程序 15-7 linux/lib/malloc.c

```
/*
 * malloc.c --- a general purpose kernel memory allocator for Linux.
 *
 * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
 *
 * This routine is written to be as fast as possible, so that it
 * can be called from the interrupt level.
 *
 * Limitations: maximum size of memory we can allocate using this routine
 *      is 4k, the size of a page in Linux.
 *
 * The general game plan is that each page (called a bucket) will only hold
 * objects of a given size.  When all of the object on a page are released,
 * the page can be returned to the general free pool.  When malloc() is
 * called, it looks for the smallest bucket size which will fulfill its
 * request, and allocate a piece of memory from that bucket pool.
 *
 * Each bucket has as its control block a bucket descriptor which keeps
 * track of how many objects are in use on that page, and the free list
 * for that page.  Like the buckets themselves, bucket descriptors are
 * stored on pages requested from get_free_page().  However, unlike buckets,
 * pages devoted to bucket descriptor pages are never released back to the
 * system.  Fortunately, a system should probably only need 1 or 2 bucket
 * descriptor pages, since a page can hold 256 bucket descriptors (which
 * corresponds to 1 megabyte worth of bucket pages.)  If the kernel is using
 * that much allocated memory, it's probably doing something wrong.  :-)
 *
 * Note: malloc() and free() both call get_free_page() and free_page()
 *      in sections of code where interrupts are turned off, to allow
 *      malloc() and free() to be safely called from an interrupt routine.
 *      (We will probably need this functionality when networking code,
 *      particularily things like NFS, is added to Linux.)  However, this
 *      presumes that get_free_page() and free_page() are interrupt-level
 *      safe, which they may not be once paging is added.  If this is the
 *      case, we will need to modify malloc() to keep a few unused pages
 *      "pre-allocated" so that it can safely draw upon those pages if
 *      it is called from an interrupt routine.
 *
 *      Another concern is that get_free_page() should not sleep; if it
 *      does, the code is carefully ordered so as to avoid any race
 *      conditions.  The catch is that if malloc() is called re-entrantly,
 *      there is a chance that unecessary pages will be grabbed from the
 *      system.  Except for the pages for the bucket descriptor page, the
 *      extra pages will eventually get released back to the system, though,
 *      so it isn't all that bad.
 */

/*
 *     malloc.c - Linux 的通用内核内存分配函数。
 *
 * 由 Theodore Ts'o 编制 (tytso@mit.edu), 11/29/91
 *
```

```
 * 该函数被编写成尽可能地快，从而可以从中断层调用此函数。
 *
 * 限制：使用该函数一次所能分配的最大内存是 4k，也即 Linux 中内存页面的大小。
 *
 * 编写该函数所遵循的一般规则是每页(被称为一个存储桶)仅分配所要容纳对象的大小。
 * 当一页上的所有对象都释放后，该页就可以返回通用空闲内存池。当 malloc()被调用
 * 时，它会寻找满足要求的最小的存储桶，并从该存储桶中分配一块内存。
 *
 * 每个存储桶都有一个作为其控制用的存储桶描述符，其中记录了页面上有多少对象正被
 * 使用以及该页上空闲内存的列表。就象存储桶自身一样，存储桶描述符也是存储在使用
 * get_free_page()申请到的页面上的，但是与存储桶不同的是，桶描述符所占用的页面
 * 将不再会释放给系统。幸运的是一个系统大约只需要 1 到 2 页的桶描述符页面，因为一
 * 个页面可以存放 256 个桶描述符(对应 1MB 内存的存储桶页面)。如果系统为桶描述符分
 * 配了许多内存，那么肯定系统什么地方出了问题☺。
 *
 * 注意！ malloc()和 free()两者关闭了中断的代码部分都调用了 get_free_page()和
 *       free_page()函数，以使 malloc()和 free()可以安全地被从中断程序中调用
 *       (当网络代码，尤其是 NFS 等被加入到 Linux 中时就可能需要这种功能)。但前
 *       提是假设 get_free_page()和 free_page()是可以安全地在中断级程序中使用的，
 *       这在一旦加入了分页处理之后就可能不是安全的。如果真是这种情况，那么我们
 *       就需要修改 malloc()来“预先分配”几页不用的内存，如果 malloc()和 free()
 *       被从中断程序中调用时就可以安全地使用这些页面。
 *
 *       另外需要考虑到的是 get_free_page()不应该睡眠；如果会睡眠的话，则为了防止
 *       任何竞争条件，代码需要仔细地安排顺序。 关键在于如果 malloc()是可以重入地
 *       被调用的话，那么就会存在不必要的页面被从系统中取走的机会。除了用于桶描述
 *       符的页面，这些额外的页面最终会释放给系统，所以并不是象想象的那样不好。
 */

#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

// 存储桶描述符结构。
struct bucket_desc {    /* 16 bytes */
        void                    *page;          // 该桶描述符对应的内存页面指针。
        struct bucket_desc      *next;          // 下一个描述符指针。
        void                    *freeptr;       // 指向本桶中空闲内存位置的指针。
        unsigned short          refcnt;         // 引用计数。
        unsigned short          bucket_size;    // 本描述符对应存储桶的大小。
};

// 存储桶描述符目录结构。
struct _bucket_dir {    /* 8 bytes */
        int                     size;       // 该存储桶的大小(字节数)。
        struct bucket_desc      *chain;     // 该存储桶目录项的桶描述符链表指针。
};

/*
 * The following is the where we store a pointer to the first bucket
 * descriptor for a given size.
