此内存区域的存取权限以及一组内存操作函数。这些函数都是 Linux 在操纵虚拟地址空间时必须用到的。当一个进程试图访问的虚拟地址不在物理内存中的时候（发生缺页中断），需要

用到一个 nopage 函数，例如当 Linux 试图将可执行映像的页面调入内存时就是这样的情况。

可执行映像映射到进程虚拟地址时将产生一组相应的 vm_area_struct 数据结构。每个 vm_area_struct 数据结构表示可执行映像的一部分：可执行代码、初始化数据(变量)、未初始化数据等等。Linux 支持许多标准的虚拟内存操作函数，创建 vm_area_struct 数据结构时有一组相应的虚拟内存操作函数与之对应。

进程可用的虚存空间共有 4GB，但这 4GB 空间并不是可以让用户态进程任意使用的，只是 0 至 3GB 之间的那一部分可以被直接使用，剩下的 1GB 空间则是属于内核的，用户态进程不能直接访问到。在创建用户进程时，内核的代码段和数据段被映射到虚拟地址 3GB 以后的虚存空间，供内核态进程使用。

有趣的是，事实上，所有进程的 3GB 至 4GB 的虚存空间的映像都是相同的，系统以此方式共享内核的代码段和数据段。

如果进程真的使用多达 4G 的虚拟空间，由此带来的管理开销巨大。例如，管理 4G 的虚拟地址空间，每个页大小为 4K，那么每个页表将占用 4M（4Byte*（4*230）/（4*210）=4*220Byte=4M Byte）物理内存。事实上目前也没有哪个进程达到如此大的规模。一个进程在运行过程中使用到的物理内存一般是不连续的，用到的虚拟地址也不是连成一片的，而是被分成几块，进程通常占用几个虚存段，分别用于代码段、数据段、堆栈段等。每个进程的所有虚存段通过指针构成链表，虚存段在此链表中的排列顺序按照它们的地址增长顺序进行。此链表的表头由 struct mm_struct 结构的成员 struct vm_area_struct * mmap 所指。为了便于理解，Linux 定义了虚存段 vma，即 virtual memory area。一个 vma 段是属于某个进程的一段连续的虚存空间，在这段虚存里的所有页面拥有一些相同的特征。例如，属于同一进程，相同的访问权限，同时被锁定（locked），同时受保护（protected）等。

vma 段由数据结构 vm_area_struct 描述如下：

include/linux/mm.h

```
59  struct vm_area_struct {
 60       struct mm_struct * vm_mm;      /* The address space we belong to. */
61       unsigned long vm_start;         /* Our start address within vm_mm. */
62       unsigned long vm_end;           /* The first byte after our end address
63                                  * within vm_mm. */
64
65       /* linked list of VM areas per task, sorted by address */
66       struct vm_area_struct *vm_next;
67
68       pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
69       unsigned long vm_flags;          /* Flags, listed below. */
70
71       rb_node  vm_rb;
                ……
98       /* Function pointers to deal with this struct. */
99       struct vm_operations_struct * vm_ops;
100
101       /* Information about our backing store: */
102       unsigned long vm_pgoff;      /* Offset (within vm_file) in PAGE_SIZE
103                                   units, *not* PAGE_CACHE_SIZE */
104       struct file * vm_file;          /* File we map to (can be NULL). */
```

```
105         void * vm_private_data;       /* was vm_pte (shared mem) */
106         unsigned long vm_truncate_count; /*
107
108 #ifndef CONFIG_MMU
109         atomic_t vm_usage;
110 #endif
111 #ifdef CONFIG_NUMA
112         Struct mempolicy *vm_policy;
113 #endif
114 };
```

---

60 vma 段指向所属进程的 mm_struct 结构的指针。

61 vma 段的起始地址 vm_start。

62 vma 段的终止地址 vm_end。

66 指向此进程 vma 链表中下一个 vma 段结构体的指针。

68 本 vma 块中页面的保护模式。pgprot_t 的定义位置在：

include/asm-i386/page.h

```
59 typedef struct { unsigned long pgprot; } pgprot_t;
```

69 本 vma 块中页面的属性标志。表明这些页面是可读、可写、可执行等。

71 用于对 vma 块进行 rb 树(Red Black Tree)操作的结构体，其定义位置在 include/linux/rbtree.h，第 100 行至第 108 行。

99 指向一个结构体的指针，该结构体中是对 vma 段进行操作的函数指针的集合。参见 include/linux/mm.h 中，第 201 行的 struct vm_operations_struct。

104 如果此 vma 段是对某个文件的映射，vm_file 为指向这个文件结构的指针。

以下是结构体 vm_operations_struct 的定义：

include/linux/mm.h

```
196 /*
197  * These are the virtual MM functions - opening of an area, closing and
198  * unmapping it (needed to keep files on disk up-to-date etc), pointer
199  * to the functions called when a no-page or a wp-page exception occurs.
200  */
201 struct vm_operations_struct {
202     void (*open)(struct vm_area_struct * area);
203     void (*close)(struct vm_area_struct * area);
204     struct page * (*nopage)(struct vm_area_struct * area, unsigned long address, int
*type);
205     int (*populate)(struct vm_area_struct * area, unsigned long address, unsigned long len,
            pgprot_t prot, unsigned long pgoff, int nonblock);
206     #ifdef CONFIG_NUMA
207     int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);
```

```
208     struct mempolicy *(*get_policy)(struct vm_area_struct *vma,
209                                     unsigned long addr);
210     #endif
211 };
```

为了提高对 vma 段查询、插入、删除操作的速度，Linux 内核为每个进程维护了一棵红黑树(Red Black Tree)，树的节点就是 vm_area_struct 类型的结构体。红黑树的节点和根节点的结构定义在：

include/linux/rbtree.h

```
100 typedef struct rb_node_s
101 {
102         struct rb_node_s * rb_parent;
103         int rb_color;
104 #define RB_RED          0
105 #define RB_BLACK        1
106         struct rb_node_s * rb_right;
107         struct rb_node_s * rb_left;
108 }
109
110 struct rb_root
111 {
112         struct rb_node * rb_node;
113 }
```

在树中，所有的 vm_area_struct 虚存段都作为树的一个节点。节点中 vm_rb 的左指针 rb_left 指向相邻的低地址虚存段，右指针 rb_right 指向相邻的高地址虚存段。

关于红黑树的基本知识请参考相关的数据结构教材。红黑树的一些操作定义在 lib/rbtree.c 中。以下是一些 rb 树的有关操作函数：

static void __rb_rotate_left(rb_node_t * node, rb_root_t * root)

static void __rb_rotate_right(rb_node_t * node, rb_root_t * root)

上面两个函数用于调整红黑树的平衡。

void rb_insert_color(rb_node_t * node, rb_root_t * root)用于向树中插入一个新节点。

static void __rb_erase_color(rb_node_t * node, rb_node_t * parent, rb_root_t * root)用于删除一个节点。

void rb_erase(rb_node_t * node, rb_root_t * root)用于删除节点后对剩余节点进行颜色调整。

3. 页面分配与回收

计算机执行的各种任务对系统中物理页面的请求十分频繁。例如当一个可执行映像被调入内存时，操作系统必须为其分配页面。当映像执行完毕和卸载时这些页面必须被释放。页面的另一个用途是存储页表等核心数据结构。

系统中所有的物理页面用包含 struct page 结构的链表 mem_map 来描述，这些结构在系

统启动时初始化。每个 struct page 描述了一个物理页面。其中与内存管理相关的重要域，例如 count，记录使用此页面的用户个数；当这个页面在多个进程之间共享时，它的值大于 1。

