



操作系统实验报告

实验（四）内存管理

姓 名	熊恪峥
学 号	22920202204622
日 期	2023年6月5日
学 院	信息学院
课程名称	操作系统

实验（四）内存管理

目录

1	实验内容	1
2	实验目的	1
3	使用kmalloc分配内存	1
3.1	kmalloc的分配上限	2
3.2	分配区域	2
4	使用vmalloc分配内存	2
4.1	分配区域	3
5	阅读并理解首次适应算法的实现	3
5.1	ff_malloc	3
5.2	free	4
5.3	calloc	4
5.4	test.c	4
6	实现最佳适应算法	4
6.1	对比	4
7	实验总结	5

1 实验内容

本实验利用内核函数 `kmalloc()`、`vmalloc()` 实现内存的分配，并要求学生根据提示实现基于最佳适应算法的 `bf_malloc` 内存分配器。

2 实验目的

1. 学习掌握 `kmalloc()` 和 `vmalloc()` 分配内存的差异；
2. 加深对首次适应算法和最佳适应算法的理解；
3. 锻炼编写内核模块的能力。

3 使用 `kmalloc` 分配内存

`kmalloc` 具有两个参数，第一个参数是要分配的内存大小，第二个参数是分配内存的类型。在本次实验中，我们使用 `GFP_KERNEL` 来在内核空间分配内存。该函数失败时会返回 `NULL`，因此我们需要检查返回值是否为 `NULL`，如果是则说明分配失败，则输出错误信息 “Failed to allocate `kmalloccmem1/ kmalloccmem2/ kmalloccmem3 / kmalloccmem4!`”。

根据以上思路，编写代码 2。代码 2 将在模块加载时执行，遇到错误输出错误消息，否则输出分配内存

代码 1 使用 `kmalloc` 分配内存

```
static int __init kmalloc_module_init(void)
{
    printk(KERN_INFO "kmalloc module loaded\n");
    kmalloccmem1 = kmalloc(1024, GFP_KERNEL);
    if (!kmalloccmem1)
        printk(KERN_INFO "Failed to allocate kmalloccmem1!\n");
    else
        printk(KERN_INFO "kmalloccmem1: %p\n", kmalloccmem1);

    kmalloccmem2 = kmalloc(8192, GFP_KERNEL);
    if (!kmalloccmem2)
        printk(KERN_INFO "Failed to allocate kmalloccmem2!\n");
    else
        printk(KERN_INFO "kmalloccmem2: %p\n", kmalloccmem2);

    kmalloccmem3 = kmalloc(10485760, GFP_KERNEL);
    if (!kmalloccmem3)
        printk(KERN_INFO "Failed to allocate kmalloccmem3!\n");
    else
        printk(KERN_INFO "kmalloccmem3: %p\n", kmalloccmem3);

    kmalloccmem4 = kmalloc(10500000, GFP_KERNEL);
    if (!kmalloccmem4)
        printk(KERN_INFO "Failed to allocate kmalloccmem4!\n");
    else
        printk(KERN_INFO "kmalloccmem4: %p\n", kmalloccmem4);

    return 0;
}
```

的大小。代码的运行结果如图 1 所示。

图 1: 使用kmalloc分配内存



可见，前两个请求成功了，而后两个请求失败了。

3.1 kmalloc的分配上限

常识告诉我们，由于kmalloc分配时需要保证有连续的物理内存页一定空闲，所以一定具有上限。同时，查阅Linux内核文档，可以发现如下内容：

The maximal size of a chunk that can be allocated with kmalloc is limited. The actual limit depends on the hardware and the kernel configuration, but it is a good practice to use kmalloc for objects smaller than page size.