 *
 * If it turns out that the Linux kernel allocates a lot of objects of a
```

```
 * specific size, then we may want to add that specific size to this list,
 * since that will allow the memory to be allocated more efficiently.
 * However, since an entire page must be dedicated to each specific size
 * on this list, some amount of temperance must be exercised here.
 *
 * Note that this list *must* be kept in order.
 */
/*
 * 下面是我们存放第一个给定大小存储桶描述符指针的地方。
 *
 * 如果 Linux 内核分配了许多指定大小的对象，那么我们就希望将该指定的大小加到
 * 该列表(链表)中，因为这样可以使内存的分配更有效。但是，因为一页完整内存页面
 * 必须用于列表中指定大小的所有对象，所以需要做总数方面的测试操作。
 */
// 存储桶目录列表(数组)。
struct _bucket_dir bucket_dir[] = {
        { 16,   (struct bucket_desc *) 0},      // 16 字节长度的内存块。
        { 32,   (struct bucket_desc *) 0},      // 32 字节长度的内存块。
        { 64,   (struct bucket_desc *) 0},      // 64 字节长度的内存块。
        { 128,  (struct bucket_desc *) 0},      // 128 字节长度的内存块。
        { 256,  (struct bucket_desc *) 0},      // 256 字节长度的内存块。
        { 512,  (struct bucket_desc *) 0},      // 512 字节长度的内存块。
        { 1024, (struct bucket_desc *) 0},      // 1024 字节长度的内存块。
        { 2048, (struct bucket_desc *) 0},      // 2048 字节长度的内存块。
        { 4096, (struct bucket_desc *) 0},      // 4096 字节(1 页)内存。
        { 0,    (struct bucket_desc *) 0}};     /* End of list marker */

/*
 * This contains a linked list of free bucket descriptor blocks
 */
/*
 * 下面是含有空闲桶描述符内存块的链表。
 */
struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0;

/*
 * This routine initializes a bucket description page.
 */
/*
 * 下面的子程序用于初始化一页桶描述符页面。
 */
//// 初始化桶描述符。
// 建立空闲桶描述符链表，并让 free_bucket_desc 指向第一个空闲桶描述符。
static inline void init_bucket_desc()
{
        struct bucket_desc *bdesc, *first;
        int     i;

// 申请一页内存，用于存放桶描述符。如果失败，则显示初始化桶描述符时内存不够出错信息，死机。
        first = bdesc = (struct bucket_desc *) get_free_page();
        if (!bdesc)
                panic("Out of memory in init_bucket_desc()");
// 首先计算一页内存中可存放的桶描述符数量，然后对其建立单向连接指针。
```

```
105         for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {
106                 bdesc->next = bdesc+1;
107                 bdesc++;
108         }
109         /*
110          * This is done last, to avoid race conditions in case
111          * get_free_page() sleeps and this routine gets called again....