页面分配代码使用 free_area 数组来分配和释放页面。free_area 定义在：

include/linux/mmzone.h

```
120 struct zone {
    ……
139         /*
140         * free areas of different sizes
141         */
142       spinlock_t        lock;
143       #ifdef CONFIG_MEMORY_HOTPLUG
144         /* see spanned/present_pages for more description */
145       seqlock_t         span_seqlock;
146       #endif
147       struct  free_area           free_area[MAX_ORDER];
    ......
249 } ____cacheline_internodealigned_in_smp;
```

MAX_ORDER 默认值是 11。free_area 的定义如下：

include/linux/mmzone.h

```
25 struct free_area_struct {
26        struct list_head       free_list;
27        unsigned long        nr_free;
28 };
```

free_area 中的每个元素都包含空闲页面块的信息。数组中元素 0 维护 1 个页面大小的空闲块的链表，元素 1 维护 2 个页面大小的空闲块的链表，而接下来的元素依次维护 4 个、8 个、16 个……页面大小的空闲块的链表，也就是维护 2n（n 是非负整数）个页面大小的空闲块的链表。free_list 域表示一个队列头，它包含指向 mem_map 数组中 page 数据结构的指针，所有的空闲页面块都在此类队列中。当第 N 块空闲时，位图的第 N 位置位。

图 14-10 给出了 free_area 结构。元素 0 有 1 个空闲块（页号 0），元素 2 有 4 个页面大小的空闲块 2 个，前一个从页号 4 开始而后一个从页号 56 开始。

(1) 页面分配

Linux 使用 Buddy 算法作为内核页面级分配器，能有效地分配与回收页面块。页面分配代码每次分配包含一个或者多个物理页面的内存块，以 2n（n 是非负整数）的形式来分配。这意味着它可以分配 1 个、2 个、4 个……页面大小的块。只要系统中有足够的空闲页面来满足这个要求。内存分配代码将在 free_area 数组维护的链表中寻找一个满足要求（即不小于请求的大小）同时又尽可能小的空闲块。free_area 中的每个元素保存着一个反映特定大小的已分配或空闲页面的位图。例如，free_area 数组中元素 3 保存着一个反映大小为 4 个页面的内存块分配情况的位图。

分配算法首先搜寻满足要求的页面块。它从 free_area 数据结构的 free_list 域着手沿链表来搜索空闲块。如果在某一元素维护的链表中没有满足要求的空闲块，则继续在下一个元素

维护的链表中（该链表中的空闲块大小是上一个链表中的 2 倍）搜索。这个过程一直将持续到 free_area 所有元素维护的链表被搜索完或找到满足要求的空闲块为止。如果找到的空闲块不小于请求块的两倍，则对该空闲块进行分割以使其大小满足请求且不浪费空间。由于块大小都是 2n（n 是非负整数）页，所以分割过程十分简单，只要等分成两块即可。分割下来一块分配给请求者，而另一块作为空闲块放入上一个元素维护的空闲块队列。

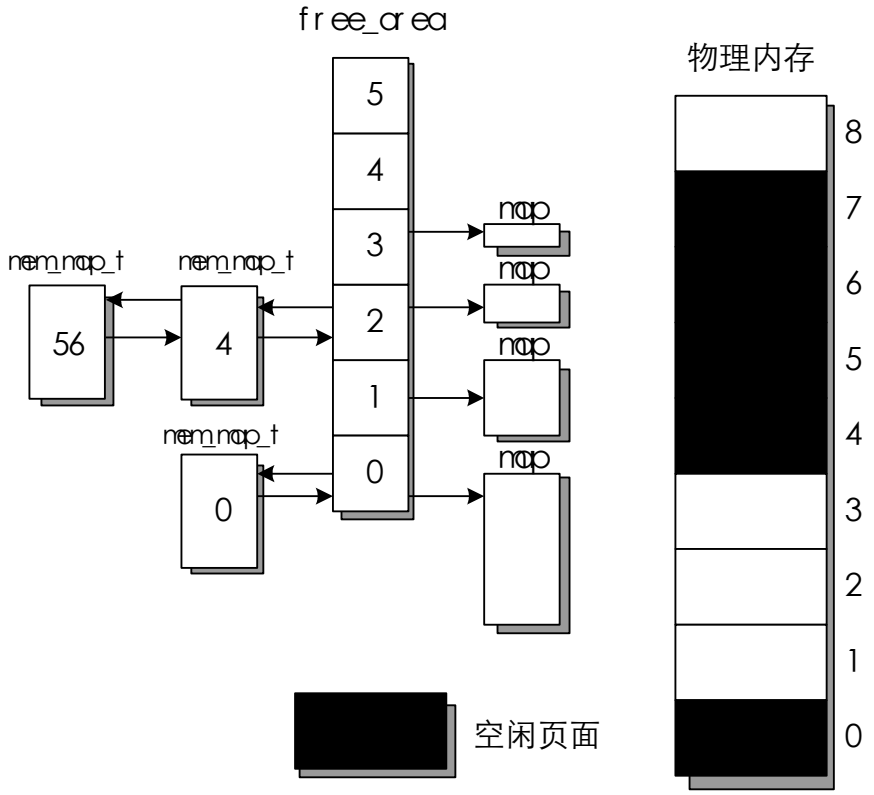

图 6-23 free_area 数据结构

在图 6-23 中，当系统中有大小为两个页面块的分配请求发出时，第一个 22 页面大小的空闲块（从页号 4 开始）将被等分成两个 21 页面大小的块。前一个，从页号 4 开始的，将分配给请求者，而后一个，从页号 6 开始，将被添加到 free_area 数组中元素 1 维护的 21 页面大小的空闲块链表中。

(2) 页面回收

将大的页面块“打碎”势必增加系统中零碎空闲页面块的数目。页面回收代码在适当时候要将这些页面结合起来形成单一大页面块。事实上页面块大小决定了页面重新组合的难易程度。

当页面块被释放时，代码将检查是否有相同大小的相邻空闲块存在。如果有，则将它们结合起来形成一个大小为原来两倍的新空闲块。每次结合完之后，算法还要检查是否可以继续合并成更大的空闲块。最佳情况是系统的空闲块将和允许分配的最大内存一样大。

在图 14-10 中，如果释放页 1，它将和空闲页 0 合并为大小为 2 个页面的空闲块，并放入 free_area 的元素 1 维护的 2 个页面大小的空闲块链表中。

(3) 按需调页

来看一下 Linux kernel 按需调页的过程：

首先由缺页中断进入 do_page_fault 函数，该函数是缺页中断服务的入口函数。该函数先查找出现缺页的虚拟内存区的 vm_area_struct 结构，如果没有找到则说明进程访问了一个非法地址，系统将向进程发送出错信号。若地址是合法的，则接着检查缺页时的访问模式是否合法。若不合法，系统将向进程发送存储访问出错的信息。通过上述两步检查之后，可以确定，此次缺页中断，的确是由于发生了缺页情况而引发的，可以进入下一步处理。