因此，确实具有上限。

首先，为了找到上限，可以修改代码 2。通过不断修改kmallocmem3和kmallocmem4的大小，保证前者请求成功，后者请求失败，并逐步减半区间长度，可以快速逼近得到上限。根据以上过程，得到了上限是4MB。

同时，该值也可以从内核源码中得到依据：查看include/linux/slab.h文件，其中定义宏KMALLOC_MAX_SIZE为(1UL << KMALLOC_SHIFT_HIGH)，而宏 KMALLOC_SHIFT_HIGH又定义为：

```
#define KMALLOC_SHIFT_HIGH      ((MAX_ORDER + PAGE_SHIFT - 1) <= 25 ? \
                                (MAX_ORDER + PAGE_SHIFT - 1) : 25)
```

在x86架构下，MAX_ORDER为11，PAGE_SHIFT为12，因此可以计算得出：

$$\text{KMALLOC_MAX_SIZE} = 2^{11+12-1} = 4\text{MB} \quad (1)$$

同时，这也说明了在任何架构下，KMALLOC_MAX_SIZE都不能超过32MB。因为宏KMALLOC_SHIFT_HIGH 保证了该值不超过 $2^{25}\text{B} = 32\text{MB}$ 。

3.2 分配区域

由于指定了分配内存的类型为GFP_KERNEL，因此分配的内存位于内核空间，分配的物理地址会处于896MB之下（32位架构），然后根据Linux内核的内存管理机制，会将这些物理地址映射到2GB之上。这符合图 ??中的结果。

4 使用vmalloc分配内存

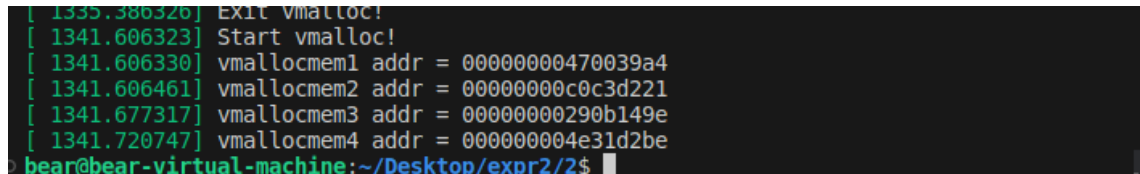
vmalloc可以分配虚拟连续的内存，但是不能保证物理连续。这降低了对物理内存的要求。编写代码非常简单，只需将代码 2中的kmalloc替换为vmalloc即可。例如分配vmallocmem1时，代码如下：

运行结果如图 2所示。可见，vmalloc可以成功分配远大于上述kmalloc上限的内存。因为真正对物理内存页的分配被推迟到了第一次访问时。这既增加了分配的灵活性，也会带来无法用于DMA等问题。

代码 2 使用kmalloc分配内存

```
printk(KERN_INFO "Start vmalloc!\n");
vmallocmem1 = vmalloc(8192);
if (!vmallocmem1)
    printk(KERN_INFO "Failed to allocate vmallocmem1!\n");
else
    printk(KERN_INFO "vmallocmem1 addr = %p\n", vmallocmem1);
```

图 2: 使用vmalloc分配内存

A terminal window showing the output of a program that uses vmalloc. The output includes timestamps and addresses for vmallocmem1 through vmallocmem4, all showing addresses near 0x00000000. The prompt is bear@bear-virtual-machine:~/Desktop/expr2/2\$.

```
[ 1335.386326] Exit vmalloc!
[ 1341.606323] Start vmalloc!
[ 1341.606330] vmallocmem1 addr = 00000000470039a4
[ 1341.606461] vmallocmem2 addr = 00000000c0c3d221
[ 1341.677317] vmallocmem3 addr = 00000000290b149e
[ 1341.720747] vmallocmem4 addr = 000000004e31d2be
bear@bear-virtual-machine:~/Desktop/expr2/2$
```

4.1 分配区域

vmalloc会将内存地址分配到单独的一个区域，这个区域由宏VMALLOC_START和VMALLOC_END指定。在i386架构（32位）下会优先选择ZONE_HIGHMEM，否则会选择ZONE_NORMAL，在图 2中，由于默认使用了32位编译，所以符合这种情况，分配的地址在0xc0000000附近。