112          */
            /*
             * 这是在最后处理的，目的是为了避免在 get_free_page()睡眠时该子程序又被
             * 调用而引起的竞争条件。
             */
    // 将空闲桶描述符指针 free_bucket_desc 加入链表中。
113         bdesc->next = free_bucket_desc;
114         free_bucket_desc = first;
115 }
116
    //// 分配动态内存函数。
    // 参数：len - 请求的内存块长度。
    // 返回：指向被分配内存的指针。如果失败则返回 NULL。
117 void *malloc(unsigned int len)
118 {
119         struct _bucket_dir      *bdir;
120         struct bucket_desc      *bdesc;
121         void                    *retval;
122
123         /*
124          * First we search the bucket_dir to find the right bucket change
125          * for this request.
126          */
            /*
             * 首先我们搜索存储桶目录 bucket_dir 来寻找适合请求的桶大小。
             */
    // 搜索存储桶目录，寻找适合申请内存块大小的桶描述符链表。如果目录项的桶字节数大于请求的字节
    // 数，就找到了对应的桶目录项。
127         for (bdir = bucket_dir; bdir->size; bdir++)
128                 if (bdir->size >= len)
129                         break;
    // 如果搜索完整个目录都没有找到合适大小的目录项，则表明所请求的内存块大小太大，超出了该
    // 程序的分配限制(最长为 1 个页面)。于是显示出错信息，死机。
130         if (!bdir->size) {
131                 printk("malloc called with impossibly large argument (%d)\n",
132                         len);
133                 panic("malloc: bad arg");
134         }
135         /*
136          * Now we search for a bucket descriptor which has free space
137          */
            /*
             * 现在我们来搜索具有空闲空间的桶描述符。
             */
138         cli();  /* Avoid race conditions */  /* 为了避免出现竞争条件，首先关中断 */
    // 搜索对应桶目录项中描述符链表，查找具有空闲空间的桶描述符。如果桶描述符的空闲内存指针
```

```
    // freeptr 不为空，则表示找到了相应的桶描述符。
139         for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
140                 if (bdesc->freeptr)
141                         break;
142         /*
143          * If we didn't find a bucket with free space, then we'll
144          * allocate a new one.
145          */
            /*
             * 如果没有找到具有空闲空间的桶描述符，那么我们就要新建立一个该目录项的描述符。
             */
146         if (!bdesc) {
147                 char            *cp;
148                 int             i;
149
    // 若 free_bucket_desc 还为空时，表示第一次调用该程序，或者链表中所有空桶描述符都已用完。
    // 此时就需要申请一个页面并在其上建立并初始化空闲描述符链表。free_bucket_desc 会指向第一
    // 个空闲桶描述符。
150                 if (!free_bucket_desc)
151                         init_bucket_desc();
    // 取 free_bucket_desc 指向的空闲桶描述符，并让 free_bucket_desc 指向下一个空闲桶描述符。
152                 bdesc = free_bucket_desc;
153                 free_bucket_desc = bdesc->next;
    // 初始化该新的桶描述符。令其引用数量等于 0；桶的大小等于对应桶目录的大小；申请一内存页面，
    // 让描述符的页面指针 page 指向该页面；空闲内存指针也指向该页开头，因为此时全为空闲。
154                 bdesc->refcnt = 0;
155                 bdesc->bucket_size = bdir->size;
156                 bdesc->page = bdesc->freeptr = (void *) cp = get_free_page();
    // 如果申请内存页面操作失败，则显示出错信息，死机。
157                 if (!cp)
158                         panic("Out of memory in kernel malloc()");
159                 /* Set up the chain of free objects */
                    /* 在该页空闲内存中建立空闲对象链表 */
    // 以该桶目录项指定的桶大小为对象长度，对该页内存进行划分，并使每个对象的开始 4 字节设置
    // 成指向下一对象的指针。
160                 for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
161                         *((char **) cp) = cp + bdir->size;
162                         cp += bdir->size;
163                 }
    // 最后一个对象开始处的指针设置为 0(NULL)。
    // 然后让该桶描述符的下一描述符指针字段指向对应桶目录项指针 chain 所指的描述符，而桶目录的
    // chain 指向该桶描述符，也即将该描述符插入到描述符链链头处。
164                 *((char **) cp) = 0;
165                 bdesc->next = bdir->chain; /* OK, link it in! */   /* OK，将其链入！*/
166                 bdir->chain = bdesc;
167         }
    // 返回指针即等于该描述符对应页面的当前空闲指针。然后调整该空闲空间指针指向下一个空闲对象，
    // 并使描述符中对应页面中对象引用计数增 1。
168         retval = (void *) bdesc->freeptr;
169         bdesc->freeptr = *((void **) retval);
170         bdesc->refcnt++;
    // 最后开放中断，并返回指向空闲内存对象的指针。
171         sti();  /* OK, we're safe again */   /* OK，现在我们又安全了*/
```

```
172         return(retval);
173 }
174
175 /*
176  * Here is the free routine.  If you know the size of the object that you
177  * are freeing, then free_s() will use that information to speed up the
178  * search for the bucket descriptor.