/arch/i386/mm/fault.c

```
217 /*
218 * This routine handles page faults.  It determines the address,
219 * and the problem, and then passes it off to one of the appropriate
220 * routines.
221 *
222 * error_code:
223 *bit 0 == 0 means no page found, 1 means protection fault
224 *bit 1 == 0 means read, 1 means write
225 *bit 2 == 0 means kernel, 1 means user-mode
226 */
227 fastcall void do_page_fault(struct pt_regs *regs,
228                     unsigned long error_code)
229 {
230         struct task_struct *tsk;
231         struct mm_struct *mm;
232         struct vm_area_struct * vma;
233         unsigned long address;
234         unsigned long page;
235         int write, si_code;
236
237         /* get the address */
238         address = read_cr2();
239
240         if (notify_die(DIE_PAGE_FAULT, "page fault", regs, error_code, 14,
241                                 SIGSEGV) == NOTIFY_STOP)
242             return;
243         /* It's safe to allow irq's after cr2 has been saved */
244         if (regs->eflags & X86_EFLAGS_IF)
245                 local_irq_enable();
246
247         tsk = current;
248
249         si_code = SEGV_MAPERR;
250
251         /*
252          * We fault-in kernel-space virtual memory on-demand. The
253          * 'reference' page table is init_mm.pgd.
254          *
```

```
255         * NOTE! We MUST NOT take any locks for this case. We may
256         * be in an interrupt or a critical region, and should
257         * only copy the information from the master page table,
258         * nothing more.
259         *
260         * This verifies that the fault happens in kernel space
261         * (error_code & 4) == 0, and that the fault was not a
262         * protection error (error_code & 1) == 0.
263         */
264        if (unlikely(address >= TASK_SIZE)) {
265            if (!(error_code & 5))
266                goto vmalloc_fault;
267            /*
268             * Don't take the mm semaphore here. If we fixup a prefetch
269             * fault we could otherwise deadlock.
270             */
271            goto bad_area_nosemaphore;
272        }
273
274        mm = tsk->mm;
275
276         /*
277          * If we're in an interrupt, have no user context or are running in an
278          * atomic region then we must not take the fault..
279          */
280        if (in_atomic() || !mm)
281            goto bad_area_nosemaphore;

                ..........

305        vma = find_vma(mm, address);
306        if (!vma)
307                goto bad_area;
308        if (vma->vm_start <= address)
309                goto good_area;
310        if (!(vma->vm_flags & VM_GROWSDOWN))
311                goto bad_area;
312        if (error_code & 4) {
313                /*
314                 * accessing the stack below %esp is always a bug.
315                 * The "+ 32" is there due to some instructions (like
316                 * pusha) doing post-decrement on the stack and that
317                 * doesn't show up until later..
318                 */
319                if (address + 32 < regs->esp)
320                        goto bad_area;
321        }
322        if (expand_stack(vma, address))
323                goto bad_area;
```

```
324  /*
325  * Ok, we have a good vm_area for this memory access, so
326  * we can handle it..
327  */
328 good_area:
329         info.si_code = SEGV_ACCERR;
330         write = 0;
331         switch (error_code & 3) {
332                 default:        /* 3: write, present */
333 #ifdef TEST_VERIFY_AREA
334                         if (regs->cs == KERNEL_CS)
335                                 printk("WP fault at %08lx\n", regs->eip);
336 #endif
337                         /* fall through */
338                 case 2:         /* write, not present */
339                         if (!(vma->vm_flags & VM_WRITE))
340                                 goto bad_area;
341                         write++;
342                         break;
343                 case 1:         /* read, present */
344                         goto bad_area;
345                 case 0:         /* read, not present */
346                         if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
347                                 goto bad_area;
348         }
349
350  survive:
351         /*
352         * If for any reason at all we couldn't handle the fault,
353         * make sure we exit gracefully rather than endlessly redo
354         * the fault.
355         */
356         switch (handle_mm_fault(mm, vma, address, write)) {
357                 case VM_FAULT_MINOR:
358                         tsk->min_flt++;
359                         break;
360                 case VM_FAULT_MAJOR:
361                         tsk->maj_flt++;
362                         break;
363                 case VM_FAULT_SIGBUS:
364                         goto do_sigbus;
365                 case VM_FAULT_OOM:
366                         goto out_of_memory;
367                 default:
368                         BUG();
369         }
370
371         /*
372         * Did it hit the DOS screen memory VA from vm86 mode?
```

```
373         */
374        if (regs->eflags & VM_MASK) {
375            unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
376            if (bit < 32)
377                tsk->thread.screen_bitmap |= 1 << bit;
378        }
379        up_read(&mm->mmap_sem);
380        return;
381
382 /*
383  * Something tried to access memory that isn't in our memory map..
384  * Fix it, but check if it's kernel or user first..
385  */
386 bad_area:
387        up_read(&mm->mmap_sem);
388
389 bad_area_nosemaphore:
390        /* User mode accesses just cause a SIGSEGV */
391        if (error_code & 4) {
392            /*
393             * Valid to do another page fault here because this one came
394             * from user space.
395             */
396            if (is_prefetch(regs, address, error_code))
397                return;
398
399            tsk->thread.cr2 = address;
400            /* Kernel addresses are always protection faults */
401            tsk->thread.error_code = error_code | (address >= TASK_SIZE);
402            tsk->thread.trap_no = 14;
403            force_sig_info_fault(SIGSEGV, si_code, address, tsk);
404            return;
405        }
406
407 #ifdef CONFIG_X86_F00F_BUG
408        /*
409         * Pentium F0 0F C7 C8 bug workaround.
410         */
411        if (boot_cpu_data.f00f_bug) {
412            unsigned long nr;
413
414            nr = (address - idt_descr.address) >> 3;
415
416            if (nr == 6) {
417                do_invalid_op(regs, 0);
418                return;
419            }
420        }
421 #endif
```

```
422
423 no_context:
424         /* Are we prepared to handle this kernel fault?  */
425         if (fixup_exception(regs))
426                 return;
427
428         /*
429          * Valid to do another page fault here, because if this fault
430          * had been triggered by is_prefetch fixup_exception would have
431          * handled it.
432          */
433         if (is_prefetch(regs, address, error_code))
434                 return;
435
436 /*
437  * Oops. The kernel tried to access some bad page. We'll have to
438  * terminate things with extreme prejudice.
439  */
440
441         bust_spinlocks(1);
442
443 #ifdef CONFIG_X86_PAE
444         if (error_code & 16) {
445                 pte_t *pte = lookup_address(address);
446
447                 if (pte && pte_present(*pte) && !pte_exec_kernel(*pte))
448                         printk(KERN_CRIT "kernel tried to execute NX-protected page - exploit
attempt? (uid: %d)\n", current->uid);
449         }
450 #endif
451         if (address < PAGE_SIZE)
452                 printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
453         else
454                 printk(KERN_ALERT "Unable to handle kernel paging request");
455         printk(" at virtual address %08lx\n",address);
456         printk(KERN_ALERT " printing eip:\n");
457         printk("%08lx\n", regs->eip);
458         page = read_cr3();
459         page = ((unsigned long *) __va(page))[address >> 22];
460         printk(KERN_ALERT "*pde = %08lx\n", page);
461         /*
462          * We must not directly access the pte in the highpte
463          * case, the page table might be allocated in highmem.
464          * And lets rather not kmap-atomic the pte, just in case
465          * it's allocated already.
466          */
467 #ifndef CONFIG_HIGHPTE
468         if (page & 1) {
469                 page &= PAGE_MASK;
```

```
470             address &= 0x003ff000;
471             page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
472             printk(KERN_ALERT "*pte = %08lx\n", page);
473         }
474 #endif
475         tsk->thread.cr2 = address;
476         tsk->thread.trap_no = 14;
477         tsk->thread.error_code = error_code;
478         die("Oops", regs, error_code);
479         bust_spinlocks(0);
480         do_exit(SIGKILL);
481
482 /*
483  * We ran out of memory, or some other thing happened to us that made
484  * us unable to handle the page fault gracefully.
485  */
486 out_of_memory:
487         up_read(&mm->mmap_sem);
488         if (tsk->pid == 1) {
489             yield();
490             down_read(&mm->mmap_sem);
491             goto survive;
492         }
493         printk("VM: killing process %s\n", tsk->comm);
494         if (error_code & 4)
495         do_exit(SIGKILL);
496         goto no_context;
497
498 do_sigbus:
499         up_read(&mm->mmap_sem);
500
501         /* Kernel mode? Handle exceptions or die */
502         if (!(error_code & 4))
503         goto no_context;
504
505         /* User space => ok to do another page fault */
506         if (is_prefetch(regs, address, error_code))
507             return;
508
509         tsk->thread.cr2 = address;
510         tsk->thread.error_code = error_code;
511         tsk->thread.trap_no = 14;
512         force_sig_info_fault(SIGBUS, BUS_ADRERR, address, tsk);
513         return;
514
515 vmalloc_fault:
516         {
517                 /*
518                  * Synchronize this task's top level page-table
```

```
519                 * with the 'reference' page table.
520                 *
521                 * Do _not_ use "tsk" here. We might be inside
522                 * an interrupt in the middle of a task switch..
523                 */
524                int index = pgd_index(address);
525                unsigned long pgd_paddr;
526                pgd_t *pgd, *pgd_k;
527                pud_t *pud, *pud_k;
528                pmd_t *pmd, *pmd_k;
529                pte_t *pte_k;
530
531                pgd_paddr = read_cr3();
532                pgd = index + (pgd_t *)__va(pgd_paddr);
533                pgd_k = init_mm.pgd + index;
534
535                if (!pgd_present(*pgd_k))
536                    goto no_context;
537
538                /*
539                 * set_pgd(pgd, *pgd_k); here would be useless on PAE
540                 * and redundant with the set_pmd() on non-PAE. As would
541                 * set_pud.
542                 */
543
544                pud = pud_offset(pgd, address);
545                pud_k = pud_offset(pgd_k, address);
546                if (!pud_present(*pud_k))
547                    goto no_context;
548
549                pmd = pmd_offset(pud, address);
550                pmd_k = pmd_offset(pud_k, address);
551                if (!pmd_present(*pmd_k))
552                     goto no_context;
553                set_pmd(pmd, *pmd_k);
554
555                pte_k = pte_offset_kernel(pmd_k, address);
556                if (!pte_present(*pte_k))
557                    goto no_context;
558                return;
559        }
560 }
```

