# 广州大学学生实验报告

**开课学院及实验室：**计算机科学与教育软件学院计算机软件实验室 2020**年**4**月**9**日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | 计算机科学与教育软件学院 | **年级/专业/班** | 软件171 | **姓名** | 谢金宏 | **学号** | 1706300001 |
| **实验课**  **程名称** | Linux系统实验 | | | | | **成绩** |  |
| **实验项**  **目名称** | 内存分配与回收 | | | | | **指导**  **老师** | 张艳玲 |

## 实验目的

1. 了解静态内存与动态内存的区别；
2. 理解动态内存的配和释放原理；
3. 掌握如何调整动态内存的大小；
4. 利用链表实现动态内存分配。

## 实验内容

1. 利用malloc和 calloc函数实现动态内存的分配；利用free函数实现动态内存的释放；
2. 利用realloc函数实现调整内存空间的大小；
3. 利用链表实现动态内存分配；
4. 编写一个虚拟字符设备驱动模块，在其驱动中完成设备空间——也就是内核空间——到用户空间的内存映射，即将内核空间的线性地址所对应的物理地址映射到用户空间的线性地址中，从而实现在用户空间直接读取内核数据的目的。

## 实验涉及的系统调用以及内核函数

1. **分配动态内存空间所使用的函数调用如下：**

*#include<stdio.h>*

*void \*malloc(size\_t size);*

*void \*calloc(size\_t nmemb, size\_t size);*

函数malloc和calloc都是用于分配动态内存空间的函数。

函数malloc的参数size表示申请分配的内存空间的大小，以字节记。

函数calloc的参数nmemb表示申请分配的内存空间占的数据项数目，参数size表示一个数据项的大小，以字节计。也就是说，calloc函数分配大小为nmemb \* size大小的内存空间。

函数calloc与函数malloc的最大区别就是函数calloc将初始化所分配的内存空间，把所有位置0。调用成功时，函数calloc与函数malloc的返回值都为被分配的内存空间的指针；调用失败时，返回值为NULL。

1. **释放动态内存空间所使用的函数调用如下：**

*#include<stdio.h>*

*void free(void \*ptr);*

此函数的作用是释放由函数calloc或函数malloc分配的动态内存。参数ptr是指向要释放的动态内存的指针。

注意：当动态内存被释放后，原来指向它的指针就会变为悬空指针。此时使用该指针将会产生错误。

1. **对于用函数calloc与函数malloc分配好的动态内存，可以使用realloc函数来调整它的大小。该函数的说明如下：**

#include<stdio.h>

void\*realloc(void \*ptr, size\_t size);

realloc函数的作用是重新调整一块动态内存区域的大小，参数ptr是指向要调整的动态内存的指针，应是函数calloc与函数malloc的返回值。参数size是新定义的动态内存的大小。Size可以大于或小于动态内存的原大小，调用realloc函数时，通常是在原来的内存空间调整动态内存的大小，原有数据不被改动。当size大于原大小，而原位置中无法完成调整时，将重新开辟内存空间并将原数据拷贝到新的内存空间中。

注意：如果参数ptr为NULL，则函数realloc的作用相当于函数malloc。如果参数size为0，则函数realloc的作用相当于函数free。

## 实验原理与重点

1. **静态内存与动态内存**

按分配内存空间的方式不同，一个程序所使用的内存区域可以分为静态内存与动态内存。在程序开始运行时由系统分配的内存称为静态内存，在程序运行过程中由用户自己申请分配的内存称为动态内存。

静态内存的申请是由编译器来分配的。当程序执行时，系统就为变量分配所需的内存空间，至使用该变量的函数执行完毕返回时，自动释放所占用的内存空间。用户并不需要了解分配内存的具体细节，也不需要时刻考虑由于程序结束前未释放所占用的内存空间而带来的可用内存泄漏。同时，静态内存也是不通过指针而使用变量的唯一方法。

使用动态内存时，用户可以根据需要随时申请内存，使用完毕后手动将此内存区释放。在实际应用中非常方便。

1. **使用链表进行动态内存的分配**

虽然使用动态内存可以方便地使用内存，但动态内存也有局限性，就是在数据输入到程序之前必须知道数据的大小，以便申请相应的动态内存。

链表是一种动态地进行存储分配的结构。链表中的各个元素是一个结构，每个元素称为链表的一个结点。此结构中包含有一个指向此结构的指针，用于指向链表中的下一个结点。链表的最后一个结点的指针NULL，表示链表结束。

1. **静态内存的缺陷（重点）**

首先，由于静态内存总是预先定义了存放数据的数组大小，这就有可能因为所传入的数据量大于数组容量而引发的溢出问题。或因为定义了一个大数组，而所传入的数据量远小于数组容量，而对内存空间造成浪费。

其次，由于在某个函数中分配的静态内存将在此函数运行结束时被系统自动释放，使用指针由子函数向主函数传递数据的设想是无法被实现的。

1. **使用动态内存存在的巨大隐患（重点）**

存在内存泄露的风险（通常将内存分配后没有被释放而导致可用内存减少称之为内存泄漏）。另外还要注意，由于分配动态内存时，用户得到的是一块void类型的内存，用户可以将其作为任何类型的内存空间使用，也可能引发一些无法预计的结果。

## 实验原始数据、结果与分析

1. **利用malloc函数和free函数来实现动态内存的申请和释放。**

*// task1\_malloc.c*

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

char \*uppercase(const char \*input\_string);

int main(void)

{

    char \*str1 = uppercase("Hello");

    char \*str2 = uppercase("Goodbye");

    printf("str1 = %s\nstr2 = %s\n", str1, str2);

    free(str1);

    free(str2);

    return 0;

}

char \*uppercase(const char \* input\_string)

{

    char \*new\_string;

    if ((new\_string = malloc(strlen(input\_string) + 1)) == NULL)

    {

        printf("Fail to allocate memory.\n");

        exit(EXIT\_FAILURE);

    }

    strcpy(new\_string, input\_string);

    for (int i = 0; i < strlen(new\_string); i++)

    {

        if (new\_string[i] >= 97 && new\_string[i] <= 122) {

            new\_string[i] -= 32;

