­­­Министерства науки и высшего образования Российской Федерации

федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

*Факультет Программной инженерии и компьютерной техники*

**Лабораторная работа №2:**

Дисциплина «Операционные системы»

Студент:

Петропавлова Т. В.

Группа:

P33212

Преподаватель:

Саржевский И. А.

Санкт-Петербург

2023 г.

Оглавление

[Задание 2](#_Toc152981365)

[Выполнение 2](#_Toc152981366)

[Листинг модуля ядра 5](#_Toc152981367)

[Заголовочный файл core\_mode.h 5](#_Toc152981368)

[Код модуля core\_mode.c 5](#_Toc152981369)

[Код пользовательского приложения user\_mode.c 9](#_Toc152981370)

[Файл сборки модуля Makefile 11](#_Toc152981371)

[Установка и запуск модуля 11](#_Toc152981372)

[Пример работы комплекса программ: 11](#_Toc152981373)

[Вывод 12](#_Toc152981374)

# Задание

Разработать комплекс программ на пользовательском уровне и уровне ярда, который собирает информацию на стороне ядра и передает информацию на уровень пользователя, и выводит ее в удобном для чтения человеком виде. Программа на уровне пользователя получает на вход аргумент(ы) командной строки (не адрес!), позволяющие идентифицировать из системных таблиц необходимый путь до целевой структуры, осуществляет передачу на уровень ядра, получает информацию из данной структуры и распечатывает структуру в стандартный вывод. Загружаемый модуль ядра принимает запрос через указанный в задании интерфейс, определяет путь до целевой структуры по переданному запросу и возвращает результат на уровень пользователя.

*Интерфейс передачи:* ioctl

*Целевая структура (утилита):* slabtop -o

# Выполнение

ioctl – это управляющий системный вызов, который передает данные устройствам и файлам. В качестве аргументов принимает дескриптор файла/файла устройства и код запроса, обозначающего операцию.

Утилита slabtop представляет собой вывод информации об использовании блоков кэша ядра (слабах). Утилита обрабатывает файл /proc/slabinfo и показывает статистику работы кэша объектов ядра (например, dentry и заголовки буферов).

Slabtop -o единоразово выводит информацию на момент выполнения команды.

Утилита предоставляет следующие метрики:

OBJS – количество объектов определенного объекта кэша

ACTIVE – активные объекты

USE – процент использования кэша

OBJ SIZE – размер одного объекта

SLABS – количество блоков памяти для данного вида кэша

OBJ/SLAB – количество объектов в блоке

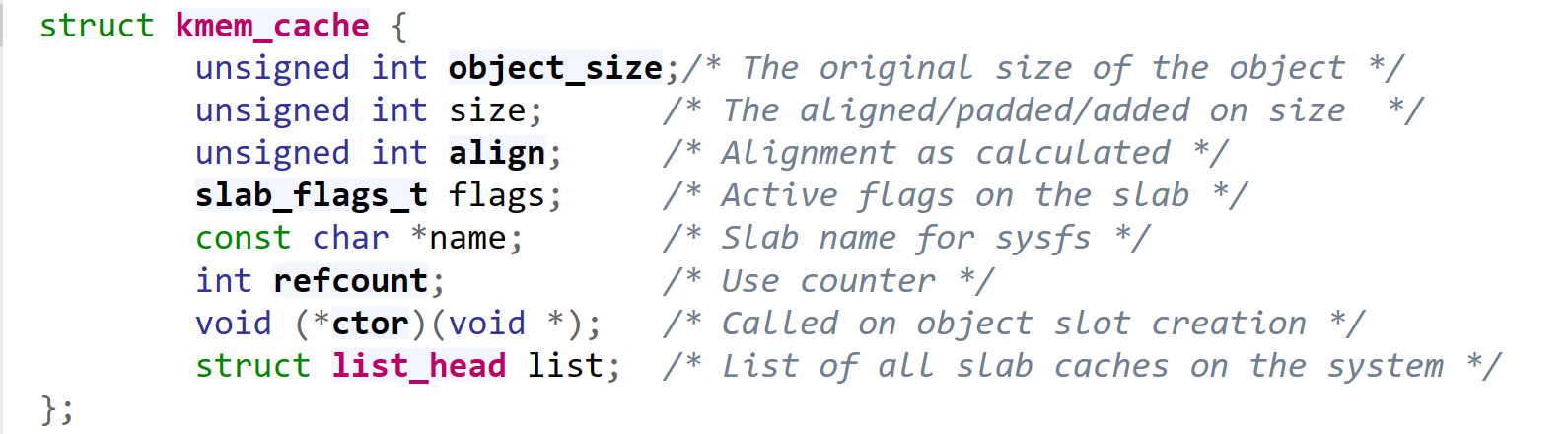
CACHE SIZE – общий размер кэша (согласно документации, в данный момент точным не является)