---

227 do_page_fault()函数入口。regs 是 struct pt_regs 结构的指针，保存了在发生异常时的寄存器内容。error_code 是一个 32 位长整型数据，但是只有最低 3 位有效，在异常发生时，由 CPU 的控制部分根据系统当前上下文的情况，生成此 3 位数据，压入堆栈。这 3 位的含义表示：

	set（=1）	clear（=0）
0 位(1b)	保护性错误，越权访问产生异常	“存在位”为 0，要访问的页面不在 RAM 中导致异常
1 位(10b)	因为写访问导致异常（write）	因为读或者运行产生异常（read or execute）
2 位(100b)	用户态（User Mode）	内核态（Kernel Mode）

238 宏定义 read_cr2()是一组汇编指令

```
#define read_cr2() ({ \
        unsigned int __dummy; \
        __asm__ __volatile__( \
                "movl %%cr2,%0\n\t" \
                :"=r" (__dummy)); \
        __dummy; \
})
```

在发生缺页异常时，CPU 会将发生缺页异常的地址拷贝到 cr2 控制寄存器中，然后进入缺页异常的处理过程。这段汇编指令以及宏调用，将该地址从 cr2 中取出，然后存放在 address 变量中。

244-245 如果发生异常时的系统状态 EFLAGS 的中断位置位，那么在保存了 cr2 之后便可以允许中断的发生了，调用 local_irq_enable()，其底层过程调用 sti 指令，允许中断。

247 获得当前的进程描述字（process descriptor），其指针存放在 tsk 中。

264-266 如果发生异常的地址在虚拟地址空间的 TASK_SIZE 之上（也就是 PAGE_OFFSET, 0xc000 0000），并且 error_code 为 010b 的情况下，才会跳转到 vmalloc_fault 语句。error_code 表示这种情况是在内核态下，对不在 RAM 中的页面进行写操作，导致缺页异常。

274 获得当前任务的 struct mm_struct 结构成员 mm 指针。

305-311 调用 find_vma()，察看 address 是否存在于 mm 已经有的 vma 段中。如果不能找到这个 vma，那么跳转到 bad_area 语句。如果检查到 vma->vm_start 在 address 之后，说明 address 在这个 vma 的 vm_start 和 vm_end 之间，这是虚拟地址正确，但是目标地址不在 RAM 中的情况，那么跳转到 good_area 运行，一般的缺页异常都是运行到这个流程。如果该条件不满足，那么 address 就只能比 vm_start 还要小，从直观上来看，不太可能。但是因为有一些 vma 用来作为堆栈，它的空间范围变化和一般的 vma 不同：一般的 vma 是保持 vm_start 不变化，通过 vm_end 增加或者减少完成 vma 区域范围的变化的，而对于设置了 VM_GROWSDOWN 标志的 vma 是保持 vm_end 不变化，通过 vm_start 来扩张 vma 的空间的。所以运行到 310 这一行，只能是这种情况，否则就跳转到 bad_area 语句。

312-323 确定是堆栈的情况。如果是在用户态情况下发生异常，那么需要判断是否做了对比 esp 寄存器的地址还要低的地址访问操作。这种情况是不被允许的。不过 319 行将 address 增加了 32，注释中已经说明了原因：是因为有一些指令（如 push, pusha）会在用户态时访问堆栈之后才做地址的减量操作，加 32 表示允许这种情况导致的地址差异。如果这种情况也不满足，那么跳转到

bad_area。322 行的函数 expand_stack()试图通过减少 vma->vm_start 扩展 vma 的堆栈，如果失败也会跳转到 bad_area。

328 good_area 标号语句。只有在 309 行这一种情况下会跳转到这个语句。

331-348 switch 语句，根据 error_code 和 3 的与值判断处理方法。可能有如下四种情况发生：

(1) （345-347）：error_code 为 100 或 000，读 RAM 中不存在的页面。如果 vma->vm_flags 不允许读或者执行，那么跳转到 bad_area 运行，否则运行出 switch 域。

(2) （343-344）：error_code 为 101 或 001，读页面，发生保护性错误，直接跳转到 error_code。

(3) （338-342）：error_code 为 010 或 110，写 RAM 中不存在的页面。如果 vma_>vm_flags 不允许页面的写动作，那么跳转到 bad_area 运行，否则将 write 置 1，用作后面 handle_mm_fault()的参数，然后跳出 switch 域。

(4) 其它情况（332-337）：error_code 不为上面 3 种情况。error_code 组合的可能性为 8 种，除去上面已经出现的 6 种排列之外，还有 111 和 011 两种情况，即写操作，但是发生保护性错误。如果定义了 TEST_VERIFY_AREA 宏，才做 334-335 的判断语句，一般情况下都不定义这个宏，而是直接运行出 switch 域范围，到 350 行 survive 语句。

350 survive 语句。除了 good_area 按顺序运行到这里的情况之外，在 out_of_memory 标号开始的语句中，也有可能会到这里。

356-369 switch 语句用于判断 handle_mm_fault()的返回结果。函数 handle_mm_fault()用于完成调页过程。入口参数中的 write 用于标记需要调入的页面是否要用来写入。该函数的返回值有如下四种情况：

(1) minor fault，在 cache 中找到了这个页面或者指示需要在内存中申请新物理页面；

(2) major fault，从外存中调入改页面；

(3) 因为调度 I/O 的错误而无法获得页面，跳转到 do_sigbus 语句；

(4) 其它情况：都是负整数，一般情况都是无法申请物理页面，如 alloc_page()出错等，直接跳转到 out_of_memory 语句。

374-378 判断是否在 vm86 模式下访问 DOS 的 SCREEN MEMORY。这种情况下需要更新 tsk->thread.screen_bitmap 中的内容，其中保存这 SCREEN MEMORY 中的内存映像标记。

379-380 完成这种情况下的调用，释放 mm->mmap_sem 信号量，直接返回。

386 bad_area 标号的语句。在函数 do_page_fault()中如果有出错情况出现，一般都跳转到这个语句来，准备返回。在这段过程中，主要是以信号和 tsk 内部数据系统报告出错原因。