        }

    }

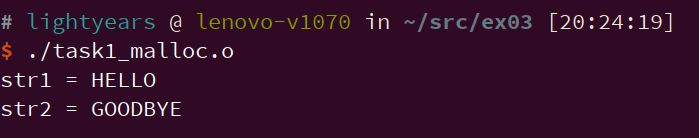
    return new\_string;

}

Main函数调用uppercase函数，并分别传入“Hello”和“Goodbye”两个参数。

在uppercase函数中，使用malloc为字符串分配内存，分配内存的长度设置为输入字符串的长度 + 1个字节（因为字符串的末尾还有一个不可见的空字符，需要额外占用1字节空间）。使用strcpy将输入字符串的内容复制到malloc开辟的新空间中，最后将新字符串中的每个小写字符转换为对应的大写字符。

在main函数中打印uppercase函数返回的新字符串，并使用free释放malloc分配的空间。



1. **利用realloc函数来修改上述malloc函数申请的动态内存的大小。**

*// task2\_realloc.c*

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

void uppercase(const char \*input\_string, char \*\*new\_string);

int main(void)

{

    char \*string = NULL;

    uppercase("Hello", &string);

    printf("str1 = %s\n", string);

    uppercase("Goodbye", &string);

    printf("str2 = %s\n", string);

    free(string);

    return 0;

}

void uppercase(const char \*input\_string, char \*\*new\_string)

{

    if (new\_string == NULL)

    {

        printf("Expect a pointer to (char \*) type but got a NULL pointer.\n");

        exit(EXIT\_FAILURE);

    }

    if ((\*new\_string = realloc(\*new\_string, strlen(input\_string) + 1)) == NULL) {

        printf("Fail to allocate memory.\n");

        exit(EXIT\_FAILURE);

    };

    strcpy(\*new\_string, input\_string);

    for (int i = 0; i < strlen(\*new\_string); i++)

    {

        if ((\*new\_string)[i] >= 97 && (\*new\_string)[i] <= 122)

        {

            (\*new\_string)[i] -= 32;

        }

    }

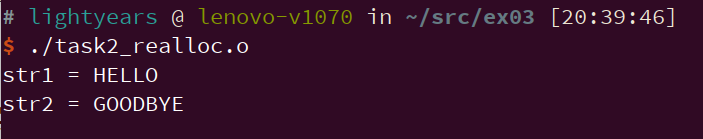
}

Main函数中声明了一个char \*类型的指针变量string，用于指向新字符串的位置。先后两次调用修改的uppercase函数，并传入指向string的地址作为参数。

在基于实验题1修改的uppercase函数中，使用realloc为新字符串开辟合适大小的内存空间，并令string指针指向用realloc分配的内存空间的起始位置。然后使用strcpy将输入字符串复制到新空间中，最后将新空间中的字符串中的小写字母转换为对应的大写字母。

调用uppercase函数后，在main函数中打印string指针执行的字符串。

Main函数最后需要用free释放string指针。



1. **利用链表结构，malloc函数和free函数实现将终端输入的一系列字符串用链表的形式保存下来。然后再将这些数据组装起来，回显到输出终端。**

程序头文件和结构定义如下：

*// task3\_list.c*

#include <stdlib.h>

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#define BUFFER\_SIZE 16

typedef struct

{

    char \*string;

    struct data\_cell \*next;

} data\_cell, \*data\_cell\_ptr;

typedef struct

{

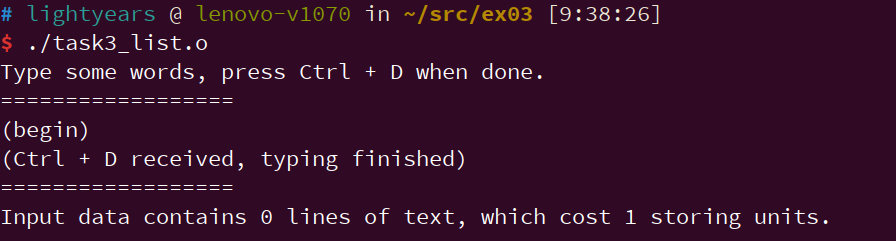
    data\_cell\_ptr first\_cell;

    data\_cell\_ptr tail\_cell;

} data\_head, \*data\_head\_ptr, \*data;

本程序中定义的链表包含一个头节点和若干个数据节点。链表头节点data\_head定义中包含指向链表中第一个数据节点的指针first\_cell。Data\_head还包含指向链表的最后一个数据节点的指针tail\_cell，这个指针用于快速定位末尾数据节点，以便加快在链表末尾插入的数据的速度。链表数据节点data\_cell定义中包含指向文本数据的string指针字段，以及指向链表中后继节点的next指针。

注意到这种数据结构的组织方式，即使在没有数据的情况下也要消耗一个内存单元（如下图所示）。



void data\_init(data\_head\_ptr\* ptr)

{

    \*ptr = malloc(sizeof(data\_head));

    (\*ptr)->first\_cell = NULL;

    (\*ptr)->tail\_cell = NULL;

}

Data\_init函数负责初始化链表的头节点。函数中使用malloc为头节点在内存中分配空间，并将头节点的两个字段初始化为NULL。

void data\_append(data\_head\_ptr head, const char \*text)

{

    data\_cell\_ptr cell = (data\_cell\_ptr)malloc(sizeof(data\_cell));

    cell->string = (char \*)malloc(strlen(text) + 1);

    strcpy(cell->string, text);

    cell->next = NULL;

    if (head->tail\_cell == NULL) {

        head->first\_cell = head->tail\_cell = cell;

    } else {

        head->tail\_cell->next = (struct data\_cell \*)cell;

        head->tail\_cell = cell;

    }

}

Data\_append函数用于在链表末尾追加数据。函数需要传入链表头节点指针head和指向追加文本数据内容的字符串指针text作为参数。函数中使用malloc为数据节点以及数据节点中的text字段开辟空间，随后使用strcpy将追加文本数据text的内容复制到malloc开辟的内存空间中，最后将新增的节点挂接到链表的末尾。

void data\_display(data\_head\_ptr head)

{

    int line\_count = 0, unit\_count = 1;

    data\_cell\_ptr cur = head->first\_cell;

    bool should\_print\_line\_mark = true;

    while (cur != NULL)