NAME – название кэша

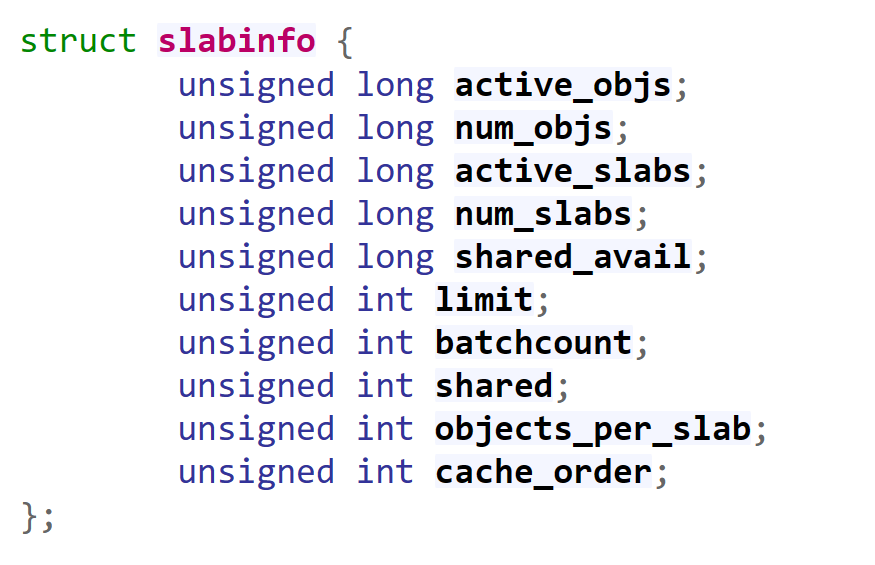
Для реализации имитации вывода утилиты slabtop без использования файла /proc/slabinfo (откуда сама утилита читает информацию) были выбраны несколько видов кэша ядра:

1. vm\_area\_struct – структура, представляющая виртуальную область памяти процессора, используется для отслеживания управлением областями этой памяти в адресном пространстве. Расположение: /include/linux /mm\_types.h
2. dentry – структура файловой системы, используется для управления кэшами иерархией имен файловой системы, содержит информацию о положениях файлов в файловой системе. Расположение: /include/linux/dcache.h
3. vmap\_area – структура, отвечающая за отображение адресов из физической памяти в виртуальную, используется для непосредственного выделения памяти в адресном пространстве при вызове vmalloc. Расположение: /include/linux/vmalloc.h
4. anon\_vma\_chain – структура, которая связывает анонимные VMA cс определенным регионом виртуальной памяти. Анонимная VMA (Virtual Memory Area) – это область, которая не отображена в физической памяти. Расположение: /include/linux/rmap.h

Непосредственно для получения информации о блоках кэша была использована структура kmem\_cache из /mm/slab.h:



Для подготовки вывода slabtop была использована структура slabinfo из этого же заголовочного файла:



**О структуре:**

*active\_objs* – количество активных объектов в определенном кэше

*num\_objs* – общее количество объектов

*active\_slabs* – количество блоков памяти, которые сейчас используются

*num\_slabs* – общее количество блоков для этого кэша

*shared\_avail* – количество доступных shared-блоков (где хранятся объекты, которые можно разделить между несколькими процессами)

*limit* – ограничение количества объектов для кэша

*batchcount* – количество объектов, добавляемых/извлекаемых из кэша за один раз

*shared* – количество объектов, которые хранятся в shared-блоках

*objects\_per\_slab* – количество объектов, которые можно разместить в одном блоке

*cache\_order* – степень размера страницы блока (размер считается так: 2^cache\_order – количество стандартных страниц, у меня на vm ubuntu – 4096 байт)

**Описание алгоритма работы комплекса программ:**

Пользователю предлагается выбрать одну из четырех структур для просмотра информации о ней.

Через интерфейс ioctl (создается файл устройства) на уровень ядра передается выбор пользователя.

При помощи макроса KMEM\_CACHE (из /include/linux/slab.h), который принимает структуру и флаги и вызывает kmem\_cache\_create() из того же файла, создается кэш заданной структуры кэша ядра. Я использовала флаги SLAB\_PANIC (при неудаче создания кэша все аварийно завершается) и SLAB\_ACCAUNT (включает счетчики и статистику для отслеживания использования кэша).

Вызывается функция get\_slabinfo, которая обрабатывает kmem\_cache и заполняет структуру slabinfo.

Вызывается функция set\_my\_type, которая обрабатывает структуру slabinfo и заполняет структуру, использующуюся для вывода, чтобы заполнить поле самой структуры ответа.

Структура ответа (результат и выбор пользователя) передаются на уровень пользователя через уже созданный файл устройства.

# Листинг модуля ядра

## Заголовочный файл core\_mode.h

#define RW\_ANS \_IOR('a', 'a', struct answer\*)  
  
#define BUFFER\_SIZE 1024  
#define DEVICE\_NAME "DEVICESLABTOP"  
#define DNAME\_INLINE\_LEN 32  
  
struct my\_slabinfo {  
 unsigned long active\_objs;  
 unsigned long num\_objs;  
 unsigned int use;  
 unsigned long obj\_size;  
 unsigned long num\_slabs;  
 unsigned int objects\_per\_slab;  
 unsigned long cache\_size;  
};  
  
struct answer {  
 struct my\_slabinfo sld;  
 int st\_num;  
};

В нем приводится структура my\_slabinfo с полями как у команды slabtop и структура ответа для передачи на уровень пользователя.

## Код модуля core\_mode.c

#include <linux/kernel.h>  
#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/kdev\_t.h>  
#include <linux/fs.h>  
#include <linux/cdev.h>  
#include <linux/device.h>  
#include <linux/slab.h> //kmalloc()  
#include <linux/uaccess.h> //copy\_to/from\_user()  
#include <linux/ioctl.h>  
#include <linux/err.h>  
#include <linux/namei.h>  
#include <linux/vmalloc.h>  
#include <linux/dcache.h>  
#include <linux/types.h>  
#include <linux/numa.h>  
#include<linux/gfp.h>  
#include <linux/mm.h>  
#include <linux/buffer\_head.h>  
#include <linux/mm\_types.h>  
#include <linux/rmap.h>  
#include "/home/boss/linux-6.2/mm/slab.h"  
#include "core\_mode.h"  
  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("TatianaP");  
MODULE\_DESCRIPTION("Lab Work 2 ioctl slabtop -o");  
  
// to registrate my device in dev  
dev\_t dev = 0;  
static struct class \*dev\_class;  
  
// character device for ioctl  
static struct cdev my\_driver\_cdev;  
  
// file status  
static int status\_driver\_open(struct inode \*inode, struct file \*file)  
{  
 pr\_info("Device File Opened...\n");  
 return 0;  
}  
  
static int status\_driver\_release(struct inode \*inode, struct file \*file)  
{  
 pr\_info("Device File Closed...\n");  
 return 0;  
}  
  
