湖南大學
HUNAN UNIVERSITY

操作系统实验报告

课 程 名 称：操作系统
实验项目名称：操作系统内核编程实验
专 业 班 级：软件 2203
姓　　名：白旭
学　　号：202226010306
指 导 教 师：周军海
完 成 时 间：2024 年 5 月 8 日

信息科学与工程学院

实验题目：实验三 内存管理

实验目的：

- **内存分配和管理之 kmalloc 和 vmalloc：**
  用 kmalloc 分配 1KB、8KB 的内存并打印指针地址
  用 vmalloc 分配 8KB、1MB，64MB 的内存并打印指针地址
  使用首次适应算法实现一个简单的 malloc 内存分配器，满足内存分配和释放
- **内存分配和管理之 iomap：**
  申请、读写、释放 I/O 端口
  申请、读写、释放 I/O 内存
- **思考题：**
  kmalloc()和 vmalloc()有何不同？

实验环境：

- 华为 ESC 弹性云服务器
- WinScp

实验内容及操作步骤：

1. 打开 ESC 弹性云服务器：

2. 使用 powershell 连接服务器与主机：

```
        Welcome to Huawei Cloud Service


Welcome to 4.19.90-2110.8.0.0119.oe1.aarch64

System information as of time:  Sun Apr 21 01:58:26 CST 2024

System load:    0.93
Processes:      164
Memory used:    4.9%
Swap used:      0.0%
Usage On:       8%
IP address:     192.168.1.122
Users online:   1



[root@openeuler ~]#
```

3. 编写代码

1) kmalloc.c

```c
#include <linux/module.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");

unsigned char *kmallocmem1;
unsigned char *kmallocmem2;

static int __init mem_module_init(void)
{
    printk("Start kmalloc!\n");
    kmallocmem1 = (unsigned char*)kmalloc(1024, GFP_KERNEL);
    if (kmallocmem1 != NULL){
        printk(KERN_ALERT "kmallocmem1 addr = %lx\n", (unsigned long)kmallocmem1);
    }else{
        printk("Failed to allocate kmallocmem1!\n");
    }
    kmallocmem2 = (unsigned char *)kmalloc(8192, GFP_KERNEL);
    if (kmallocmem2 != NULL){
        printk(KERN_ALERT "kmallocmem2 addr = %lx\n", (unsigned long)kmallocmem2);
    }else{
        printk("Failed to allocate kmallocmem2!\n");
    }
    return 0;
}

static void __exit mem_module_exit(void)
{
    kfree(kmallocmem1);
    kfree(kmallocmem2);
    printk("Exit kmalloc!\n");
}

module_init(mem_module_init);
module_exit(mem_module_exit);
```

2) vmalloc.c

```c
#include <linux/module.h>
#include <linux/vmalloc.h>

MODULE_LICENSE("GPL");

unsigned char *vmallocmem1;
unsigned char *vmallocmem2;
unsigned char *vmallocmem3;

static int __init mem_module_init(void)
{
    printk("Start vmalloc!\n");
    vmallocmem1 = (unsigned char*)vmalloc(8192);
    if (vmallocmem1 != NULL){
        printk("vmallocmem1 addr = %lx\n", (unsigned long)vmallocmem1);
    }else{
        printk("Failed to allocate vmallocmem1!\n");
    }
    vmallocmem2 = (unsigned char*)vmalloc(1048576);
    if (vmallocmem2 != NULL){
        printk("vmallocmem2 addr = %lx\n", (unsigned long)vmallocmem2);
    }else{
        printk("Failed to allocate vmallocmem2!\n");
    }
    vmallocmem3 = (unsigned char*)vmalloc(67108864);
    if (vmallocmem3 != NULL){
        printk("vmallocmem3 addr = %lx\n", (unsigned long)vmallocmem3);
    }else{
        printk("Failed to allocate vmallocmem3!\n");
    }
    return 0;
}

static void __exit mem_module_exit(void)
{
    vfree(vmallocmem1);
    vfree(vmallocmem2);
    vfree(vmallocmem3);
    printk("Exit vmalloc!\n");
}

module_init(mem_module_init);
module_exit(mem_module_exit);
```