    {

        const char \*string = cur->string;

        size\_t len = strlen(string);

        for (int i = 0; i < len; ++i) {

            if (should\_print\_line\_mark) {

                printf("%2d > ", line\_count + 1);

                should\_print\_line\_mark = false;

            }

            fputc(string[i], stdout);

            if (string[i] == '\n') {

                ++line\_count;

                should\_print\_line\_mark = true;

            }

        }

        ++unit\_count;

        cur = (data\_cell\_ptr)(cur->next);

    }

    printf("Input data contains %d lines of text, which cost %d storing units.\n",

        line\_count, unit\_count);

}

Data\_display用于遍历链表并显示链表数据节点中的文本数据。函数中还统计了储存的文本数据的行数以及总共占用储存节点的数量。

void data\_destroy(data\_head\_ptr \*ptr)

{

    data\_cell\_ptr cur = (\*ptr)->first\_cell, next;

    while (cur != NULL) {

        next = (data\_cell\_ptr)(cur->next);

        free(cur->string);

        free(cur);

        cur = next;

    }

    free(\*ptr);

    \*ptr = NULL;

}

Data\_destroy函数用于销毁链表，使用free回收malloc分配的链表空间。销毁链表时，先释放数据节点的text字段指向的空间，随后释放数据节点，最后释放链表的头节点。

int main(void)

{

    data console\_data;

    data\_init(&console\_data);

    printf("Type some words, press Ctrl + D when done.\n");

    printf("==================\n");

    printf("(begin)\n");

    char text\_buffer[BUFFER\_SIZE];

    while (fgets(text\_buffer, sizeof(text\_buffer), stdin))

    {

        data\_append(console\_data, text\_buffer);

    }

    printf("(Ctrl + D received, typing finished)\n");

    printf("==================\n");

    data\_display(console\_data);

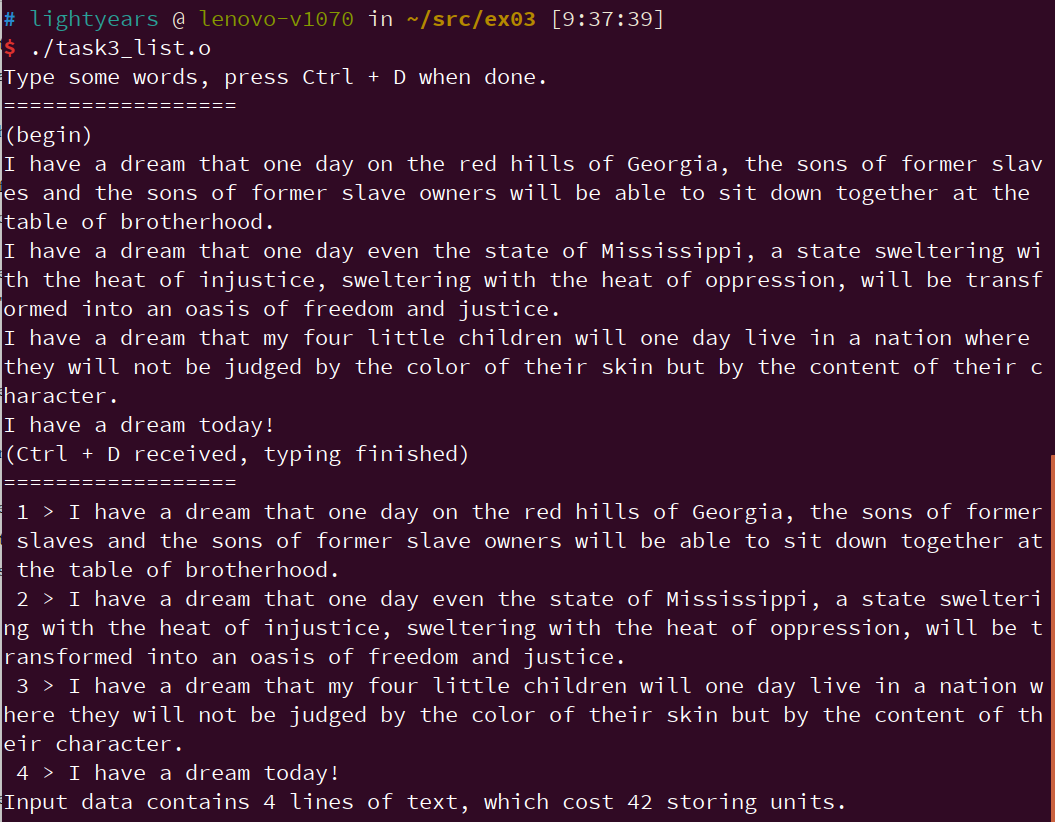
    data\_destroy(&console\_data);

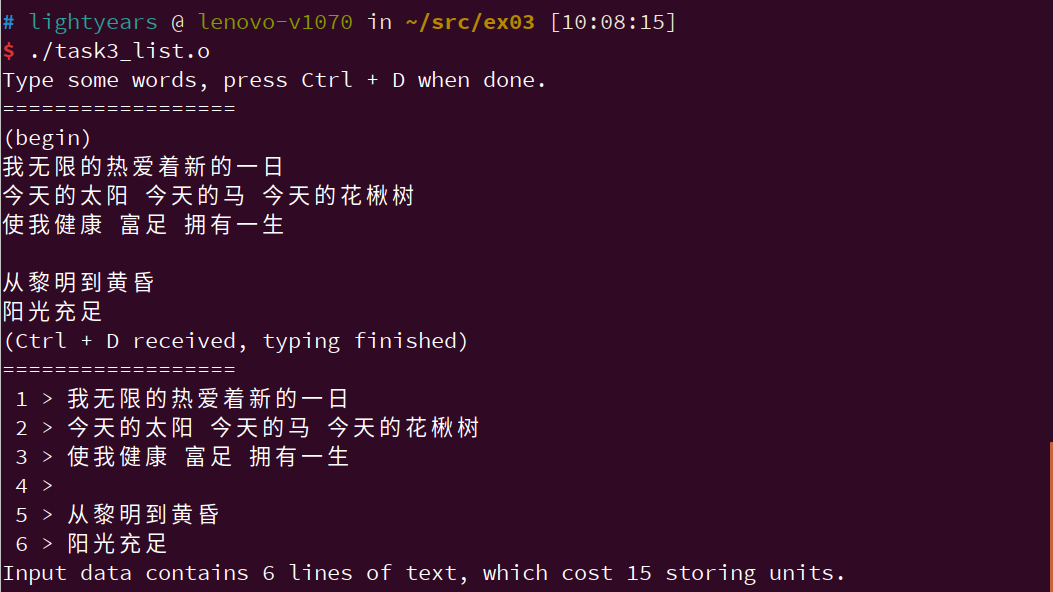
    return 0;