  
// process status  
static ssize\_t status\_driver\_read(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*off)  
{  
 pr\_info("Read Function\n");  
 return 0;  
}  
  
static ssize\_t status\_driver\_write(struct file \*filp, const char \_\_user \*buf, size\_t len, loff\_t \*off)  
{  
 pr\_info("Write function\n");  
 return len;  
}  
  
  
// print my slabinfo struct to log  
static void print\_slabinfo(struct my\_slabinfo\* slb){  
 pr\_info("Slabinfo \n");  
 pr\_info("number of objects = %lu, \n", slb->num\_objs);  
 pr\_info("active objects = %lu, \n", slb->active\_objs);  
 pr\_info("fake use = %u, \n", slb->use);  
 pr\_info("object size in bytes = %lu", slb->obj\_size);  
 pr\_info("number of slabs = %lu, \n", slb->num\_slabs);  
 pr\_info("objects per slab = %u, \n", slb->objects\_per\_slab);  
 pr\_info("fake cache size = %luK, \n", slb->cache\_size);  
 pr\_info("\n");  
}  
  
// print answer struct to log  
static void print\_answer(struct answer\* a){  
 pr\_info("\ngoing to send:");  
 print\_slabinfo(&(a->sld));  
}  
  
  
// functions from source code  
static inline unsigned long node\_nr\_slabs(struct kmem\_cache\_node \*n)  
{  
 return atomic\_long\_read(&n->nr\_slabs);  
}  
  
static inline unsigned long node\_nr\_objs(struct kmem\_cache\_node \*n)  
{  
 return atomic\_long\_read(&n->total\_objects);  
}  
  
static unsigned long count\_partial(struct kmem\_cache\_node \*n,  
 int (\*get\_count)(struct slab \*))  
{  
 unsigned long flags;  
 unsigned long x = 0;  
 struct slab \*slab;  
  
 spin\_lock\_irqsave(&n->list\_lock, flags);  
 list\_for\_each\_entry(slab, &n->partial, slab\_list)  
 x += get\_count(slab);  
 spin\_unlock\_irqrestore(&n->list\_lock, flags);  
 return x;  
}  
static int count\_free(struct slab \*slab)  
{  
 return slab->objects - slab->inuse;  
}  
  
#define OO\_SHIFT 16  
#define OO\_MASK ((1 << OO\_SHIFT) - 1)  
static inline unsigned int oo\_order(struct kmem\_cache\_order\_objects x)  
{  
 return x.x >> OO\_SHIFT;  
}  
  
static inline unsigned int oo\_objects(struct kmem\_cache\_order\_objects x)  
{  
 return x.x & OO\_MASK;  
}  
  
// from kmem\_cache to slabinfo from source code  
void get\_slabinfo(struct kmem\_cache \*s, struct slabinfo \*sinfo)  
{  
 unsigned long nr\_slabs = 0;  
 unsigned long nr\_objs = 0;  
 unsigned long nr\_free = 0;  
 int node;  
 struct kmem\_cache\_node \*n;  
  
 for\_each\_kmem\_cache\_node(s, node, n) {  
 nr\_slabs += node\_nr\_slabs(n);  
 nr\_objs += node\_nr\_objs(n);  
 nr\_free += count\_partial(n, count\_free);  
 }  
  
 sinfo->active\_objs = nr\_objs - nr\_free;  
 sinfo->num\_objs = nr\_objs;  
 sinfo->active\_slabs = nr\_slabs;  
 sinfo->num\_slabs = nr\_slabs;  
 sinfo->objects\_per\_slab = oo\_objects(s->oo);  
 sinfo->cache\_order = oo\_order(s->oo);  
}  
  
