Федеральное государственное автономное образовательное учреждение высшего образования "Национальный Исследовательский Университет ИТМО"
Мегафакультет Компьютерных Технологий и Управления
Факультет Программной Инженерии и Компьютерной Техники

Лабораторная работа №2

по дисциплине

'Операционные системы'

Выполнил Студент группы P33102
Лапин Алексей Александрович
Преподаватель:
Осипов Святослав Владимирович

г. Санкт-Петербург
2023г.

Содержание

1 Текст задания:

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 между программой пользователя и ядром и целевая структура задается преподавателем. Интерфейс передачи может быть один из следующих:

1. syscall - интерфейс системных вызовов.
2. ioctl - передача параметров через управляющий вызов к файлу/устройству.
3. procfs - файловая система /proc, передача параметров через запись в файл.
4. debugfs - отладочная файловая система /sys/kernel/debug, передача параметров через запись в файл.

Целевая структура может быть задана двумя способами:

1. Именем структуры в заголовочных файлах Linux
2. Файлом в каталоге /proc. В этом случае необходимо определить целевую структуру по пути файла в /proc и выводимым данным.

2 Код модуля ядра:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/ioctl.h>
#include <linux/kernel_stat.h>
#include <linux/types.h> /* u64 */
#include <linux/cpumask.h>
#include <linux/vmalloc.h>
#include <linux/tick.h>
#include <linux/jiffies.h>
#include <linux/time64.h>
#include <linux/math64.h>
```

```c
#include <linux/utsname.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Aleksei Lapin");
MODULE_DESCRIPTION("Simple module to start");

struct ioctl_arg{
    u64 val;
};

static struct cpustat {
    u64 user;
    u64 nice;
    u64 system;
    u64 idle;
    u64 iowait;
    u64 irq;
    u64 softirq;
    u64 steal;
    u64 guest;
    u64 guest_nice;
};

struct os_stat{
    char sysname[__NEW_UTS_LEN + 1];
    char nodename[__NEW_UTS_LEN + 1];
    char release[__NEW_UTS_LEN + 1];
    char version[__NEW_UTS_LEN + 1];
    char machine[__NEW_UTS_LEN + 1];
    char domainname[__NEW_UTS_LEN + 1];
};

/* Documentation/ioctl/ioctl-number.txt */
#define IOC_MAGIC 'a'

#define GET_OS_STAT _IOR(IOC_MAGIC, 0, struct os_stat)
#define SET_CPU _IOW(IOC_MAGIC, 1, struct ioctl_arg)
#define GET_CPU_STAT_BY_NUM _IOR(IOC_MAGIC, 2, struct cpustat)
#define GET_ONLINE_CPU_NUM _IOR(IOC_MAGIC, 3, struct ioctl_arg)
#define GET_POSSIBLE_CPU_NUM _IOR(IOC_MAGIC, 4, struct ioctl_arg)
#define GET_CPU_STAT_ALL _IOR(IOC_MAGIC, 5, struct cpustat*)

#define DEVICE_NAME "os_lab"


#define NSEC_TO_SEC(nsec) (nsec) / 1000000000u
static dev_t major = 0; /* The major number assigned to the device
   driver */
static struct class *dev_class;
static struct cdev cs_dev;

static u64 cpu = 0;

```

```
71 static int __init cpu_stat_init(void);
72 static void __exit cpu_stat_exit(void);
73 static ssize_t cs_read(struct file *filp, char __user *buf, size_t
     len, loff_t *off);
74 static ssize_t cs_write(struct file *filp, const char __user *buf,
     size_t len, loff_t *off);
75 static int cs_open(struct inode *inode, struct file *file);
76 static int cs_release(struct inode *inode, struct file *file);
77 static long cs_ioctl(struct file *file, unsigned int cmd, unsigned
     long arg);
78 static u64 get_possible_cpu_num(void);
79 static u64 get_online_cpu_num(void);
80 static struct cpustat* get_cpustat_by_num(u64 cpu_num);
81 static struct cpustat* get_cpustat_all(void);
82
83
84
85 static struct file_operations fops = {
86     .owner          = THIS_MODULE,
87     .read           = cs_read,
88     .write          = cs_write,
89     .open           = cs_open,
90     .release        = cs_release,
91     .unlocked_ioctl = cs_ioctl,
92 };
93
94 static ssize_t cs_read(struct file *filp, char __user *buf, size_t
     len, loff_t *off){
95     pr_info("Reading from device.\n");
96     return 0;
97 }
98
99 static ssize_t cs_write(struct file *filp, const char __user *buf,
     size_t len, loff_t *off){
100    pr_info("Writing to device.\n");
101    return 0;
102 }
103
104 static int cs_open(struct inode *inode, struct file *file){
105    pr_info("Device opened.\n");
106    return 0;
107 }
108
109 static int cs_release(struct inode *inode, struct file *file){
110    pr_info("Device closed.\n");
111    return 0;
112 }
113
114 static long cs_ioctl(struct file *file, unsigned int cmd, unsigned
     long arg){
115    struct ioctl_arg cpu_num;
116    switch (cmd) {
117        case GET_ONLINE_CPU_NUM:{
```