}

Main函数中依次调用data\_init、data\_append、data\_display和data\_destroy函数，演示它们的功能。

程序的一次运行截图如下。





1. **编写一个虚拟字符设备驱动模块，在其驱动中完成设备空间——也就是内核空间——到用户空间的内存映射，即将内核空间的线性地址所对应的物理地址映射到用户空间的线性地址中，从而实现在用户空间直接读取内核数据的目的。**

内核模块的代码如下：

*// map\_driver.c*

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/string.h>

#include <linux/errno.h>

#include <linux/mm.h>

#include <linux/vmalloc.h>

#include <linux/slab.h>

#include <linux/sched.h>

#include <linux/io.h>

#include <linux/mman.h>

#define MAP\_PAGE\_COUNT 10

#define MAPLEN (PAGE\_SIZE \* MAP\_PAGE\_COUNT)

#define MAP\_DEV\_MAJOR 280

#define MAP\_DEV\_NAME "mapnopage"

#define PRINTK\_TAG KERN\_INFO MAP\_DEV\_NAME ": "

void map\_vopen(struct vm\_area\_struct \*vma);

void map\_vclose(struct vm\_area\_struct \*vma);

static int mapdrv\_mmap(struct file \*file, struct vm\_area\_struct \*vma);

static int mapdrv\_open(struct inode \*inode, struct file \*file);

static vm\_fault\_t map\_fault(struct vm\_fault \*vmf);

static struct file\_operations mapdrvo\_fops = {

    .owner = THIS\_MODULE,

    .mmap = mapdrv\_mmap,

    .open = mapdrv\_open,

};

static struct vm\_operations\_struct map\_vm\_ops = {

    .open = map\_vopen,

    .close = map\_vclose,

    .fault = map\_fault,

};

static char \*vmalloc\_area = NULL;

static int \_\_init mapdrv\_init(void)

{

    int result;

    unsigned long virt\_addr;

    int i = 1;

    result = register\_chrdev(MAP\_DEV\_MAJOR, MAP\_DEV\_NAME, &mapdrvo\_fops);

    if (result < 0)

    {

        return result;

    }

    vmalloc\_area = vmalloc(MAPLEN);

    virt\_addr = (unsigned long)vmalloc\_area;

    for (virt\_addr = (unsigned long)vmalloc\_area;

         virt\_addr < (unsigned long)vmalloc\_area + MAPLEN;

         virt\_addr += PAGE\_SIZE)

    {

        SetPageReserved(vmalloc\_to\_page((void \*)virt\_addr));

        sprintf((char \*)virt\_addr, "Test %d", i++);

    }

    printk(PRINTK\_TAG "[%s] vmalloc\_area apply completely!\n", \_\_func\_\_);

    return 0;

}

static void \_\_exit mapdrv\_exit(void)

{

    unsigned long virt\_addr;

*// Unreserve all pages.*

    for (virt\_addr = (unsigned long)vmalloc\_area;

         virt\_addr < (unsigned long)vmalloc\_area + MAPLEN;

         virt\_addr += PAGE\_SIZE)

    {

        ClearPageReserved(vmalloc\_to\_page((void \*)virt\_addr));

    }

*// Free vmalloc\_area and unregister char device.*

    if (vmalloc\_area)

        vfree(vmalloc\_area);

    unregister\_chrdev(MAP\_DEV\_MAJOR, MAP\_DEV\_NAME);

}

static int mapdrv\_mmap(struct file \*file, struct vm\_area\_struct \*vma)

{

    unsigned long offset = vma->vm\_pgoff << PAGE\_SHIFT;

    unsigned long size = vma->vm\_end - vma->vm\_start;

    if (size > MAPLEN)

    {

        printk(PRINTK\_TAG "[%s] size too big.\n", \_\_func\_\_);

        return ENXIO;

    }

*// Support shared mapping only.*

    if ((vma->vm\_flags & VM\_WRITE) && !(vma->vm\_flags & VM\_SHARED))

    {

        printk(PRINTK\_TAG "[%s] writeable mappings must be shared, rejecting.\n", \_\_func\_\_);

        return -EINVAL;

    }

*// Lock area to avoid it swapping out.*

    vma->vm\_flags |= VM\_LOCKONFAULT;

    if (offset == 0)

        vma->vm\_ops = &map\_vm\_ops;

    else

    {

        printk(PRINTK\_TAG "[%s] offset out of range.\n", \_\_func\_\_);

        return -ENXIO;

    }

    return 0;

}

static int mapdrv\_open(struct inode \*inoe, struct file \*file)

{

    printk(PRINTK\_TAG "[%s] process: %s(%d)\n", \_\_func\_\_, current->comm, current->pid);

    return 0;

}

void map\_vopen(struct vm\_area\_struct \*vma)

{

    printk(PRINTK\_TAG "[%s] mapping vma is opened.\n", \_\_func\_\_);

}

void map\_vclose(struct vm\_area\_struct \*vma)

{

    printk(PRINTK\_TAG "[%s] mapping vma is closed.\n", \_\_func\_\_);

}

*// PageFault handler*

static vm\_fault\_t map\_fault(struct vm\_fault \*vmf)

{

    struct page \*page;

    void \*page\_ptr;

    unsigned long offset = vmf->address - vmf->vma->vm\_start;

    unsigned long pfn\_start;

    unsigned long virt\_start = (unsigned long)vmalloc\_area + (unsigned long)(vmf->pgoff << PAGE\_SHIFT);

    page\_ptr = NULL;

    if ((vmf->vma == NULL) || (vmalloc\_area == NULL))