// preparing real slabinfo for answer  
void set\_my\_type (struct my\_slabinfo \*m\_sinfo, struct slabinfo \*sinfo, struct kmem\_cache \*kmems){  
 m\_sinfo->active\_objs = sinfo->active\_objs;  
 m\_sinfo->num\_objs = sinfo->num\_objs;  
 m\_sinfo->num\_slabs = sinfo->num\_slabs;  
 m\_sinfo->use = 0;  
 m\_sinfo->obj\_size = kmems->object\_size;  
 m\_sinfo->objects\_per\_slab = sinfo->objects\_per\_slab;  
 m\_sinfo->cache\_size = 0;  
  
}  
  
  
#define SLAB\_PANIC ((slab\_flags\_t \_\_force)0x00040000U)  
#define SLAB\_NO\_USER\_FLAGS ((slab\_flags\_t \_\_force)0x10000000U)  
  
  
static long core\_mode\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg){  
  
 if(cmd == RW\_ANS){  
 struct my\_slabinfo \*m\_slb;  
 struct kmem\_cache \*my\_cache;  
 struct slabinfo \*norm\_slab;  
 struct answer\* a = vmalloc(sizeof(struct answer));  
 if (!a) {  
 pr\_err("Memory allocation failed!\n");  
 a->status = "NOT OK";  
 return -ENOMEM;  
 }  
  
 // copy error handler  
 if (copy\_from\_user(a, (struct answer \*)arg, sizeof(struct answer))) {  
 pr\_err("Can't read answer from user space!\n");  
 a->status = "NOT OK";  
 vfree(a);  
 return -EFAULT;  
 }  
  
  
 pr\_info("STRUCT\_ID = %d\n", a->st\_num);  
  
 switch(a->st\_num){  
 default:  
 case 1:  
 my\_cache = KMEM\_CACHE(vm\_area\_struct, SLAB\_PANIC|SLAB\_ACCOUNT);  
 break;  
 case 2:  
 my\_cache = KMEM\_CACHE(dentry, SLAB\_PANIC|SLAB\_ACCOUNT);  
 break;  
 case 3:  
 my\_cache = KMEM\_CACHE(vmap\_area, SLAB\_PANIC|SLAB\_ACCOUNT);  
 break;  
 case 4:  
 my\_cache = KMEM\_CACHE(anon\_vma\_chain, SLAB\_PANIC|SLAB\_ACCOUNT);  
 break;  
 }  
  
  
 if (!my\_cache){  
 printk("No cache");  
 return -EFAULT;  
 }  
 else{  
 norm\_slab = kmalloc(sizeof(struct slabinfo), GFP\_KERNEL);  
 if (!norm\_slab) {  
 pr\_err("Memory allocation for slabinfo failed!\n");  
 a->status = "NOT OK";  
 vfree(a);  
 return -ENOMEM;  
 }  
  
 // sample fill slab  
 norm\_slab->active\_objs = 10;  
  
 // getting slabinfo  
  
 get\_slabinfo(my\_cache, norm\_slab);  
 m\_slb = kmalloc(sizeof(struct my\_slabinfo), GFP\_KERNEL);  
 if (!m\_slb) {  
 pr\_err("Memory allocation for my\_slabinfo failed!\n");  
 a->status = "NOT OK";  
 return -ENOMEM; // allocation error handler  
 }  
  
  
 set\_my\_type(m\_slb, norm\_slab, my\_cache);  
 a->sld =\*m\_slb;  
 a->status = "OK";  
 print\_answer(a);  
 kfree(norm\_slab);  
 kfree(m\_slb);  
 if (copy\_to\_user((struct answer \*) arg, a, sizeof(struct answer))) pr\_err("Data read error!\n");  
 }  
  
 vfree(a);  
 }  
  
 return 0;  
}  
  
  
// device file operation structure  
static struct file\_operations fops =  
{  
 .owner = THIS\_MODULE,  
 .read = status\_driver\_read,  
 .write = status\_driver\_write,  
 .open = status\_driver\_open,  
 .unlocked\_ioctl = core\_mode\_ioctl,  
 .release = status\_driver\_release,  
};  
  