```
118             cpu_num.val = get_online_cpu_num();
119             if(copy_to_user((struct ioctl_arg*) arg, &cpu_num,
                   sizeof(struct ioctl_arg))){
120                 pr_err("Fail to copy to user space.");
121             }
122             return 0;
123         }
124         case GET_POSSIBLE_CPU_NUM: {
125             cpu_num.val = get_possible_cpu_num();
126             if(copy_to_user((struct ioctl_arg*) arg, &cpu_num,
                   sizeof(struct ioctl_arg))){
127                 pr_err("Fail to copy to user space.");
128             }
129             return 0;
130         }
131         case GET_CPU_STAT_BY_NUM:{
132             struct cpustat* stat = NULL;
133             stat = get_cpustat_by_num(cpu);
134             if(copy_to_user((struct cpustat*) arg, stat,
                   sizeof(struct cpustat))){
135                 pr_err("Fail to copy to user space.");
136             }
137             return 0;
138         }
139         case GET_CPU_STAT_ALL:{
140             struct cpustat* stat = NULL;
141             u64 buffer_size = (get_possible_cpu_num() + 1) *
                   sizeof(struct cpustat);
142             stat = get_cpustat_all();
143             if(copy_to_user((struct cpustat*) arg, stat,
                   buffer_size)){
144                 pr_err("Fail to copy to user space.");
145             }
146             return 0;
147 
148         }
149         case GET_OS_STAT: {
150             struct os_stat os_stat;
151             struct new_utsname *nutsname = utsname();
152             strcpy(os_stat.sysname, nutsname->sysname);
153             strcpy(os_stat.nodename, nutsname->nodename);
154             strcpy(os_stat.release, nutsname->release);
155             strcpy(os_stat.version, nutsname->version);
156             strcpy(os_stat.machine, nutsname->machine);
157             strcpy(os_stat.domainname, nutsname->domainname);
158             if(copy_to_user((struct os_stat*) arg, &os_stat,
                   sizeof(struct os_stat))){
159                 pr_err("Fail to copy to user space.");
160             }
161             return 0;
162         }
163         case SET_CPU: {
```

```
164             if(copy_from_user(&cpu_num, (struct ioctl_arg*) arg,
                    sizeof(struct ioctl_arg))){
165                 pr_err("Fail to copy to kernel space.");
166             }
167             cpu = cpu_num.val;
168             return 0;
169         }
170         default:
171             pr_err("Unknown command\n");
172     }
173     return 0;
174 }
175
176 static u64 get_possible_cpu_num(void){
177     u64 cpu_num = num_present_cpus();
178     return cpu_num;
179 }
180
181 static u64 get_online_cpu_num(void){
182     u64 cpu_num = num_online_cpus();
183     return cpu_num;
184 }
185
186 u64 cs_nsec_to_clock_t(u64 x)
187 {
188 #if (NSEC_PER_SEC % USER_HZ) == 0
189     return div_u64(x, NSEC_PER_SEC / USER_HZ);
190 #elif (USER_HZ % 512) == 0
191     return div_u64(x * USER_HZ / 512, NSEC_PER_SEC / 512);
192 #else
193     /*
194          * max relative error 5.7e-8 (1.8s per year) for USER_HZ <=
                1024,
195          * overflow after 64.99 years.
196          * exact for HZ=60, 72, 90, 120, 144, 180, 300, 600, 900, ...
197          */
198     return div_u64(x * 9, (9ull * NSEC_PER_SEC + (USER_HZ / 2)) /
           USER_HZ);
199 #endif
200 }
201
202 #ifdef arch_idle_time
203 static u64 cs_get_idle_time(struct kernel_cpustat *kcs, int cpu)
204 {
205     u64 idle;
206
207     idle = kcs->cpustat[CPUTIME_IDLE];
208     if (cpu_online(cpu) && !nr_iowait_cpu(cpu))
209         idle += arch_idle_time(cpu);
210     return idle;
211 }
212
213 static u64 cs_get_iowait_time(struct kernel_cpustat *kcs, int cpu)
```

```
{
    u64 iowait;

    iowait = kcs->cpustat[CPUTIME_IOWAIT];
    if (cpu_online(cpu) && nr_iowait_cpu(cpu))
        iowait += arch_idle_time(cpu);
    return iowait;
}

