操作系统实验五

OpenEuler 内存管理实验

邓语苏 2021级
计科 2024/06/04

实验介绍

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，加深初学者对 Linux 内存分配的理解。

在实验开始之前，需要注意以下两点：

1. 本次实验服务器已完成内核编译（openEuler 4.19.08），可直接开始实验；
2. 本次实验可能用到的内核函数有：kmalloc(), vmalloc(), kfree(), sbrk(), memset()，参数和返回类型请 在 https://manpages.org/查询。

实验目的

- 学习掌握 kmalloc()和 vmalloc()分配内存的差异；
- 加深学生对首次适应算法和最佳适应算法的理解；
- 锻炼学生编写内核模块的能力

任务一：使用 kmalloc 分配内存

编写kmalloc.c

调用 kmalloc() 函数分别为 kmallocmem1 和 kmallocmem2 分配 1KB和 8KB 大小 的内存空间并使用 printk() 打印指针地址；处理分配失败时的逻辑，在分配失败时打印`“Failed to allocate kmallocmem1/ kmallocmem2/ kmallocmem3 / kmallocmem4!\n”`

1. kmallocmem1

```c
kmallocmem1 = kmalloc(1024, GFP_KERNEL);
if (!kmallocmem1) {
printk(KERN_ERR "Failed to allocate kmallocmem1!\n"); }
else {
printk(KERN_INFO "kmallocmem1 addr = %p\n", kmallocmem1); }
```

2. kmallocmem2

```c
kmallocmem2 = kmalloc(8192, GFP_KERNEL);
if (!kmallocmem2) {
printk(KERN_ERR "Failed to allocate kmallocmem2!\n"); }
else {
printk(KERN_INFO "kmallocmem2 addr = %p\n", kmallocmem2); }
```

测试 kmalloc()可分配的内存大小是否有上限，若有，则寻找 kmalloc()申请内存的上限，为 kmallocmem3 申请最大可分配上限的内存空间，在实验报告中描述你是如何确定该上限的，并使用 printk() 打印指针地址； 同时为 kmallocmem4 申请比最大可分配上限稍大的内存空间；

3. kmallocmem3
    - 设置一个很大的初始猜测值，尝试使用系统的指针大小来确定内存的最大大小
    - 循环，每次迭代减小 max_alloc_size ，直到 kmalloc() 可以成功分配。如果成功分配了内存，就会打印出最大可分配大小和分配的内存地址，并且退出循环。

- 如果在循环中尝试了所有可能的大小后， kmalloc() 仍然无法分配内存，那么 kmallocmem3 将保持为空（NULL），并且会打印一条错误消息

```c
size_t max_alloc_size = (1 << (sizeof(void *) * 8 - 1)) - 1; // Initial guess
while (max_alloc_size > 0) {
    kmallocmem3 = kmalloc(max_alloc_size, GFP_KERNEL);
    if (kmallocmem3) {
        printk(KERN_INFO "kmallocmem3 max alloc size = %zu\n", max_alloc_size);
        printk(KERN_INFO "kmallocmem3 addr = %p\n", kmallocmem3);
        break;
    }
    max_alloc_size >>= 1; // Halve the size
}
if (!kmallocmem3) {
    printk(KERN_ERR "Failed to allocate kmallocmem3!\n");
}
```

4. kmallocmem4

```c
    kmallocmem4 = kmalloc(max_alloc_size + 10, GFP_KERNEL);
    if (!kmallocmem4) {
        printk(KERN_ERR "Failed to allocate kmallocmem4!\n");
    } else {
        printk(KERN_INFO "kmallocmem4 addr = %p\n", kmallocmem4);
    }
```

验证结果

1. 编写 Makefile 文件，执行 make (注意修改：KERNELDIR ?= /usr/lib/modules/$(shell uname -r)/build，使用本地内核)；
   由于linux系统更新问题，使用本地内核出现版本差错，因此仍使用root下的内核进行后续的实验
2. 加载模块，查看加载的模块内容，查看打印出的指针地址；

```
[  129.909947] kmallocmem1 addr = 000000001c9c8485
[  129.909970] kmallocmem2 addr = 000000009e70cb06
[  129.910054] kmallocmem3 max alloc size = 4194303
[  129.910075] kmallocmem3 addr = 00000000c53939e4
[  129.910174] Failed to allocate kmallocmem4!
[  132.154984] Exit kmalloc!
```