  
// driver initialization  
static int \_\_init my\_driver\_init(void)  
{  
 // allocation of MYMAJOR (driver sys id)  
 if((alloc\_chrdev\_region(&dev, 0, 1, "my\_driver")) <0){  
 pr\_err("Cannot allocate major number\n");  
 return -1;  
 }  
 pr\_info("Major = %d Minor = %d \n", MAJOR(dev), MINOR(dev));  
  
 // initialization of the device  
 cdev\_init(&my\_driver\_cdev,&fops);  
  
 // adding device to /dev  
 if((cdev\_add(&my\_driver\_cdev,dev,1)) < 0){  
 pr\_err("Cannot add the device to the system\n");  
 goto r\_class;  
 }  
  
 // driver class in /sys/class  
 if(IS\_ERR(dev\_class = class\_create(THIS\_MODULE,"my\_driver\_class"))){  
 pr\_err("Cannot create the struct class\n");  
 goto r\_class;  
 }  
  
 // device creation  
 if(IS\_ERR(device\_create(dev\_class,NULL,dev,NULL, DEVICE\_NAME))){  
 pr\_err("Cannot create the Device \n");  
 goto r\_device;  
 }  
 pr\_info("Device Driver Inserted Done\n");  
 return 0;  
  
// error handler  
r\_device:  
 class\_destroy(dev\_class);  
r\_class:  
 unregister\_chrdev\_region(dev,1);  
 return -1;  
}  
  
// exit driver  
static void \_\_exit my\_driver\_exit(void)  
{  
 device\_destroy(dev\_class,dev);  
 class\_destroy(dev\_class);  
 cdev\_del(&my\_driver\_cdev);  
 unregister\_chrdev\_region(dev, 1);  
 pr\_info("Device Driver Remove...Done\n");  
}  
  
module\_init(my\_driver\_init);  
module\_exit(my\_driver\_exit)

В этом файле реализуется создание и инициализация файла устройства в папке /dev, получение нужной структуры данных, передача на уровень пользователя.

## Код пользовательского приложения user\_mode.c

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <sys/ioctl.h>  
#include <math.h>  
#include "core\_mode.h"  
  
  
static void print\_slabinfo(struct my\_slabinfo\* slb){  
 printf("Slabinfo \n");  
 printf("number of objects = %lu, \n", slb->num\_objs);  
 printf("active objects = %lu, \n", slb->active\_objs);  
 double usage = 0.0;  
 if (slb->num\_objs != 0){  
 usage = (double)slb->active\_objs / (double)slb->num\_objs \* 100.0;  
 }  
 printf("use = %.2f %%, \n", usage);  
 double obj\_kb = slb->obj\_size / 1024.0;  
 printf("object size = %.2fK\n", obj\_kb);  
 printf("number of slabs = %lu, \n", slb->num\_slabs);  
 printf("objects per slab = %u, \n", slb->objects\_per\_slab);  
 double cache\_kb = (slb->active\_objs \* slb->obj\_size)/1024.0;  
 printf("cache size = %.2fK, \n", cache\_kb);  
 printf("\n");  
}  
  
  
  
static void print\_answer(struct answer\* a){  
  
 print\_slabinfo(&(a->sld));  
  
  
}  
  
int main(int argc, char \*argv[]){  
  
 printf("Select: \n 1 - vm\_area\_struct \n 2 - dentry \n 3 - vmap\_area \n 4 - anon\_vma\_chain \n");  
 if (argc < 1) {  
 printf("The program needs an argument\n");  
 return -1;  
 }  
  
 int fd;  
 int32\_t value, number;  
  
 long numer = strtol(argv[1], NULL, 10);  
 switch (numer){  
 case 1:  
 printf("it is vm\_area\_struct");  
 break;  
 case 2:  
 printf("it is dentry");  
 break;  
 case 3:  
 printf("it is vmap\_area");  
 break;  
 case 4:  
 printf("it is anon\_vma\_chain");  
 break;  
 }  
  
  
 struct answer answer;  
  
 printf("\nOpening a driver...\n");  
 fd = open("/dev/DEVICESLABTOP", O\_WRONLY);  
 if (fd < 0) {  
 printf("Cannot open device file\n");  
 return 0;  
 }  
  
  
 answer.st\_num = numer;  
  
 ioctl(fd, RW\_ANS, (struct answer \*) &answer);  
  
 print\_answer(&answer);  
  
 printf("Closing Driver...\n");  
 close(fd);  
}

Непосредственно пользовательская программа.