#else

static u64 cs_get_idle_time(struct kernel_cpustat *kcs, int cpu)
{
    u64 idle, idle_usecs = -1ULL;

    if (cpu_online(cpu))
        idle_usecs = get_cpu_idle_time_us(cpu, NULL);

    if (idle_usecs == -1ULL)
        /* !NO_HZ or cpu offline so we can rely on cpustat.idle */
        idle = kcs->cpustat[CPUTIME_IDLE];
    else
        idle = idle_usecs * NSEC_PER_USEC;

    return idle;
}

static u64 cs_get_iowait_time(struct kernel_cpustat *kcs, int cpu)
{
    u64 iowait, iowait_usecs = -1ULL;

    if (cpu_online(cpu))
        iowait_usecs = get_cpu_iowait_time_us(cpu, NULL);

    if (iowait_usecs == -1ULL)
        /* !NO_HZ or cpu offline so we can rely on cpustat.iowait */
        iowait = kcs->cpustat[CPUTIME_IOWAIT];
    else
        iowait = iowait_usecs * NSEC_PER_USEC;

    return iowait;
}

#endif

static struct cpustat* get_cpustat_by_num(u64 cpu_num){
    struct kernel_cpustat *kcs = &kcpustat_cpu(cpu_num);
    struct cpustat* stat = vmalloc(sizeof(struct cpustat));
    if (!stat) {
        pr_err("vmalloc failed.\n");
        return NULL;
    }
    stat->user = cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_USER]);
```

```
267     stat->nice = cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_NICE]);
268     stat->system = cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_SYSTEM]);
269     stat->idle = cs_nsec_to_clock_t(cs_get_idle_time(kcs, cpu_num));
270     stat->iowait = cs_nsec_to_clock_t(cs_get_iowait_time(kcs,
            cpu_num));
271     stat->irq = cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_IRQ]);
272     stat->softirq =
            cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_SOFTIRQ]);
273     stat->steal = cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_STEAL]);
274     stat->guest = cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_GUEST]);
275     stat->guest_nice =
            cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_GUEST_NICE]);
276     return stat;
277 }
278
279 static struct cpustat* get_cpustat_all(void){
280     int i = 0;
281     u64 cpu_num = get_online_cpu_num() + 1;
282     struct cpustat* stat_array = vmalloc(cpu_num * sizeof(struct
            cpustat));
283     if (!stat_array) {
284         pr_err("vmalloc failed.\n");
285         return NULL;
286     }
287     for_each_possible_cpu(i) {
288         struct kernel_cpustat *kcs = &kcpustat_cpu(i);
289         stat_array[0].user +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_USER]);
290         stat_array[0].nice +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_NICE]);
291         stat_array[0].system +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_SYSTEM]);
292         stat_array[0].idle +=
                cs_nsec_to_clock_t(cs_get_idle_time(kcs, cpu_num));
293         stat_array[0].iowait +=
                cs_nsec_to_clock_t(cs_get_iowait_time(kcs, cpu_num));
294         stat_array[0].irq +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_IRQ]);
295         stat_array[0].softirq +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_SOFTIRQ]);
296         stat_array[0].steal +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_STEAL]);
297         stat_array[0].guest +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_GUEST]);
298         stat_array[0].guest_nice +=
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_GUEST_NICE]);
299     }
300
301     for_each_online_cpu(i) {
302
303         struct kernel_cpustat *kcs = &kcpustat_cpu(i);
304         stat_array[i+1].user =
                cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_USER]);
```

```
305         stat_array[i+1].nice =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_NICE]);
306         stat_array[i+1].system =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_SYSTEM]);
307         stat_array[i+1].idle =
               cs_nsec_to_clock_t(cs_get_idle_time(kcs, i));
308         stat_array[i+1].iowait =
               cs_nsec_to_clock_t(cs_get_iowait_time(kcs, i));
309         stat_array[i+1].irq =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_IRQ]);
310         stat_array[i+1].softirq =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_SOFTIRQ]);
311         stat_array[i+1].steal =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_STEAL]);
312         stat_array[i+1].guest =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_GUEST]);
313         stat_array[i+1].guest_nice =
               cs_nsec_to_clock_t(kcs->cpustat[CPUTIME_GUEST_NICE]);
314     }
315     return stat_array;
316 }
317
318
319
320
321 static int __init cpu_stat_init(void){
322     pr_info("cpu_stat: Module loaded\n");
323
324     /* Allocating major numbers */
325     if(alloc_chrdev_region(&major, 0, 1, DEVICE_NAME) < 0){
326         pr_err("Cannot allocate major numbers.\n");
327         return -1;
328     }
329
330     /* cdev structure initialization */
331     cdev_init(&cs_dev, &fops);
332
333     /* Adding device to the system */
334     if(cdev_add(&cs_dev, major, 1) < 0){
335         pr_err("Cannot add the device to the system.\n");
336         goto rm_major;
337     }
338
339     /* Creating structure class */
340     if((dev_class = class_create(THIS_MODULE, DEVICE_NAME)) == NULL)
           {
341         pr_err("Cannot create the structure class.\n");
342         goto rm_major;
343     }
344
345     if(device_create(dev_class, NULL, major, NULL, DEVICE_NAME) < 0){
346         pr_err("Cannot create the device");
347         goto rm_class;
```