在64位x86架构下，由于地址空间大大扩展了，因此分配的区域也更大，该区域处在地址的Canonical Bits为1的区域，处在内存空间较高的位置，根据文档¹，该区域的起始地址为ffffc90000000000，结束地址为ffffe8fffffffffff。

5 阅读并理解首次适应算法的实现

首先，内存分配器使用结构header进行管理：

```
union header
{
    struct
    {
        union header *next;
        unsigned len;
    } meta;
    long x; /* Presence forces alignment of headers in memory. */
};
```

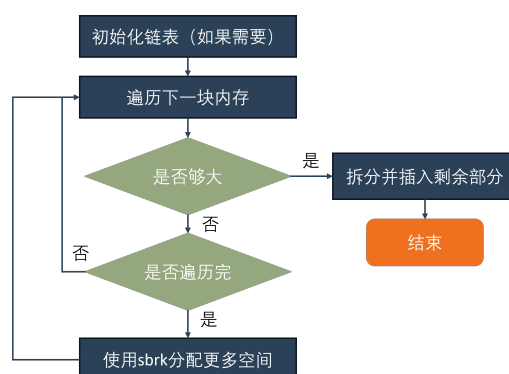
它记录了每一个空闲块的长度和下一块的位置。能够把内存块通过链表的方式进行记录和管理。具体的链表头为list指针。

5.1 ff_malloc

ff_malloc为内存分配的具体逻辑，用于分配size大小的内存。同时，如果未初始化，也会初始化链表和相应的指针。初始化完成后，就会查找第一个适配的块并进行拆分，如果没有这样的块就再调用sbrk进行分配。具体流程如图 3。

¹https://elixir.bootlin.com/linux/v5.0/source/Documentation/x86/x86_64/mm.txt

图 3: 首次适配算法



5.2 free

该函数会遍历链表，找到内存地址合适的位置，将块插入。此外，还会合并相邻的空闲块。

5.3 calloc

calloc分配指定大小的内存并置为0，这是通过调用ff_malloc并使用memset赋值实现的。

5.4 test.c

该程序使用并测试了首次适配的分配和释放功能，它进行三轮测试，每轮测试都是连续分配逐渐增大大小的内存块并写入数据，并输出相应信息。最后，对这些内存块进行释放。

6 实现最佳适应算法

最佳适配算法中free、calloc的实现都与首次适配相同，只需要修改具体的分配逻辑。在分配时遍历整个链表，选择与所要求尺寸差值最小的块进行分配，如代码 3。当找到这样的块时，采用和首次适配相同的拆分策略，并返回相应的内存。如果没有找到这样的块，则使用sbrk系统调用分配内存，将新的块插入链表中，然后再次调用bf_malloc进行分配。由于这次一定能找到合适的块，因此不会再次调用bf_malloc。因此，这种做法不会带来过多的额外开销。如代码 4。

6.1 对比

为了对比两种不同的分配算法，我进一步修改代码，在每次分配成功以后输出整个空闲链表的内容，其中0代表头节点，如图 4。首先，测试函数会进行10次内存分配，然后释放一部分块，此时根据输出，有三个

图 4: 使用不同的策略分配内存

```

# bear@bear-virtual-machine:~/Desktop/expr2/3$ ./test
freelist:{0 1022 };
freelist:{0 1020 };
freelist:{0 1018 };
freelist:{0 1016 };
freelist:{0 1013 };
freelist:{0 1010 };
freelist:{0 1007 };
freelist:{0 1004 };
freelist:{0 1000 };
freelist:{0 996 };
freelist:{996 3 5 0 };
freelist:{3 1 0 996 };
freelist:{0 991 3 1 };
freelist:{0 986 3 1 };
freelist:{0 981 3 1 };

# bear@bear-virtual-machine:~/Desktop/expr2/3$ ./test
freelist:{0 1022 };
freelist:{0 1020 };
freelist:{0 1018 };
freelist:{0 1016 };
freelist:{0 1013 };
freelist:{0 1010 };
freelist:{0 1007 };
freelist:{0 1004 };
freelist:{0 1000 };
freelist:{0 996 };
freelist:{7 1 0 996 };
freelist:{996 3 1 0 };
freelist:{0 991 3 1 };
freelist:{0 986 3 1 };
freelist:{0 981 3 1 };
  