3. 根据机器是 32 位或者是 64 位的情况，分析分配结果是否成功以及地址落在的区域，并给出相应的解释
    - kmallocmem1 的地址为 0x000000001c9c8485，kmallocmem2 的地址为 0x000000009e70cb06。这些地址都是合法的内存地址，因此分配成功。
    - kmallocmem3 的最大可分配大小为 4194303 字节（大约 4 MB），地址为 0x00000000c53939e4。在 32 位系统中，通常内核空间的大小为 4 GB，其中约 1 GB 是内核空间，3 GB 是用户空间。kmalloc 在分配内存时会尽量使用物理连续的内存块，而在 32 位系统中，内核空间是连续的，所以 kmalloc 可以分配的最大连续空间大小为 4 MB 左右。这个值会因系统不同而有所差异，但大致在这个范围内。
    - 对于 kmallocmem4 的分配失败，打印了“Failed to allocate kmallocmem4!”。这是因为kmalloc 尝试分配比最大可分配上限稍大的内存空间，但由于已经超过了系统可用的连续内存大小，因此分配失败。

任务二：使用 vmalloc 分配内存

编写vmalloc.c

调用 vmalloc() 函数分别为 vmallocmem1、vmallocmem2、vmallocmem3 分配 8KB、1MB 和 64MB 大小的内存空间并使用 printk() 打印指针地址； 处 理 分配 失 败 时 的 逻 辑 ， 在 分 配 失 败 时 打 印 `“ Failed to allocate vmallocmem1/ vmallocmem2/ vmallocmem3!\n”

1. vmallocmem1

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

2. vmallocmem2

```c
vmallocmem2 = vmalloc(1048576);
if (!vmallocmem2) {
    printk(KERN_ERR "Failed to allocate vmallocmem2!\n");
} else {
    printk(KERN_INFO "vmallocmem2 addr = %p\n", vmallocmem2);
}
```

3. vmallocmem3

```c
vmallocmem3 = vmalloc(67108864);
if (!vmallocmem3) {
    printk(KERN_ERR "Failed to allocate vmallocmem3!\n");
} else {
    printk(KERN_INFO "vmallocmem3 addr = %p\n", vmallocmem3);
}
```

根据你在任务一中找到的 kmalloc 内存分配上限，请你为 vmallocmem4 分配比该上限稍大的内存；
4. vmallocmem4

```c
vmallocmem4 = vmalloc(4194303+10);
if (!vmallocmem4) {
    printk(KERN_ERR "Failed to allocate vmallocmem4!\n");
} else {
    printk(KERN_INFO "vmallocmem4 addr = %p\n", vmallocmem4);
}
```

验证结果

1. 编写 Makefile 文件，执行 make；
   由于linux系统更新问题，使用本地内核出现版本差错，因此仍使用root下的内核进行后续的实验。
2. 加载模块，查看加载的模块内容，查看打印出的指针地址；

```
[  796.693616] Start vmalloc!
[  796.693682] vmallocmem1 addr = 000000004486c2a3
[  796.694338] vmallocmem2 addr = 00000000bf8152fb
[  796.726273] vmallocmem3 addr = 0000000049c417d5
[  796.728337] vmallocmem4 addr = 000000003e24e19c
[  802.148878] Exit vmalloc!
```

3. 根据机器是 32 位或者是 64 位的情况，分析分配结果是否成功以及地址落在的区域，并给出相应的解释
    - vmallocmem1，vmallocmem2，vmallocmem3，vmallocmem4 的地址都是合法的内存地址，因此分配成功。
    - vmallocmem4 的地址小于 vmallocmem3，可能是由于地址空间的非连续性导致
    - vmalloc 可以分配虚拟连续的内存，但是不能保证物理连续。而 **kmalloc** 则要求物理续的页面。
      vma1loc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MALLOC_END指定。在i386架构(32位)下会优先选择 ZONE_HIGHMEM, 否则会选择ZONE_NORMAL, 在实验中，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ffffe8fffffffff。

任务三：阅读并理解首次适应算法的实现

相关知识点

首次适应**(First Fit)**算法在进行内存分配时，从空闲分区链首开始查找，直至找到一个能满足其大小需求的空闲分区，然后再按照作业的大小从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链中。

理解ff_malloc.c

数据结构

```c
union header {
    struct {
        union header *next;
        unsigned len;
    } meta;
    long x; /* 用于强制内存对齐 */
};
static union header list;//空闲链表
static union header *first = NULL;
```