387 首先释放信号量。因为以后的操作不会涉及到 mm 数据的修改。

391-405 如果 error_code 标记为用户态的话，那么直接返回给用户进程一个 Segmentation Fault 的信号 SIGSEGV 就可以了。分别初始化 tsk->thread 的 cr2, error_code 和 trap_no 成员为异常虚拟地址、error_code 和 14（缺页异常）。然后初始化 info，调用 force_sig_info()函数将信号和相关信息发送给任务 tsk。

407-421 Pentium 的一个 bug 修正，有关情况可以参见 http://x86.ddj.com/errata/dec97/f00fbug.htm。

423 no_context 标号语句，当在中断过程中或者内核进程运行过程中出现缺页异常时就会跳转到该语句。

425-426 从 exception_table 中根据当前的 regs->eip 查找是否存在对应的 fixup 函数，如果有，那么将 eip 初始化为 fixup 函数的地址，然后返回，一般在系统调用中传递地址参数可能会出现这种异常，能事先写好 fixup 函数做好处理。如果没有查找到这种 fixup 函数，那么就可能是内核程序中的错误。

441 运行到这段语句的都是内核试图访问一个不存在的页面而产生的，内核会产生一个 oops 信息打印在终端上。内核开发程序员通过 ksymoops 和内核的符号表查找出错代码，调试内核。bust_spinlocks()就是用来解开一切用于再终端显示需要的自旋锁的函数。

451-454 如果 address 地址小于 PAGE_SIZE，被内核认为就相当于 0 地址，打印 452 行说明的信息；否则，打印“内核无法处理调页请求”的信息。

455-460 打印出当前的一些重要数据，如异常的虚拟地址，eip 值，页面地址等等。

486 out_of_memory 语句。在 handle_mm_fault()返回负整数的情况下才会运行到这段代码，这种情况下出现了无法申请页面的错误，表示内存不够用了。

488-492 如果出现异常的进程是 1 号进程，也就是 init 进程，不能杀掉这个进程，只能修改 init 进程的调度策略为 SCHED_YIELD，让它等待其它进程的内存释放，然后调用 schedule()，重新进入进程调度。之后转入 survive 语句，重新再试一次。

493-496 如果不是 init 进程，而是用户进程，就直接杀掉。如果是内核进程，跳转到 no_context 语句。

498 do_sigbus 标号语句。在申请页面过程中因为 I/O 调度错误而无法申请页面的情况，向进程发送 SIGBUS 的信号，让进程中止运行。

509-513 初始化 info 和 tsk->thread 相关成员，调用 force_sig_info()发送信号和信息给进程。然后直接返回。

515 vmalloc_fault 标号语句，在内核态写不在 RAM 中的页面才会运行到这段语句。

宏定义 handle_mm_fault()，即__handle_mm_fault()函数，先生成一个指向页表项的指针，该页表项对应的虚拟地址范围包含了导致缺页的虚拟地址，然后以生成的指针作为参数调用函数 handle_pte_fault()继续处理缺页。

/mm/memory.c

```
2368 int __handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
2369             unsigned long address, int write_access)
2370 {
2371     pgd_t *pgd;
2372     pud_t *pud;
2373     pmd_t *pmd;
2374     pte_t *pte;
```

```
2375
2376     __set_current_state(TASK_RUNNING);
2377
2378     inc_page_state(pgfault);
2379
2380     if (unlikely(is_vm_hugetlb_page(vma)))
2381         return hugetlb_fault(mm, vma, address, write_access);
2382
2383     pgd = pgd_offset(mm, address);
2384     pud = pud_alloc(mm, pgd, address);
2385     if (!pud)
2386         return VM_FAULT_OOM;
2387     pmd = pmd_alloc(mm, pud, address);
2388     if (!pmd)
2389         return VM_FAULT_OOM;
2390     pte = pte_alloc_map(mm, pmd, address);
2391     if (!pte)
2392         return VM_FAULT_OOM;
2393
2394     return handle_pte_fault(mm, vma, address, pte, pmd, write_access);
2395 }
```

2383 通过 address 得到 pgd(Page Global Directory)，即全局页目录项的指针。

2384 通过 address 和 pgd，得到 pud(Page Upper Directory)，即上层页目录项的指针。

2387 通过 address 和 pud 得出 pmd(Page Middle Directory)，即中间层页目录项。函数 pmd_alloc()得到 address 所对应的中间层页目录项的地址。由于 x86 平台上没有使用中间页目录，所以实际上只是返回给定的 pgd 指针。

2390 通过 pmd 得到一个 pte(Page Table Entry)，即得到一个与 address 地址相对应的页表项的指针。

2394 进入下一个步骤 handle_pte_fault。

handle_pte_fault 函数：

/mm/memory.c

```
2311 static inline int handle_pte_fault(struct mm_struct *mm,
2312         struct vm_area_struct *vma, unsigned long address,
2313         pte_t *pte, pmd_t *pmd, int write_access)
2314 {
2315     pte_t entry;
2316     pte_t old_entry;
2317     spinlock_t *ptl;
2318
2319     old_entry = entry = *pte;
2320     if (!pte_present(entry)) {
2321         if (pte_none(entry)) {
```

```
2322                if (!vma->vm_ops || !vma->vm_ops->nopage)
2323                    return do_anonymous_page(mm, vma, address,
2324                        pte, pmd, write_access);
2325                return do_no_page(mm, vma, address,
2326                            pte, pmd, write_access);
2327            }
2328            if (pte_file(entry))
2329                return do_file_page(mm, vma, address,
2330                            pte, pmd, write_access, entry);
2331            return do_swap_page(mm, vma, address,
2332                                pte, pmd, write_access, entry);
2333        }
2334
2335        ptl = pte_lockptr(mm, pmd);
2336        spin_lock(ptl);
2337        if (unlikely(!pte_same(*pte, entry)))
2338            goto unlock;
2339        if (write_access) {
2340            if (!pte_write(entry))
2341                return do_wp_page(mm, vma, address,
2342                            pte, pmd, ptl, entry);
2343            entry = pte_mkdirty(entry);
2344        }
2345        entry = pte_mkyoung(entry);
2346        if (!pte_same(old_entry, entry)) {
2347            ptep_set_access_flags(vma, address, pte, entry, write_access);
2348            update_mmu_cache(vma, address, entry);
2349            lazy_mmu_prot_update(entry);
2350        } else {
2351            /*
2352             * This is needed only for protection faults but the arch code
2353             * is not yet telling us if this is a protection fault or not.
2354             * This still avoids useless tlb flushes for .text page faults
2355             * with threads.
2356             */
2357            if (write_access)
2358                flush_tlb_page(vma, address);
2359        }
2360 unlock:
2361        pte_unmap_unlock(pte, ptl);
2362        return VM_FAULT_MINOR;
2363 }
```

2320 检查该页是否存在于物理内存中。

2321 判断该页是从未被映射到内存中还是已装入内存但被换出到交换空间中去了。

2325 该页从未被映射到内存，则调用 do_no_page()函数来创建一个新的页面映射。

2328 该页曾作为文件映射，被映射到内存，则调用 do_file_page()函数来创建一个新的

页面映射。
2331 该页处于交换空间中，则调用 do_swap_page()函数将它从交换空间换回。
2335 如果程序能够执行到这里，说明页表项所指明的页面已经处于物理内存中。