3） emalloc.c

```c
// emalloc.c

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define HEAP_SIZE 1024 * 1024 // 堆的大小为1MB，这里作为示例设置

typedef struct BlockHeader
{
    size_t size;               // 内存块的大小
    bool is_free;              // 是否空闲
    struct BlockHeader *next; // 指向下一个内存块的指针
} BlockHeader;

typedef struct
{
    BlockHeader *free_blocks; // 指向第一个空闲内存块的指针
    char heap[HEAP_SIZE];      // 堆内存数组
} MemoryAllocator;

MemoryAllocator allocator; // 内存分配器实例
```

```c
void *emalloc(size_t size)
{
    BlockHeader *block = allocator.free_blocks;  // 从第一个空闲块开始搜索
    BlockHeader **prev = &allocator.free_blocks; // 用于更新链表指针的指针

    // 在空闲块链表中查找一个足够大的内存块
    while (block != NULL)
    {
        if (block->is_free && block->size >= size + sizeof(BlockHeader))
        {
            break;
        }
        prev = &block->next;
        block = block->next;
    }

    if (block == NULL)
    {
        return NULL; // 未找到合适的内存块
    }

    // 如果找到的内存块大于请求的大小，将剩余部分分割成一个新的空闲块
    if (block->size > size + sizeof(BlockHeader))
    {
        BlockHeader *new_block = (BlockHeader *)((char *)block + sizeof(BlockHeader) + size);
        new_block->size = block->size - size - sizeof(BlockHeader);
        new_block->is_free = true;
        new_block->next = block->next;

        block->size = size + sizeof(BlockHeader);
        block->next = new_block;
    }

    block->is_free = false;
    *prev = block->next; // 从空闲块链表中移除该块

    return (char *)block + sizeof(BlockHeader); // 返回分配内存的地址
}
```

```c
void efree(void *ptr)
{
    if (ptr == NULL)
        return;

    BlockHeader *block = (BlockHeader *)((char *)ptr - sizeof(BlockHeader)); // 获取内存块的头部
    block->is_free = true;

    // 如果下一个内存块也是空闲的，合并它们
    BlockHeader *next_block = (BlockHeader *)((char *)block + block->size);
    if (next_block->is_free)
    {
        block->size += next_block->size;
        block->next = next_block->next;
    }
    else
    {
        block->next = allocator.free_blocks; // 将块插入到空闲块链表头部
        allocator.free_blocks = block;
    }
}
```

emalloc.h

```c
// emalloc.h

#ifndef EMALLOC_H
#define EMALLOC_H
#define HEAP_SIZE 1024 * 1024 // 堆的大小为1MB，这里作为示例设置

#include <stdlib.h>

typedef struct BlockHeader
{
    size_t size;
    bool is_free;
    struct BlockHeader *next;
} BlockHeader;

typedef struct
{
    BlockHeader *free_blocks;
    char heap[HEAP_SIZE];
} MemoryAllocator;

extern void *emalloc(size_t size);
extern void efree(void *ptr);

#endif
```

test.c

```c
#include "emalloc.h"

MemoryAllocator allocator; // 声明内存分配器实例

#define HEAP_SIZE 1024 * 1024 // 堆的大小为1MB，这里作为示例设置

int main()
{
    // 初始化内存分配器
    allocator.free_blocks = (BlockHeader *)allocator.heap;
    allocator.free_blocks->size = HEAP_SIZE - sizeof(BlockHeader);
    allocator.free_blocks->is_free = true;
    allocator.free_blocks->next = NULL;

    bool allocation_ok = true;
    for (int i = 0; i < 50; i++)
    {
        for (int j = 0; j < 50; j++)
        {
            char *ptr = (char *)emalloc(50);
            if (ptr == NULL)
            {
                allocation_ok = false;
                break;
            }
            else
            {
                printf("array[%d][%d] is OK!\n", i, j);
            }
            efree(ptr);
        }
    }

    if (allocation_ok)
    {
        printf("Memory allocation is OK.\n");
    }
    else
    {
        printf("Memory allocation has issues.\n");
    }

    return 0;
}
```

4） 申请、读写、释放 I/O 端口

```c
#include <linux/module.h>
#include <asm/io.h>
#include <linux/ioport.h>

MODULE_LICENSE("GPL");

struct resource *myregion; // 用于保存请求到的资源信息的指针

static int __init mem_module_init(void)
{
    printk("Start request region!\n");

    // 请求IO端口区域
    myregion = request_region(22222, 10, "ve");

    if (myregion != NULL)
    {
        printk("it's ok for %lld .", myregion->start);
    }
    else
    {
        printk("Failed to request region!\n");
    }

    return 0;
}

static void __exit mem_module_exit(void)
{
    // 释放IO端口区域
    release_region(22222, 10);
    printk("Exit request_region!\n");
}

module_init(mem_module_init);
module_exit(mem_module_exit);
```