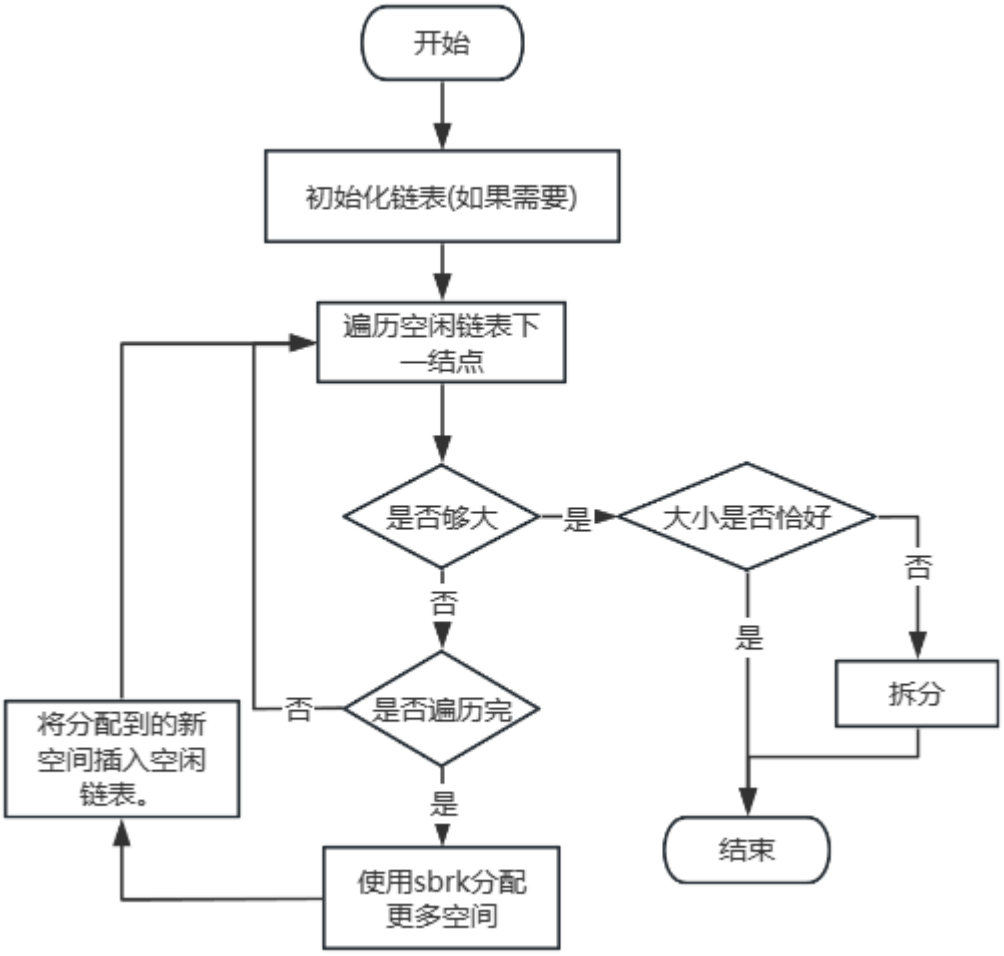
- header块记录了当前空闲块的长度和下一块的位置。
- list为空闲链表
- first为空闲链表的头指针

free函数

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

ff_malloc函数

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。具体如流程图



验证结果

将ff_malloc.c、test.c、makefile文件置于同一文件夹下。在命令行输入`make`进行编译，输入`./test`运行。

```
[root@openEuler exp7]# ./test
arrays [0][0] is OK.
arrays [1][0] is OK.
arrays [1][1] is OK.
arrays [2][0] is OK.
arrays [2][1] is OK.
arrays [2][2] is OK.
arrays [3][0] is OK.
arrays [3][1] is OK.
arrays [3][2] is OK.
arrays [3][3] is OK.
arrays [4][0] is OK.
arrays [4][1] is OK.
arrays [4][2] is OK.
arrays [4][3] is OK.
arrays [4][4] is OK.
arrays [5][0] is OK.
arrays [5][1] is OK.
arrays [5][2] is OK.
arrays [5][3] is OK.
```

首部

```
arrays [49][39] is OK.
arrays [49][40] is OK.
arrays [49][41] is OK.
arrays [49][42] is OK.
arrays [49][43] is OK.
arrays [49][44] is OK.
arrays [49][45] is OK.
arrays [49][46] is OK.
arrays [49][47] is OK.
arrays [49][48] is OK.
arrays [49][49] is OK.
[root@openEuler exp7]#
```