下面是 do_no_page 函数，该函数在缺页服务中负责建立一个新的页面映射。

/mm/memory.c

```
2150 static int do_no_page(struct mm_struct *mm, struct vm_area_struct *vma,
2151         unsigned long address, pte_t *page_table, pmd_t *pmd,
2152         int write_access)
2153 {
2154     spinlock_t *ptl;
2155     struct page *new_page;
2156     struct address_space *mapping = NULL;
2157     pte_t entry;
2158     unsigned int sequence = 0;
2159     int ret = VM_FAULT_MINOR;
2160     int anon = 0;
2161
2162     pte_unmap(page_table);
2163     BUG_ON(vma->vm_flags & VM_PFNMAP);
2164
2165     if (vma->vm_file) {
2166         mapping = vma->vm_file->f_mapping;
2167         sequence = mapping->truncate_count;
2168         smp_rmb(); /* serializes i_size against truncate_count */
2169     }
2170 retry:
2171     new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, &ret);
2172     /*
2173      * No smp_rmb is needed here as long as there's a full
2174      * spin_lock/unlock sequence inside the ->nopage callback
2175      * (for the pagecache lookup) that acts as an implicit
2176      * smp_mb() and prevents the i_size read to happen
2177      * after the next truncate_count read.
2178      */
2179
2180     /* no page was available -- either SIGBUS or OOM */
2181     if (new_page == NOPAGE_SIGBUS)
2182         return VM_FAULT_SIGBUS;
2183     if (new_page == NOPAGE_OOM)
2184         return VM_FAULT_OOM;
2185
          ......
2258 }
```

2171 调用 vma 提供的 nopage 函数，试图得到一个新页面。
2181 没得到新页面。

与 do_no_page 函数相对应的是 do_swap_page 函数，该函数负责从交换空间换入页面。

/mm/memory.c

```
1980 static int do_swap_page(struct mm_struct *mm, struct vm_area_struct *vma,
1981         unsigned long address, pte_t *page_table, pmd_t *pmd,
1982         int write_access, pte_t orig_pte)
1983 {
1984     spinlock_t *ptl;
1985     struct page *page;
1986     swp_entry_t entry;
1987     pte_t pte;
1988     int ret = VM_FAULT_MINOR;
1989
1990     if (!pte_unmap_same(mm, pmd, page_table, orig_pte))
1991         goto out;
1992
1993     entry = pte_to_swp_entry(orig_pte);
1994 again:
1995     page = lookup_swap_cache(entry);
1996     if (!page) {
1997         swapin_readahead(entry, address, vma);
1998         page = read_swap_cache_async(entry, vma, address);
      ......
2074 }
```

1993 将 pte 转换成 swp_entry。
1995 先去查看对换 cache，如果存在着这个页面，则赋给 page。
1997 如果 cache 中不存在，则开始将外存页面换入 cache 我们用一次性换入一批的方法，即 swapin_readahead，这样保证了聚簇性。
1998 从 cache 中读出一页。

二、Linux 的缺页中断处理

1．请求调页中断

Linux 进程线性地址空间里的页面不必常驻内存，页面可被交换到后援存储器，或者写入一个只读页面(COW)。Linux 采用请求调页技术来解决硬件的缺页中断异常，并且通过预约式换页策略。

Linux 为每种 CPU 体系结构提供一个 do_page_fault (struct pt_regs *regs, error_code)处理缺页中断，该函数提供了大量信息，如发生异常地址，是页面没找到还是页面保护错误，

是读异常还是写异常，来自用户空间还是内核空间。它负责确定异常类型及异常如何被体系结构无关的代码处理。

一旦异常处理程序确定异常是有效内存区域中的有效缺页中断，它将调用与体系结构无关的函数 handle_mm_fault()。如果请求页表项不存在，就分配请求的页表项，并调用 handle_pte_fault()。下面是 Linux 缺页中断处理流程：

第一步调用 pte_present 检查 PTE 标志位，确定其是否在内存中，然后调用 pte_none()检查 PTE 是否分配。如果 PTE 还没有分配的话，将调用 do_no_page()处理请求页面的分配，否则说明该页已经交换到磁盘，也是调用 do_swap_page()处理请求换页。如果换出的页面属于虚拟文件则由 do_no_page()处理。

第二步确定是否写页面。如果 PTE 写保护，就调用 do_swap_page()，因为这个页面是写时复制页面。COW 页面识别方法：页面所在 VMA 标志位可写，但相应的 PTE 确不是可写的。如果不是 COW 页面，通常将之标志为脏，因为它已经被写过了。

第三步确定页面是否已经读取及是否在内存中，但还会发生异常。这是因为在某些体系结构中没有 3 级页表，在这种情况下建立 PTE 并标志为新即可。

2．请求页面分配

第一次访问页面，首先分配页面，一般由 do_no_page()填充数据。如果父 VMA 的 vm_area_struct －>vm_ops 提供了 nopage()函数，则用它填充数据；否则调用 do_anonymous_page()匿名函数来填充数据。如果被文件或设备映射，如果是文件映射，filemap_nopage()将替代 nopage()函数，如果由虚拟文件映射而来，则 shmem_nopage()。每种设备驱动将提供不同的 nopage()函数，该函数返回 struct page 结构。

3．请求换页

将页面交换至后援存储器后，函数 do_swap_page()负责将页面读入内存，将在后面讲述。通过 PTE 的信息就足够查找到交换的页面。页面交换出去时，一般先放到交换高速缓存中。

缺页中断时如果页面在高速缓存中，则只要简单增加页面计数，然后把它放到进程页表中并计数次缺页中断发生的次数。

如果页面仅存在磁盘中，Linux 将调用 swapin_readahead()读取它及后续的若干页面。

4．页面帧回收

除了 slab 分 配器，系统中所有正在使用的页面都存放在页面高速缓存中，并由 page->lru 链接在一起。Slab 页面不存放到高速缓存中因为基于被 slab 使 用的对象对页面计数很困难。除了查找每个进程页表外没有其他方法能把 struct page 映射为 PTE，查找页表代价很大。如果页面高速缓存中存在大量的进程映射页面，系统将会遍历进程页表，通过 swap_out()函数交换出页面直 到有足够的页面空闲，而共享页会给 swap_out()带来问题。如果一个页面是共享的，同时一个交换项已经被分配，PTE 就会填写所需信息以便在交换分区里重新找到该页并将引用计数减 1。只有引用计数为 0 时该页才能被替换出去。

内存和磁盘缓存申请越来越多的页面但确无法判断如何释放进程页面，请求分页机制在进程页面缺失时申请新页面，但它却不能强制释放进程不再使用的页面。The Page Frame Reclaiming Algorithm(PFRA)页面回收算法用于从用户进程和内核 cache 中回收页面放到伙伴系统的空闲块列表中。PFRA 必须在系统空闲内存达到某个最低限度时进行页面回收，回收的对象必须是非空闲页面。

可将系统页面划分为四种：

(1) Unreclaimable 不可回收的，包括空闲页面、保留页面设置了 PG_reserved 标志、内核动态分配的页面、进程内核栈的页面、设置了 PG_locked 标志的临时锁住的页面、设置了 VM_LOCKED 标志的内存页面。

(2) Swappable 可交换的页面，用户进程空间的匿名页面(用户堆栈)、tmpfs 文件系统的映射页面（入 IPC 共享内存页面），页面存放到交换空间。

(3) Syncable 可同步的页面，入用户态地址空间的映射页面、保护磁盘数据的页面缓存的页面、块设备缓冲页、磁盘缓存的页面（入 inode cache），如果有必要的话，需同步磁盘映像上的数据。

(4) Discardable 可丢弃的页面，入内存缓存中的无用页面（slab 分配器中的页面） 、dentry cache 的页面。

PFRA 算法是基于经验而非理论的算法，它的设计原则如下：

(1) 首先释放无损坏的页面。进程不再引用的磁盘和内存缓存应该先于用户态地址空间的页面释放。

(2) 标志所有进程态进程的页面为可回收的。

(3) 多进程共享页面的回收，要先清除引用该页面的进程页表项，然后再回收。

(4) 回收“不在使用的”页面。PFRA 用 LRU 链表把进程划分为 in-use 和 unused 两种，PFRA 仅回收 unused 状态的页面。Linux 使用 PTE 中的 Accessed 比特位实现非严格的 LRU 算法。