5） 申请、读写、释放 I/O 内存

```c
MODULE_LICENSE("GPL");

int value;
static void *io_memory;
static unsigned long io_memory_start;
static unsigned long io_memory_size = 0x1000; // 示意使用4KB

static int __init iomem_demo_init(void)
{
    printk(KERN_INFO "Start request mem region!\n");

    // 请求IO内存区域
    if (!request_mem_region(io_memory_start, io_memory_size, "mem"))
    {
        printk(KERN_ALERT "Cannot reserve IO memory region.\n");
        return -EBUSY;
    }

    // 将IO内存区域映射到内核地址空间
    io_memory = ioremap(io_memory_start, io_memory_size);
    if (!io_memory)
    {
        printk(KERN_ALERT "Failed to map IO memory.\n");
        release_mem_region(io_memory_start, io_memory_size);
        return -ENOMEM;
    }

    // 向IO内存写入数据，为0x6666
    writel(0x6666, io_memory);
    printk(KERN_INFO "it's ok for %p.\n", io_memory);

    // 从IO内存读取数据，读取之前写入的数据
    value = readl(io_memory);

    // 解除映射并释放IO内存区域
    iounmap(io_memory);
    release_mem_region(io_memory_start, io_memory_size);

    return 0;
}
```

4. 模块功能验证

1) kmalloc

```
[root@openeuler task1]# ls
kmalloc.c  Makefile  readme.txt
[root@openeuler task1]# make
make -C /usr/lib/modules/4.19.90-2110.8.0.0119.oe1.aarch64/build M=/root/mem
ory/exp1/task1 modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarc
h64'
  CC [M]  /root/memory/exp1/task1/kmalloc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/memory/exp1/task1/kmalloc.mod.o
  LD [M]  /root/memory/exp1/task1/kmalloc.ko
make[1]: Leaving directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch
64'
[root@openeuler task1]# ls
kmalloc.c   kmalloc.mod.c  kmalloc.o  modules.order   readme.txt
kmalloc.ko  kmalloc.mod.o  Makefile   Module.symvers
[root@openeuler task1]# insmod kmalloc.ko
[root@openeuler task1]# lsmod | grep kmalloc
kmalloc               262144  0
[root@openeuler task1]# rmmod kmalloc
[root@openeuler task1]# dmesg | tail -n6
[ 1568.339353] kmalloc: loading out-of-tree module taints kernel.
[ 1568.339815] kmalloc: module verification failed: signature and/or require
d key missing - tainting kernel
[ 1568.340907] Start kmalloc!
[ 1568.341091] kmallocmem1 addr = ffff8000cc873400
[ 1568.341385] kmallocmem2 addr = ffff8000c19a0000
[ 1605.130426] Exit kmalloc!
```

虚拟地址开头为 ffff8000 代表两个地址都在内核空间中

2) vmalloc

```
[root@openeuler task2]# make
make -C /usr/lib/modules/4.19.90-2110.8.0.0119.oe1.aarch64/build M=/root/mem
ory/exp1/task2 modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarc
h64'
  CC [M]  /root/memory/exp1/task2/vmalloc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/memory/exp1/task2/vmalloc.mod.o
  LD [M]  /root/memory/exp1/task2/vmalloc.ko
make[1]: Leaving directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.aarch
64'
[root@openeuler task2]# insmod vmalloc.ko
[root@openeuler task2]# lsmod | grep vmalloc
vmalloc               262144  0
[root@openeuler task2]# rmmod vmalloc
[root@openeuler task2]# dmesg | tail -n5
[ 8132.536332] Start vmalloc!
[ 8132.536644] vmallocmem1 addr = ffff00000d5a0000
[ 8132.536948] vmallocmem2 addr = ffff00000f1c0000
[ 8132.537438] vmallocmem3 addr = ffff000024ca0000
[ 8143.180529] Exit vmalloc!
```

虚拟地址开头为 ffff0000 代表三个地址都在内核空间中

这些地址是内核通过 vmalloc 函数动态分配的内存块的起始地址，用于存放大内存块。

3) emalloc

```
PS D:\桌面\work\操作系统\实验\实验三\memory\exp1\task3> gcc -o test test.c emalloc.c
PS D:\桌面\work\操作系统\实验\实验三\memory\exp1\task3> ./test
```

```
array[49][45] is OK!
array[49][46] is OK!
array[49][47] is OK!
array[49][48] is OK!
array[49][49] is OK!
Memory allocation is OK.
PS D:\桌面\work\操作系统\实验\实验三\memory\exp1\task3>
```