尾部

与test.c预想结果相同。

任务四：实现最佳适应算法

相关知识点

最佳适应(Best Fit)算法指从全部空闲区中找出能满足作业要求且大小最小的空闲分区的一种计算方法，这种方法能使碎片尽量小

编写bf_malloc.c

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作为最佳适应块。

```c
    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;
    }
```

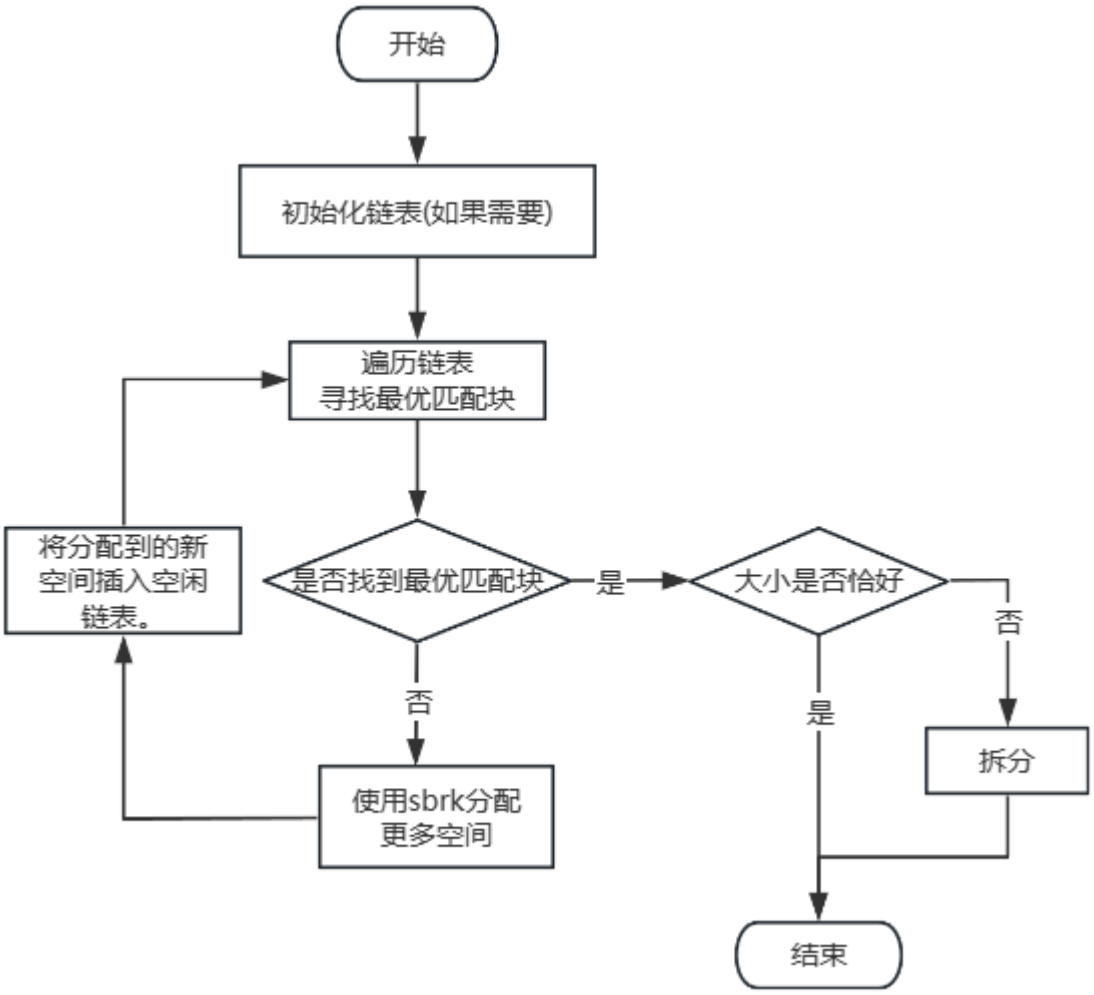
当找到这样的块时，采用和首次适配相同的拆分策略，并返回相应的内存。

```c
    if (best_fit != NULL)
    {
        union header *q = first;
        printf("freelist:{");
        do
        {
            printf("%d ", q->meta.len);
            q = q->meta.next;
        } while (q != first);
        printf("};\n");

        if (best_fit->meta.len == true_size)
        {
            /* If the fragment is exactly the right size, we do not
have
             * to split it. */
            best_fit_prev->meta.next = best_fit->meta.next;
        }
        else
        {
            /* Otherwise, split the fragment, returning the first
half and
             * storing the back half as another element in the
list. */
            best_fit->meta.len -= true_size;
            best_fit += best_fit->meta.len;
            best_fit->meta.len = true_size;
        }
        first = best_fit_prev;

        return (void *)(best_fit + 1);
    }
```