179  *
180  * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181  */
    /*
     * 下面是释放子程序。如果你知道释放对象的大小，则 free_s()将使用该信息加速
     * 搜寻对应桶描述符的速度。
     *
     * 我们将定义一个宏，使得"free(x)"成为"free_s(x, 0)"。
     */
    //// 释放存储桶对象。
    // 参数：obj - 对应对象指针；size - 大小。
182 void free_s(void *obj, int size)
183 {
184         void            *page;
185         struct _bucket_dir      *bdir;
186         struct bucket_desc      *bdesc, *prev;
187
188         /* Calculate what page this object lives in */
            /* 计算该对象所在的页面 */
189         page = (void *)  ((unsigned long) obj & 0xfffff000);
190         /* Now search the buckets looking for that page */
            /* 现在搜索存储桶目录项所链接的桶描述符，寻找该页面 */
    //
191         for (bdir = bucket_dir; bdir->size; bdir++) {
192                 prev = 0;
193                 /* If size is zero then this conditional is always false */
                    /* 如果参数 size 是 0，则下面条件肯定是 false */
194                 if (bdir->size < size)
195                         continue;
    // 搜索对应目录项中链接的所有描述符，查找对应页面。如果某描述符页面指针等于 page 则表示找到
    // 了相应的描述符，跳转到 found。如果描述符不含有对应 page，则让描述符指针 prev 指向该描述符。
196                 for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
197                         if (bdesc->page == page)
198                                 goto found;
199                         prev = bdesc;
200                 }
201         }
    // 若搜索了对应目录项的所有描述符都没有找到指定的页面，则显示出错信息，死机。
202         panic("Bad address passed to kernel free_s()");
203 found:
    // 找到对应的桶描述符后，首先关中断。然后将该对象内存块链入空闲块对象链表中，并使该描述符
    // 的对象引用计数减 1。
204         cli(); /* To avoid race conditions */   /* 为了避免竞争条件 */
205         *((void **)obj) = bdesc->freeptr;
206         bdesc->freeptr = obj;
207         bdesc->refcnt--;
```

```
    // 如果引用计数已等于 0，则我们就可以释放对应的内存页面和该桶描述符。
208         if (bdesc->refcnt == 0) {
209                 /*
210                  * We need to make sure that prev is still accurate.  It
211                  * may not be, if someone rudely interrupted us....
212                  */
                    /*
                     * 我们需要确信 prev 仍然是正确的，若某程序粗鲁地中断了我们
                     * 就有可能不是了。
                     */
    // 如果 prev 已经不是搜索到的描述符的前一个描述符，则重新搜索当前描述符的前一个描述符。
213                 if ((prev && (prev->next != bdesc)) ||
214                     (!prev && (bdir->chain != bdesc)))
215                         for (prev = bdir->chain; prev; prev = prev->next)
216                                 if (prev->next == bdesc)
217                                         break;
    // 如果找到该前一个描述符，则从描述符链中删除当前描述符。
218                 if (prev)
219                         prev->next = bdesc->next;
    // 如果 prev==NULL，则说明当前一个描述符是该目录项首个描述符，也即目录项中 chain 应该直接
    // 指向当前描述符 bdesc，否则表示链表有问题，则显示出错信息，死机。因此，为了将当前描述符
    // 从链表中删除，应该让 chain 指向下一个描述符。
220                 else {
221                         if (bdir->chain != bdesc)
222                                 panic("malloc bucket chains corrupted");
223                         bdir->chain = bdesc->next;
224                 }
    // 释放当前描述符所操作的内存页面，并将该描述符插入空闲描述符链表开始处。
225                 free_page((unsigned long) bdesc->page);
226                 bdesc->next = free_bucket_desc;
227                 free_bucket_desc = bdesc;
228         }
    // 开中断，返回。
229         sti();
230         return;
231 }
232
233
```