运行时请使用如下命令直接运行（makefile 没写好）

gcc -o test test.c emalloc.c

./test

4） 申请、读写、释放 I/O 端口

```
[root@openeuler task1]# make
make -C /usr/lib/modules/4.19.90-2110.8.0.0119.oe1.aarch64/build M=/root
/memory/exp2/task1 modules
make[1]: Entering directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.
aarch64'
  CC [M]  /root/memory/exp2/task1/request_region.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/memory/exp2/task1/request_region.mod.o
  LD [M]  /root/memory/exp2/task1/request_region.ko
make[1]: Leaving directory '/usr/src/kernels/4.19.90-2110.8.0.0119.oe1.a
arch64'
[root@openeuler task1]# insmod request_region.ko
[root@openeuler task1]# lsmod |grep request_region
request_region        262144  0
[root@openeuler task1]# rmmod request_region
[root@openeuler task1]# dmesg | tail -n4
[ 8143.180529] Exit vmalloc!
[21650.016342] Start request region!
[21650.016595] it's ok for 22222 .
[21674.973552] Exit request_region!
```

5） 申请、读写、释放 I/O 内存

```
[root@openeuler task2]# insmod request_mem_region.ko
[root@openeuler task2]# lsmod |grep request_mem_region
request_mem_region    262144  0
[root@openeuler task2]# rmmod request_mem_region
[root@openeuler task2]# dmesg | tail -n3
[25614.283536] Start request mem region!
[25614.285205] it's ok for 000000009d6a5eae.
[25621.219171] Exit request_region!
```

实验结果及分析：

1. 实验结果与预期相符，第三次实验学习了内存管理相关函数与相关知识，完成得比较顺利，源代码可以正常运行

收获与体会：

1. 最后读取内存中的数据时会发生溢出错误
2. 做完实验要及时将 ESC 关机以节约经费
3. 掌握了如何进行内核模块编程

思考题：

1. kmalloc()和 vmalloc()有何不同？

   1) kmalloc() 用于分配小块连续的物理内存，通常用于分配不大的内存块。它分配的内存位于内核的堆区，是物理内存的连续区域。由于分配的内存是连续的，所以在一些对内存连续性有要求的场景中比较适用。使用 kmalloc() 需要指定要分配的内存大小和分配标志（如 GFP_KERNEL）

   2) vmalloc() 用于分配大块的虚拟内存，通常用于分配比较大的内存块。它分配的内存位于内核的虚拟地址空间，可能是非连续的，通过页表映射到物理内存的不同区域。由于分配的内存不要求连续，所以在需要大块内存但连续性要求不高的场景中比较适用。使用 vmalloc() 需要指定要分配的内存大小

2. 首次适应算法的特点和缺点

   特点：

   1) 简单直观：首次适应算法是最简单的内存分配算法之一，容易实现和理解

   2) 快速分配：由于分配器从链表的开头开始查找空闲块，因此可以快速找到满足要求的内存块进行分配

   缺点：

   1) 性能下降：当内存中存在大量大小相近的空闲块时，首次适应算法可能会导致性能下降，因为每次都要遍历链表查找满足要求的空闲块

   2) 不公平性：首次适应算法可能导致某些小的空闲块一直无法被分配，而大的空闲块却能够被很快分配，造成不公平性

3. 什么是内部碎片，什么是外部碎片

   1) 内部碎片

      当分配的内存空间大于请求的内存空间时，会产生内部碎片。这是因为分配器为了满足对齐等要求，可能会分配比请求的内存空间更大的块，而未使用的部分就成为了内部碎片。内部碎片是分配器和应用程序之间的浪费，无法被其他程序使用

   2) 外部碎片

      当已分配的内存块之间存在很多小的不连续的未分配空间时，会产生外部碎片。这些未分配的小空间虽然总和可能足够分配一个大块内存，但是由于它们不连续，无法被有效利用，造成了外部碎片。外部碎片是分配器和操作系统之间的浪费，通常需要特殊的算法来处理

4. 简单介绍最佳适应算法/下次适应算法

1) 最佳适应算法会在空闲内存块中选择最小且能满足需求的块分配给请求。它会遍历所有空闲块来找到最合适的块。这样可以减少内部碎片，但可能会增加搜索时间。因为需要找到最小的合适块，所以算法复杂度较高

2) 下次适应算法是首次适应算法的改进版。它从上一次分配结束的地方开始查找合适的空闲块。这样可以减少搜索的时间，但可能会导致更多的内部碎片。因为相邻的小块可能无法合并使用

实验成绩	

实验报告撰写说明

1．实验题目和目的

请从实验指导资料中获取。

2．实验步骤和内容

包括：

（1）本次实验的要求；

（2）源程序清单或者主要伪代码；

（3）预期结果；

（4）上机执行或调试结果：包括原始数据、相应的运行结果和必要的说明（截图）；

3．实验体会

调试中遇到的问题及解决办法；若最终未完成调试，要试着分析原因；调试程序的心得与体会；对课程及实验的建议等。