页面回收通常在三种情况下执行：

(1) 系统可用内存比较低时进行回收(通常发生在申请内存失败)。

(2) 内核进入 suspend-to-disk 状态时进行回收。

(3) 周期性回收，内核线程周期性激活并在必要时进行页面回收。

Low on memory 回收有以下几种情形：

(1) _ _getblk( )调用的 grow_buffers( )函数分配新缓存页失败；

(2) create_empty_buffers( )调用的 alloc_page_buffers( )函数为页面分配临时的 buffer head 失败；

(3) _ _alloc_pages( )函数在给定内存区里分配一组连续的页面帧失败。

周期性回收涉及的两种内核线程：

(1) Kswapd 内核线程在内存区中检测空闲页面是否低于 pages_high 的门槛值；

(2) 预定义工作队列中的事件内核线程，PFRA 周期性调度该工作队列中的 task 回收 slab 分配器中所有空闲的 slab；

所有用户空间进程和页面缓存的页面被分为活动链表和非活动链表，统称 LRU 链表。每个区描述符中包括 active_list 和 inactive_list 两 个链表分别将这些页面链接起来。nr_active 和 nr_inactive 分别表示这两种页面数量，lru_lock 用于同步。页描述符中的 PG_lru 用于标志一个页面是否属于 LRU 链表，PG_active 用于标志页面是否属于活动链表，lru 字段用于把 LRU 中的链表串起来。活动链表和 非活动链表的页面根据最近的访问情况进行动态调整 。PG_referenced 标志就是此用途。

处理 LRU 链表的函数有：

add_page_to_active_list()、add_page_to_inactive_list()、activate_page()、lru_cache_add()、lru_cache_add_active()等，这些函数比较简单。

shrink_active_list ( )用于将页表从活动链表移到非活动链表。该函数在 shrink_zone()函数执行用户地址空间的页面回收时执行。

5．交换分区

系统可以有 MAX_SWAPFILES 的交换分区，每个分区可放在磁盘分区上或者普通文件里。每个交换区由一系列页槽组成。每个交换区有个 swap_header 结 构描述交换区版本等信息。每个交换区有若干个 swap_extent 组成，每个 swap_extent 是连续的物理区域。对于磁盘交换区只有一个 swap_extent，对于文件交换区则由多个 swap_extent 组成，因为文件并不是放在连续的磁盘块上的。mkswap 命令可以创建交换分区。

swp_type() 和 swp_offset()函数根据页槽索引和交换区号得到 type 和 offset 值，函数 swp_entry(type,offset)得到交换槽。最后一位总是清 0 表示页不在 RAM 上。槽最大 224(64G)。第一个可用槽索引为 1。槽索引不能全为 0。

一个页面可能被多个进程共用，它可能被从一个进程地址空间换出但仍然在物理内存上，因此一个页面可能被多次换出。但物理上仅第一次被换出并存储在交换区上，接下来的换出操作只是增加 swap_map 的引用计数。swap_duplicate(swp_entry_t entry)的功能正是用户尝试换出一个已经换出的页面。

6．交换缓存

多个进程同时换进一个共享匿名页时，或者一个进程换入一个正被 PFRA 换出的页时可能存在竞争条件，引入交换缓存解决这种同步问题。通过 PG_locked 标志可以保证对一个页的并发的交换操作只作用在一个页面上，从而避免竞争条件。

7. 页面回收算法描述

下图是各种情况下进行页面回收时的函数调用关系图。可以看出最终调用函数为 cache_reap()、shrink_slab()和 shrink_list()。cache_reap()用于周期性回收 slab 分配器中的无用 slab。shrink_slab()用于回收磁盘缓存的页 面。shrink_list()是页面回收的核心函数，在最新代码中该函数名改为 shrink_page_list()。下面会重点讲解。

图 中 shrink_caches() 最 新 函 数 名 为 shrink_zones() 、 shrink_cache() 最 新 函 数 名 为 shrink_inactive_list()。其他函数不变。

8. 低内存回收页面

如上图所示，当 内存分配失败时，内核调用 free_more_memory(),该函数首先调用 wakeup_bdflush( )唤醒 pdflush 内核线程触发写操作，从磁盘页面缓冲中写 1024 个 dirty 页面到磁盘上以释放包含缓冲、缓冲头和 VFS 的数据结构所占用的页表； 然后进行系统调用 sched_yield( )，以使 pdflush 线程能够有机会运行；最后该函数循环遍历系统节点，对每个节点上的低内存区(ZONE_DMA 和 ZONE_NORMAL)调用 try_to_free_pages( )函数。

try_to_free_pages(struct zone **zones, gfp_t gfp_mask) 函数的目标是通过循环调用 shrink_slab()和 shrink_zones()释放至少 32 个页帧，每次调用增加优先级参数，初始优先级是 12，最高为 0。如果循环 13 次，仍然没有释放掉 32 个页面，则 PFRA 进行内存异出保护：调用 out_of_memory()函数选择一个进程回收它所有的页面。

shrink_zones(int priority, struct zone **zones, struct scan_control *sc)函 数对 zones 列表中每个区调用 shrink_zone()函数。调用 shrink_zone()前，先用 sc->priority 的值更新 zone 描述

符中的 prev_priority，如果 zone->all_unreclaimable 字段不为 0 且优先级不是 12，则不对该区进行 页面回收。

shrink_zone(int priority, struct zone *zone, struct scan_control *sc)函数尝试回收 32 个页面。该函数循环进行 shrink_active_list()和 shrink_inactive_list 的操作以达到目标。该函数流程如下：

(1) atomic_inc(&zone->reclaim_in_progress)增加区的回收计数；

(2) 增加 zone->nr_scan_active，根据优先级，增加范围是 zone->nr_active/212 to zone->nr_active/20 。如果 zone->nr_scan_active >= 32 则赋给 nr_active 变量，同时 zone->nr_scan_active 设为 0，否则 nr_active＝0；

(3) zone->nr_scan_inactive 和 nr_inactive 做同样处理；

(4) 如果 nr_active 和 nr_inactive 不同时为空，则进行 while 循环进行 5、6 步操作：

(5) 如果 nr_active 非 0，则从 active 链表移动某些页面到 inactive 链表：

```
nr_to_scan = min(nr_active,(unsigned long)sc->swap_cluster_max);
nr_active -= nr_to_scan;
shrink_active_list(nr_to_scan, zone, sc, priority);
```

(6) 如果 nr_inactive 非 0，则回收 inactive 链表中的页面：

```
nr_to_scan = min(nr_inactive,(unsigned long)sc->swap_cluster_max);
nr_inactive -= nr_to_scan;
nr_reclaimed += shrink_inactive_list(nr_to_scan, zone, sc);
```

(7) atomic_dec(&zone->reclaim_in_progress) 减小回收计数，并返回回收页面数 nr_reclaimed；

shrink_inactive_list(unsigned long max_scan, struct zone *zone, struct scan_control *sc) 函数从区的 inactive 链表中抽取一组页面，放到临时链表中，调用 shrink_page_list()对链表中的每个页面进行回收。下面是 shrink_inactive_list()主要步骤：

(1) 调用 lru_add_drain() 将当前 CPU 上 pagevec 结构的 lru_add_pvecs 和 lru_add_active_pvecs 中的页面分别移到活动链表和非活动链表中；

(2) 获取 LRU 锁 spin_lock_irq(&zone->lru_lock);

(3) 最多扫描 max_scan 个页面，对每个页面增加使用计数，检查该页面是否正被释放到伙伴系统中，将该页面移动一个临时链表中；

(4) 从 zone->nr_inactive 中减去移到临时链表中的页面数；

(5) 增加 zone->pages_scanned 计数；

(6) 释放 LRU 锁：spin_unlock_irq(&zone->lru_lock)；

(7) 对临时链表调用 shrink_page_list(&page_list, sc)回收页面；

(8) 增加 nr_reclaimed 计数；

(9) 获取 LRU 锁 spin_lock(&zone->lru_lock)；

(10) 将 shrink_page_list(&page_list, sc) 没有回收掉的页面重新添加到 active 链表和 inactive 链表中。该函数在回收过程中可能会设置 PG_active 标志，所以也要考虑往 active 链表中添加。