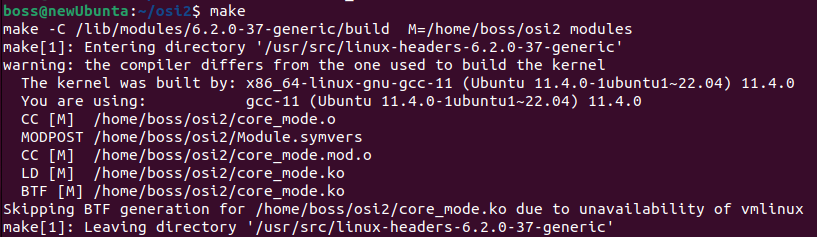
## Файл сборки модуля Makefile

KERNEL\_SRC := /usr/src/linux-hwe-6.2-headers-6.2.0-37  
CFLAGS += -msse2  
  
obj-m += core\_mode.o  
  
KDIR = /lib/modules/$(shell uname -r)/build  
  
  
all:  
 make -C $(KDIR) M=$(shell pwd) modules  
  
clean:  
 make -C $(KDIR) M=$(shell pwd) clean

В данном файле собирается модуль с использованием исходного кода ядра (моя версия 6.2.0.37-generic).

# Установка и запуск модуля

1. В директории проекта запускается команда make:



1. Модуль устанавливается в ядро командой sudo insmod core\_mode.ko:

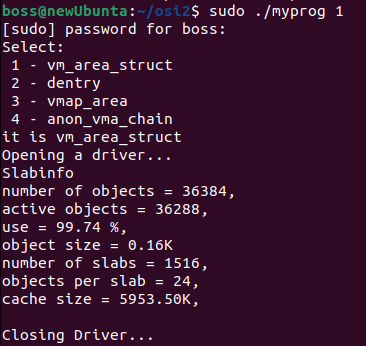


1. Пользовательское приложение собирается командой:

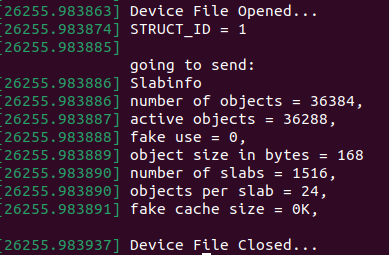
gcc user\_mode.c -o myprog:



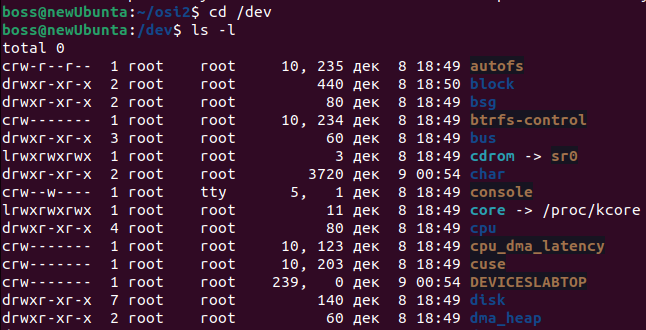
# Пример работы комплекса программ:



Лог программы (команда dmesg)



Отображение файла устройства



# Вывод

В ходе выполнения лабораторной работы были изучены структуры файлов в операционной системе Linux, реализован модуль ядра с интерфейсом ioctl и сымитирован вывод утилиты slabtop -o через структуры kmem\_cache и slabinfo без использования файла /proc/slabinfo.