```

空闲的内存块，它们的长度分别是7,5,996。

代码 3 查找最接近的内存块

```

while (1)
{
    if (p->meta.len >= true_size)
    {
        if (best_fit == NULL || p->meta.len < best_fit->meta.len)
        {
            best_fit = p;
            best_fit_prev = prev;
            if (p->meta.len == true_size)
            {
                break;
            }
        }
    }
    /* If we reach the beginning of the list, no satisfactory fragment
     * was found, so we have to request a new one. */
    if (p == first)
    {
        break;
    }
    prev = p;
    p = p->meta.next;
}

```

此后的分配中，首次适配和最佳适配将表现出差别：接下来，会分配大小为4的块。此时，首次适配找到块7，拆分后返回。这是，剩下的三块为3,5,996。而最佳适配策略会找出与请求尺寸大小最接近的块，则会找到块5，进行拆分后剩余的内存块为7,1,996。

一方面，最佳适配会遍历整个链表，查找的时间会更长。假设有 n 个块，每个块成为最佳适配的概率则为 $\frac{1}{n}$ 。那么最佳适配的查找长度的数学期望为：

$$\mathbb{E}[L] = \sum_{i=1}^n \frac{i}{n} = \frac{n+1}{2} \quad (2)$$

而首次适配中，每个块平均情况下能否被选中的次数大致相等，则选中的概率为 $\frac{1}{2}$ ，选择第一块的概率为 $\frac{1}{2}$ ，选择第二块的概率为 $\frac{1}{2} \times (1 - \frac{1}{2})$ ，以此类推。查找长度的期望为：

$$\mathbb{E}[L] = \sum_{i=1}^n \frac{1}{2^n} \times n = \left(2 - \frac{1}{2^{n-1}}\right) - \frac{n}{2^n} \quad (3)$$

显然，比较(2)和(3)，当 n 足够大时，首次适配的查找长度更短。

另一方面，首次适配可能拆分较大的内存块，导致剩余的内存块较小，而最佳适配则会尽量保留较大的内存块。然而，由图 4 中的例子可以看出，最佳适配的效果并不好，因为它留下大量可能无法再分配的细碎的内存块，例如上例中的大小为1的块。这会导致内存碎片的显著增加。

7 实验总结

本次实验让我更深入地了解了Linux内存分配和管理的原理和方法，以及各种算法的实现。通过实践，我学会了如何使用`kmalloc()`和`vmalloc()`函数，并学会了如何实现first fit算法和best fit算法。在实践过程中进一步锻炼了内核模块编程和算法实现的能力。在进行实验的过程中，我还学会了如何查找Linux系统内核的文档，并且更了解了Linux操作系统的运作方式和内部结构，这对于我今后的学习和研究都有很大的帮助。

代码 4 处理找不到块的情况

```
else
{
    char *page;
    union header *block;
    unsigned alloc_size = true_size;
    /* We have to request memory of at least a certain size. */
    if (alloc_size < NALLOC)
    {
        alloc_size = NALLOC;
    }
    page = sbrk((intptr_t)(alloc_size * sizeof(union header)));
    if (page == (char *)-1)
    {
        /* There was no memory left to allocate. */
        errno = ENOMEM;
        return NULL;
    }
    /* Create a fragment from this new memory and add it to the list
     * so the above logic can handle breaking it if necessary. */
    block = (union header *)page;
    block->meta.len = alloc_size;
    free((void *) (block + 1));
    return bf_malloc(size);
}
```