(11) 如果扫描页面数 nr_scanned 小于 max_scan 则循环进行 3～10 的操作；

(12) 返回回收的页面数；

shrink_page_list(struct list_head *page_list, struct scan_control *sc)做真正的页面回收工作，该函数流程如下：

(1) 调用 cond_resched()进行条件调度；

(2) 循环遍历 page_list 中每个页面，从列表中移出该页面描述符并回收该页面，如果回收失败，则把该页面插入一个局部链表中；该步流程参见流程图。

- 调用 cond_resched() 进行条件调度；
- 从 LRU 链表中取出第一个页面并从 LRU 链表中删除；
- 如果页面被锁定，这调过该页面，该页加到临时链表中；
- 如果页面不能部分回收并且页面是进程页表的映射，这跳过该页；
- 如果进程是回写的 dirty 页面，则跳过；
- 如果页面被引用并且页面映射在使用，这跳过并激活该页面，以便后面放入 active 列表；
- 如果是匿名页面且不在交换区中，这调用 add_to_swap()为该页面分配交换区空间并把该页加到交换缓存中；
- 如果页面是进程空间映射并且页面映射地址非空，则调用 try_to_unmap()移除该页面的页表映射；
- 如果页面为 dirty 页面并且无引用、交换可写、且是 fs 文件系统映射，调用 pageout()写出该页面。

(3) 循环结束，把局部链表中的页面移回到 page_list 链表中；

(4) 返回回收页面数。

每个页面帧处理后只有三种结果：

(1) 通过调用 free_code_page()页面被释放到伙伴系统中，页面被有效回收；

(2) 页面没有回收，被重新插入到 page_list 链表中，并且认为该页面在将来可能会被再次回收，因而清除 PG _active 标志，以便在后面加入到 inactive 链表中；

(3) 页面没有回收，被重新插入到 page_list 链表中，并且认为该页面在可预见的将来不会被再次回收，因而设置 PG _active 标志，以便在后面加入到 active 链表中

回收一个匿名页面时，该页面必须添加到交换缓存中，交换区中必须为其预留一个新页槽。如果页面在某些进程的用户态地址空间 中，shrink_page_list()调用 try_to_unmap 定位所有指向该页面帧的进程 PTE 项，只有返回成功时才继续回收；如果页面是 dirty 状态，必须要写到磁盘上才能回收，这需要调用 pageout()函数，回收只有在 pageout()很快完成写操作或者不必进行写操作时才继续 进行；如果页面保护 VFS buffers，则调用 try_to_release_page()释放 buffer heads。

最后如果上面都进展顺利的话， shrink_page_list()函数检查页的引用计数：如果值正好为 2，则一个为页面缓存或交换缓存，另一个是 PFRA 自身 (shrink_inactive_page()函数中增加该值)。这种情况下，该页面可以回收，并且它不为 dirty。根据页面 PG _swapcache 标志，页面从页面缓存或交换缓存中移除；然后调用 free_code_page()。

9. 换出页面

add_to_swap(struct page * page, gfp_t gfp_mask)换出操作首先是为页面分配交换页槽，并分配交换缓存；步骤如下：

(1) get_swap_page()为换出页面预留交换槽位；

(2) 调用__add_to_swap_cache()传入槽索引、页描述符和 gfp 标志将页面加到交换缓存中，并标记为 dirty；

(3) 设置页面 PG _uptodate 和 PG_dirty 标志，以便 shrink_inactive_page()能够强制将页面写到磁盘上；

(4) 返回；

try_to_unmap(struct page *page, int migration)，换出操作第二步，在 add_to_swap 后面调用，该函数查找所有用户页表中指向该匿名页帧的页表项，并在 PTE 中设置换出标志。

Page_out()换出操作第三步将 dirty 页面写到磁盘：

(1) 检查页面缓存或交换缓存中的页，并查看该页面是否近被页面缓存或交换缓存占有；如果失败，返回 PAGE_KEEP。

(2) 检查 address_space 对象的 writepage 方法是否定义，如没有返还 PAGE_ACTIVATE；

(3) 检查当前进程是否可以发送写请求到当前映射地址空间对象对应的块设备上请求队列上。

(4) SetPageReclaim(page)设置页面回收标志；

(5) 调用 mapping->a_ops->writepage(page, &wbc)进行写操作，如果失败则清除回收标志；

(6) 如果 PageWriteback(page) 失败，页面没有写回，则清除回收标志 ClearPageReclaim(page)；

(7) 返回成功；

对于交换分区，writepage 的实现函数是 swap_writepage()，该函数流程如下:

(1) 检查是否有其他进程引用该页面，如果没有，从交换缓存中移除该页面返回 0；

(2) get_swap_bio()分配初始化 bio 描述符，该函数从交换页标志中找到交换区，然后遍历交换扩展链表找到页槽的起始磁盘分区。bio 描述符包含对一个页面的请求，并设置完成方法为 end_swap_bio_write()。

(3) set_page_writeback(page)设置页面 writeback 标志，unlock_page()该页面解锁；

(4) submit_bio(rw, bio)向块设备提交 bio 描述符进行写操作；

(5) 返回；

一旦写操作完成，end_swap_bio_write()被执行。该函数唤醒等待页面 PG_writeback 标志清除的进程，清除 PG_writeback 标志，是否 bio 描述符。

10. 换入页面

换入页面操作发生在一个进程访问被换出到磁盘上的页面时。当下列条件发生时页面出错处理程序会进行换入操作：

(1) 包含引发异常的地址的页面是一个当前进程内存区域的有效页面；

(2) 该页面不在内存中，PTE 的页面 present 表示被清 0；

(3) 与页面相关的 pte 不为 null，Dirty 位被清 0，这意味着该 pte 包含换出页的标志；

当上述条件同时满足时，hand_pte_fault()调用 do_swap_page()函数换入请求页面。

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)

该函数处理流程如下：

(1) entry = pte_to_swp_entry(orig_pte)得到交换槽位信息；

(2) page = lookup_swap_cache(entry)查看交换槽对应的页面是否存在于交换缓存中，如果是则跳到第 6 步；

(3) 调用 swapin_readahead(entry, address, vma)从交换区中读取一组页面，对每个页面调用 read_swap_cache_async()读取该页面；

(4) 对于进程访问异常的页面再次调用 read_swap_cache_async()。因为 swapin_readahead 调用可能失败，在它成功的情况下 read_swap_cache_async()发现该页面在交换缓存里，很快返回；

(5) 如果页面还是没有在交换缓存中，可能存在其他内核控制路径已经把该页面换入。比较 page_table 对应的 pte 内容与 orig_pte 是否相同，如果不同，说明页面已经换入。函数跳出返回。

(6) 如果页面在交换缓存中，调用 mark_page_accessed 并锁住该页面；

(7) pte_offset_map_lock(mm, pmd, address, &ptl)获取 page_table 对应的 pte 内容，与 orig_pte 比较，判断是否有其他内核控制路径进行换入操作；

(8) 测试 PG_uptodate 标志，如果未设置，则出错返回；

(9) 增加 mm->anon_rss 的计数；

(10) mk_pte(page, vma->vm_page_prot)创建 PTE 并设置标志，插入到进程页表中；

(11) page_add_anon_rmap()为该匿名页插入反向映射数据结构的内容；

(12) swap_free(entry)释放页槽；

(13) 检查交换缓存负载是否达到 50%，如果是，并且该页面仅被触发页面访问异常的进程占有，则从交换缓存中释放该页。

(14) 如果 write_access 标志为 1，说明是 COW 写时复制，调用 do_wp_page()拷贝一份该页面；释放页锁和页面缓存等，并返回结果。