    {

        printk(PRINTK\_TAG "[%s] return VM\_FAULT\_SIGBUS!\n", \_\_func\_\_);

        return VM\_FAULT\_SIGBUS;

    }

    if (offset >= MAPLEN)

    {

        printk(PRINTK\_TAG "[%s] return VM\_FAULT\_SIGBUS!\n", \_\_func\_\_);

        return VM\_FAULT\_SIGBUS;

    }

    page\_ptr = vmalloc\_area + offset;

    page = vmalloc\_to\_page(page\_ptr);

    pfn\_start = vmalloc\_to\_pfn(page\_ptr);

    get\_page(page);

    vmf->page = page;

    printk(PRINTK\_TAG "[%s] map 0x%lx(0x%016lx) to 0x%lx, size: 0x%lx, page: %ld.\n",

           \_\_func\_\_, virt\_start, pfn\_start << PAGE\_SHIFT, vmf->address, PAGE\_SIZE, vmf->pgoff);

    return 0;

}

MODULE\_LICENSE("GPL");

module\_init(mapdrv\_init);

module\_exit(mapdrv\_exit);

内核模块的主要功能是：在插入内核时向系统中注册字符设备mapnopage，并申请内核空间中的一块虚存，并锁定这块虚存对应的物理内存，防止它被换出。

当用户程序通过mmap将mapnopage设备映射到自己的虚存空间时，内核进入mapdrv\_mmap函数。这个函数主要对mmap的一些flags进行检查，并设置一些必要的flags。

当用户程序第一次read或write被映射到设备的虚存空间时，产生缺页，内核进入map\_fault函数。这个函数将用户程序的映射到mapnopage设备的虚存空间与mapnopage驱动程序在内核中申请的虚存空间相联系，使得用户程序虚存与驱动程序的在内核空间的虚存对应同一块物理内存，从而使得用户程序能够直接读取内核中的数据。

在离开内核时，程序注销注册的字符设备mapnopage，将初始化时申请的虚存对应的物理内存解锁，并向内核归还申请的虚存。

用户测试程序如下：

*// maptest\_read.c*

#include <stdio.h>

#include <unistd.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#define LEN (10 \* 4096)

int main(void)

{

   int fd;

   if ((fd = open("/dev/mapnopage", O\_RDWR)) < 0)

      return 0;

   char \*vadr = mmap(0, LEN, PROT\_READ, MAP\_PRIVATE | MAP\_LOCKED, fd, 0);

   for (int loop = 0; loop < 2; loop++)

      printf("[%lx]\t%s\n", (unsigned long)vadr + 4096 \* loop, vadr + 4096 \* loop);

   while (1)

      sleep(1);

   return 0;

}

*// maptest\_write.c*

#include <stdio.h>

#include <unistd.h>

#include <sys/mman.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdlib.h>

#define LEN (10 \* 4096)

int main(void)

{

   int fd;

   if ((fd = open("/dev/mapnopage", O\_RDWR)) < 0)

      return 0;

   char \*vadr = mmap(0, LEN, PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_LOCKED, fd, 0);

   sprintf(vadr, "Write from userspace");

   while (1)

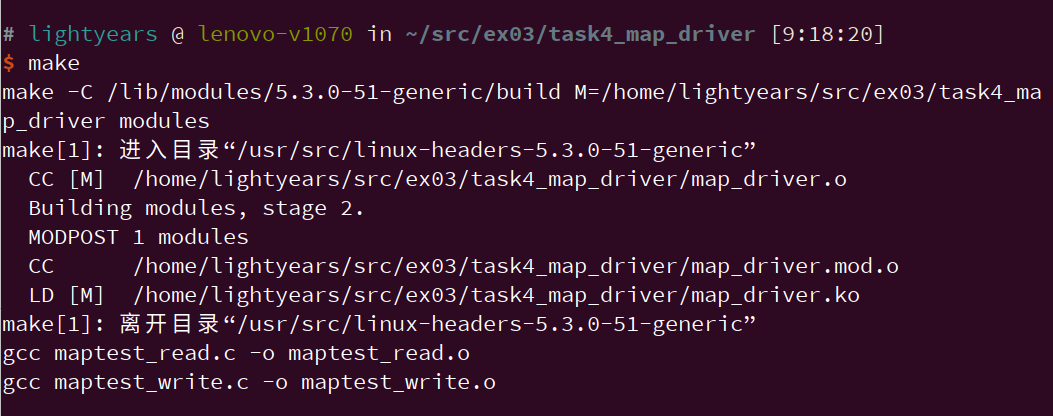
      sleep(1);

   return 0;

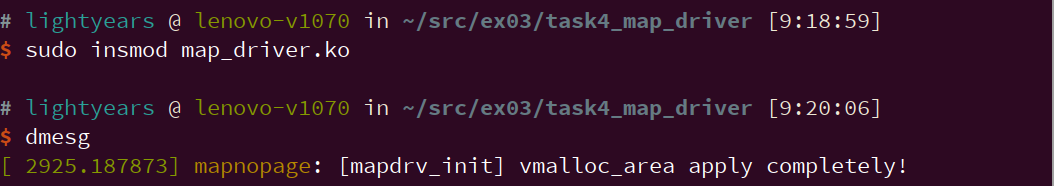
}

用户测试程序通过mmap将设备空间映射到自己的虚存，也就是将内核的线性地址空间映射到了用户的线性地址空间。随后用户测试程序通过read和write就能实现在用户空间直接读写内核数据的目的。

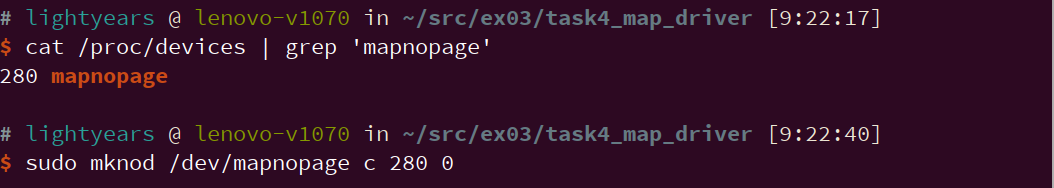
使用make编译内核模块和用户程序。



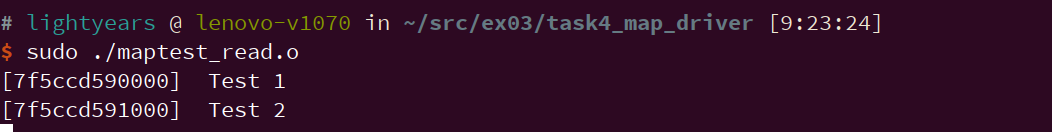
使用insmod插入内核模块，并使用dmesg查看内核日志，可见驱动程序已成功插入。