没找到这样的块，则使用sbrk系统调用分配内存，将新的块插入空闲链表中，然后再次调用bf_malloc函数进行分配。

```c
    else/*没找到最佳空闲块则使用sbrk系统调用分配内存*/
    {
        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);
    }
```

具体如以下流程图



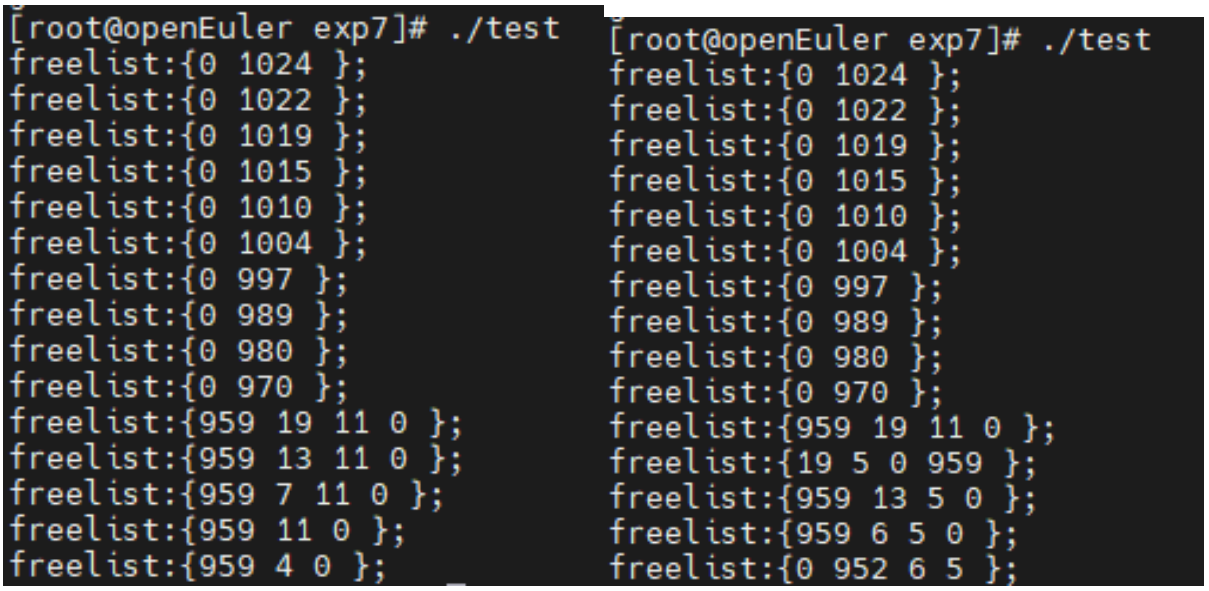
bf_malloc.c具体代码见`./code`文件夹

编写test.c

编写一个test函数，其含义是：进行10次内存分配，接着释放掉`arr[3]、arr[4]`和`arr[7]、arr[8]`这两片连续的内存。再进行5次内存分配。最后将没有释放的内存释放。根据代码可推导出，释放掉两片连续内存后的空闲队列应为`{959,19,11,0}`

test.c和makefile具体代码见`./code`文件夹

对比ff_malloc和bf_malloc差异



ff_malloc分配结果　　bf_malloc分配结果

可以看出，进行第11~15次的内存分配时，比较首次适应算法和最佳适应算法的处理方式不同。

第11次内存分配，应该分配给`arr[10]` `10/2+1=6`块内存。

- ff_malloc遍历空闲队列，首先遇到大小为19的内存片，于是拆分出其中的6块内存给`arr[10]`。空闲队列更新为`{959、13、11、0}`
- bf_malloc遍历空闲队列，找到了最接近6的内存片11，于是选择了其作为最佳匹配块，拆分出其中的6块内存给`arr[10]`。空闲队列更新为`{19、5、0、959}`

第12次内存分配，应该分配给`arr[11]` `11/2+1=6`块内存。

- ff_malloc遍历空闲队列，首先遇到大小为13的内存片，于是拆分出其中的6块内存给`arr[11]`。空闲队列更新为`{959、7、11、0}`
- bf_malloc遍历空闲队列，找到了最接近6的内存片19，于是选择了其作为最佳匹配块，拆分出其中的6块内存给`arr[11]`。空闲队列更新为`{959、13、5、0}`

实验心得体会

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。