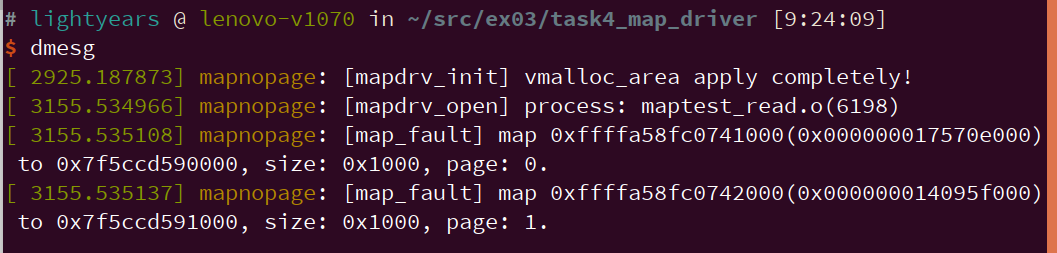
使用mknod在系统的dev目录下创建字符设备节点。



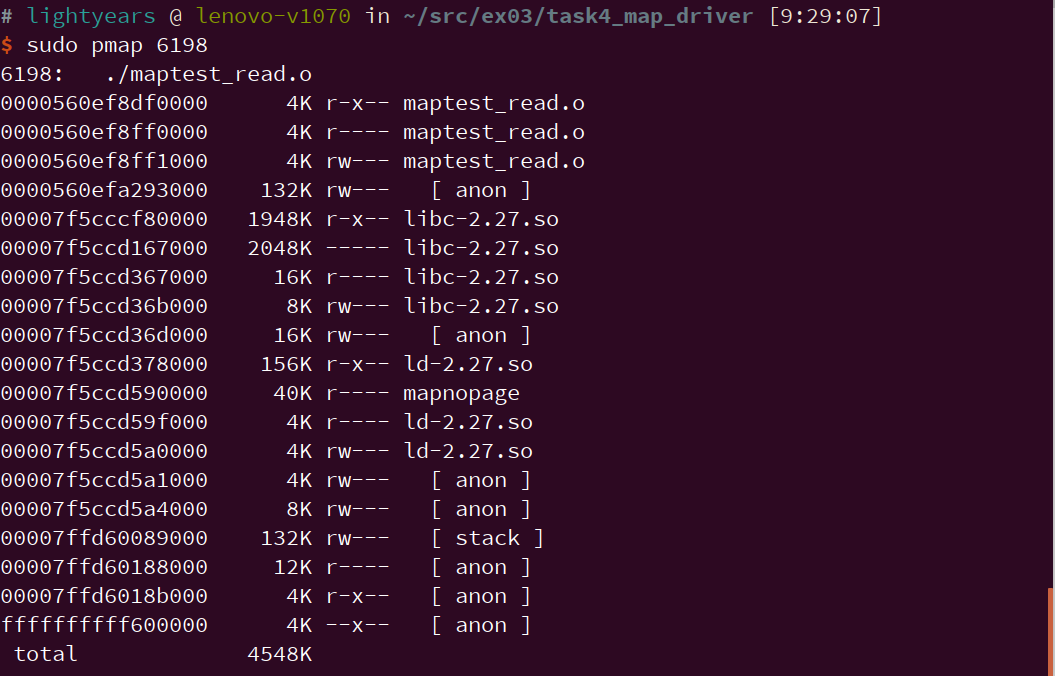
执行读程序，读出驱动程序在初始化时设置的文本信息。



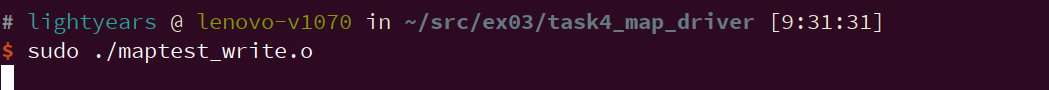
使用dmesg查看内核日志，打印如下。



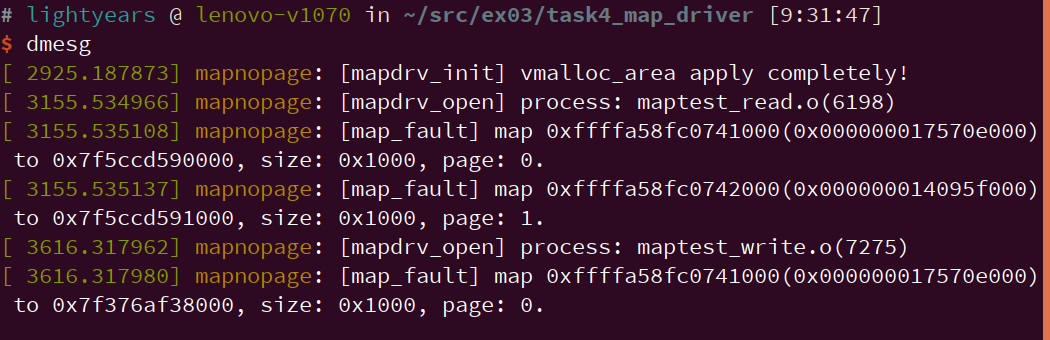
使用pmap查看进程的内存映射信息，可见设备空间映射到了用户程序的线性空间。



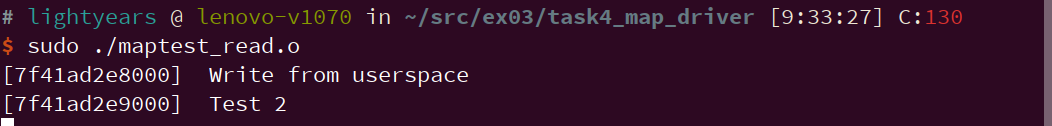
执行写程序，向内核空间写入一行文本数据。

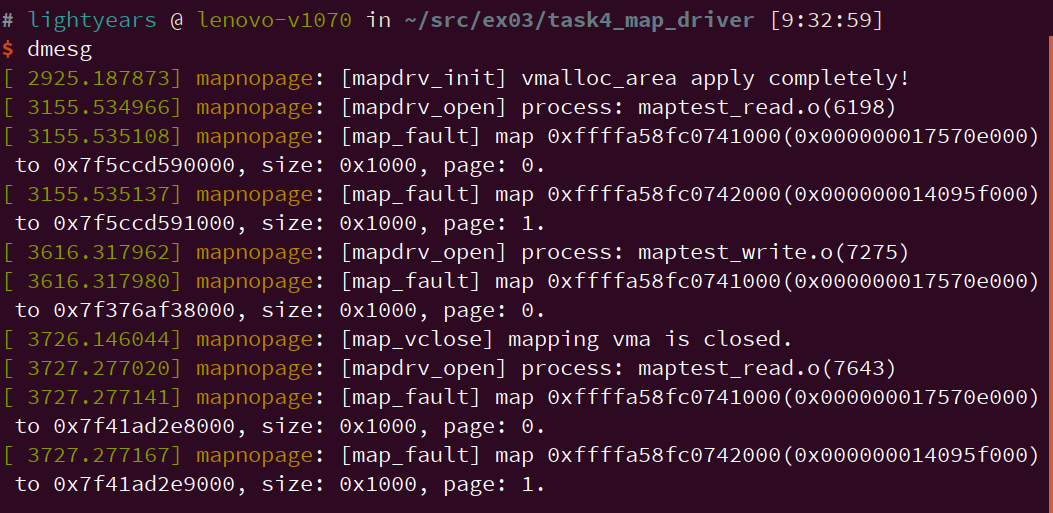


使用dmesg查看内核日志。

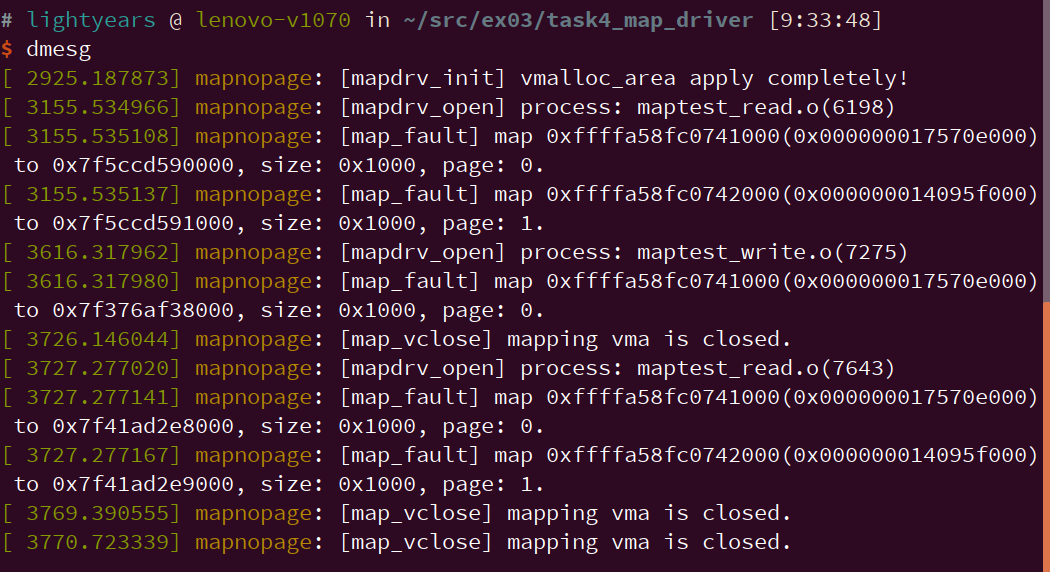


再执行读程序，可以验证写程序已经成功向内核空间写入数据。





结束读程序和写程序，并使用dmesg查看内核日志，结果如下。可见驱动已经进行了必要的清理工作。



## 实验思考题

**查阅资料后认真分析：在使用动态内存时，哪些情况下可能出现内存泄漏？如何防止内存泄漏？**

内存泄漏是指由于疏忽或错误造成程序未能释放已经不再使用的内存。下面的代码中，demo1\_leak.c是一个出现内存泄漏问题的程序，而demo1\_fixed是一个正常的程序。出现内存泄漏的进程会占用大量的资源，最后操作系统不得不将其杀死（如下图所示）。

*// demo\_framework.c*

#include <stdlib.h>

#include <stdbool.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

extern void test(void);

int main(void)

{

    if (fork() == 0) {

        test();

    }

    int ret;

    wait(&ret);

    if (WIFEXITED(ret)) {

        printf("exited, status=%d\n", WEXITSTATUS(ret));

    }

    else if (WIFSIGNALED(ret)) {

        printf("killed by signal %d\n", WTERMSIG(ret));

    }

    else if (WIFSTOPPED(ret)) {

        printf("stopped by signal %d\n", WSTOPSIG(ret));

    }

}

*// demo1\_leak.c*

#include <stdlib.h>

#include <stdbool.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

void test(void) {

    for (long long cnt = 0; cnt < 1e7; ++cnt)

    {

        void \*p = malloc(40960000);

        if (p == NULL)

        {

            printf("Can't allocate memoery!\n");

            exit(EXIT\_FAILURE);

        }

    }

    exit(EXIT\_SUCCESS);

}

*// demo1\_fixed.c*

#include <stdlib.h>

#include <stdbool.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

void test(void) {

    for (long long cnt = 0; cnt < 1e7; ++cnt)

    {

        void \*p = malloc(40960000);

        if (p == NULL)

        {

            printf("Can't allocate memoery!\n");

            exit(EXIT\_FAILURE);

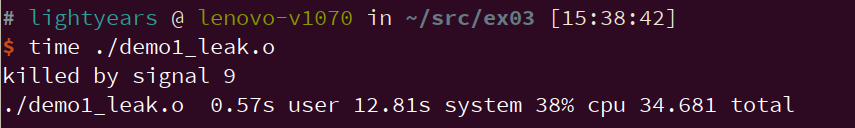
        }

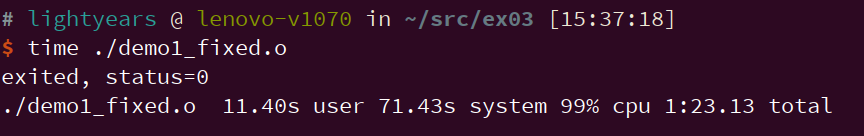
        free(p);

    }

    exit(EXIT\_SUCCESS);

}





内存泄漏有以下几类常见情形：

1. **单纯地忘记释放申请的内存。**

*// demo2\_wrong.c*

#include <stdlib.h>

*// Wrong*

void test(void) {

    void \*p = malloc(4096);

*// Do something with p...*

    return; *// Forget to free(p)!*

}

针对这种情况，应当养成良好的编程习惯，malloc可以与free成对书写。

*// demo2\_correct.c*

#include <stdlib.h>

*// Correct*

void test(void) {

    void \*p = malloc(4096);

*// Do something with p...*

    free(p);

}

要尽量在同一个函数中申请和释放动态内存，否则容易出现忘记释放内存的情况。例如：

*// demo3\_bad.c*

#include <stdlib.h>

*// Bad*

int\* iota(int len) {

    int \*p = malloc(sizeof(int) \* len);

    for (int i = 0; i < len; ++i) {

        p[i] = i;

    }

}

int main(void) {

    int \*arr = iota(100);

*// Do somehing with arr...*

    return 0; *// Forget to free(arr)!*

}

针对这种情况，可以适当地改变API的形式，将内存的申请和释放集中起来。

*// demo3\_better.c*

#include <stdlib.h>

*// Better*

void iota(int \*p, int len) {

    for (int i = 0; i < len; ++i) {

        p[i] = i;

    }

}

int main(void) {

    int len = 100;

    int \*arr = malloc(sizeof(int) \* len);

    iota(arr, len);

*// Do something with arr...*

    free(arr);

}

在动态申请结构体的内存时要小心。例如：

*// demo4\_bad.c*

#include <stdlib.h>

#include <string.h>

struct data {

    char \*p;

    int size;

};

*// Wrong*

int main(void) {

    const char \*text = "This is a text";

    struct data \*pdata = malloc(sizeof(struct data));

    pdata->p = malloc(strlen(text) + 1);

    strcpy(pdata->p, text);

    pdata->size = strlen(text) + 1;

*// Do something with data...*

    free(pdata);  *// Forget to free(pdata -> p)!*

    return 0;

}

上面的代码中忘记释放为结构体的字段申请的内存。针对这种情况，可以明确地使用构造函数和析构函数。

*// demo4\_better.c*

#include <stdlib.h>

#include <string.h>

struct data {

    char \*p;

    int size;

};

void init\_data(struct data \*\*pdata, const char \*text) {

    \*pdata = malloc(sizeof(struct data));

    int len = strlen(text);

    (\*pdata)->p = malloc(len + 1);

    strcpy((\*pdata)->p, text);

    (\*pdata)->size = len;

}

void destroy\_data(struct data \*\*pdata) {

    free((\*pdata)->p);

    free(\*pdata);

    \*pdata = NULL;

}

int main(void) {

    const char \*text = "This is a text";

    struct data \*pdata;

    init\_data(&pdata, text);

*// Do something with pdata...*

    destroy\_data(&pdata);

    return 0;

}

1. **对指针重新赋值时，忘记释放原来的内存。**

*// demo5\_wrong.c*

#include <stdlib.h>

#include <stdio.h>

int main(void) {

    char \*tmp = malloc(16);

    strcpy(tmp, "Hello, guys!");

    printf("%s\n", tmp);

    tmp = malloc(32); *// Forget to free(tmp) before reassigment!*

    strcpy(tmp, "I'm Tom, Nice to meet you here.");

    printf("%s\n", tmp);

}

在对指针重新赋值前，需要检查是否需要释放内存；或者用realloc代替malloc重新赋值。

*// demo5\_correct.c*

#include <stdlib.h>

#include <stdio.h>

int main(void) {

    char \*tmp = malloc(16);

    strcpy(tmp, "Hello, guys!");

    printf("%s\n", tmp);

    if (tmp != NULL) {

        free(tmp);

        tmp = NULL;

    }

    tmp = malloc(32);

    strcpy(tmp, "I'm Tom, Nice to meet you here.");

    printf("%s\n", tmp);

}

*// demo5\_better.c*

#include <stdlib.h>

#include <stdio.h>

int main(void) {

    char \*tmp = malloc(16);

    strcpy(tmp, "Hello, guys!");

    printf("%s\n", tmp);

    tmp = realloc(tmp, 32);

    strcpy(tmp, "I'm Tom, Nice to meet you here.");

    printf("%s\n", tmp);

}

1. **没有在所有分支路径释放内存。**

*// demo6\_wrong.c*

#include <stdbool.h>

bool random(void) {

    char \*tmp = malloc(8);

    if (rand() % 2 == 1) {

        return false;

*// Forget to free(tmp) before return in this branch!*

    }

    free(tmp);

    return true;

}

应该注意在所有的分支路径都释放动态申请的内存。

*// demo6\_correct.c*

#include <stdlib.h>

#include <stdbool.h>

bool random(void) {

    char \*tmp = malloc(8);

    if (rand() % 2 == 1) {

        free(tmp);

        return false;

    }

    free(tmp);

    return true;

}

1. **释放已经释放过的指针。**

*// demo7\_wrong.c*

#include <stdlib.h>

int main(void) {

    char \*buffer = malloc(16);

    char \*hint = malloc(16);

*// Do something...*

    free(buffer);

*// Do something...*

    free(buffer); *// buffer is freed twice!*

    free(hint);

}

可将释放过的悬空指针及时设置为NULL，这样即使再次释放这个指针，程序也不会出问题。

*// demo7\_wrong.c*

#include <stdlib.h>

int main(void) {

    char \*buffer = malloc(16);

    char \*hint = malloc(16);

*// Do something...*

    free(buffer);

    buffer = NULL;

*// Do something...*

    free(buffer); *// A NULL pointer can be safely freed.*

    free(hint);

}