```
    }

    pr_info("Device created on /dev/%s\n", DEVICE_NAME);

    return 0;

rm_class:
    class_destroy(dev_class);
rm_major:
    unregister_chrdev_region(major, 1);

    return -1;
}

static void __exit cpu_stat_exit(void){
    device_destroy(dev_class, major);
    class_destroy(dev_class);
    cdev_del(&cs_dev);
    unregister_chrdev_region(major, 1);
    pr_info("cpu_stat: Module unloaded.\n");
}

module_init(cpu_stat_init);
module_exit(cpu_stat_exit);
```

3 Код программы пользователя:

```
#include "common.h"

#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <inttypes.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define IOC_MAGIC 'a'

#define GET_OS_STAT _IOR(IOC_MAGIC, 0, struct os_stat)
#define SET_CPU _IOW(IOC_MAGIC, 1, struct ioctl_arg)
#define GET_CPU_STAT_BY_NUM _IOR(IOC_MAGIC, 2, struct cpustat)
#define GET_ONLINE_CPU_NUM _IOR(IOC_MAGIC, 3, struct ioctl_arg)
#define GET_POSSIBLE_CPU_NUM _IOR(IOC_MAGIC, 4, struct ioctl_arg)
#define GET_CPU_STAT_ALL _IOR(IOC_MAGIC, 5, struct cpustat*)


#define DEVICE_PATH "/dev/os_lab"
#define F_OPT_P 0x01
#define F_OPT_INTERVAL 0x02

```

```
26
27 struct ioctl_arg{
28     uint64_t val;
29 };
30
31 struct cpustat {
32     uint64_t user;
33     uint64_t nice;
34     uint64_t system;
35     uint64_t idle;
36     uint64_t iowait;
37     uint64_t irq;
38     uint64_t softirq;
39     uint64_t steal;
40     uint64_t guest;
41     uint64_t guest_nice;
42 };
43
44 struct cpu_percent_stat {
45     double user;
46     double nice;
47     double system;
48     double idle;
49     double iowait;
50     double irq;
51     double softirq;
52     double steal;
53     double guest;
54     double guest_nice;
55 };
56
57
58 #define __NEW_UTS_LEN 64
59
60 struct os_stat{
61     char sysname[__NEW_UTS_LEN + 1];
62     char nodename[__NEW_UTS_LEN + 1];
63     char release[__NEW_UTS_LEN + 1];
64     char version[__NEW_UTS_LEN + 1];
65     char machine[__NEW_UTS_LEN + 1];
66     char domainname[__NEW_UTS_LEN + 1];
67 };
68
69
70
71 uint64_t possible_cpu = 0;
72 uint8_t* cpu_mask = NULL;
73 unsigned int opt_flags = 0;
74
75 #define KEY_all "all"
76
77
78 void usage(char *progname)
```

```
79 {
80     fprintf(stderr, "Usage: %s [ options ][ <interval> [ <count> ]
       ]\n",
81         progname);
82
83     fprintf(stderr, "Options are:\n"
84                 "[ -P { <cpu_list> | ALL } ]\n");
85     exit(1);
86 }
87
88
89 unsigned long long* get_global_cpu_mpstats(struct cpustat*
    cpus_stat, uint64_t possible_cpus){
90     unsigned long long* tot_jiffies = malloc(sizeof(unsigned long
       long) * (possible_cpus + 1));
91     if(!tot_jiffies){
92         printf("Unable to malloc tot_jiffies.");
93         return NULL;
94     }
95     for (int i = 0; i < possible_cpus + 1; i++){
96         tot_jiffies[i] = cpus_stat[i].user + cpus_stat[i].nice +
              cpus_stat[i].system
97         + cpus_stat[i].idle + cpus_stat[i].iowait + cpus_stat[i].irq
              + cpus_stat[i].softirq
98         + cpus_stat[i].steal + cpus_stat[i].guest +
              cpus_stat[i].guest_nice;
99     }
100     return tot_jiffies;
101
102 }
103
104 struct cpu_percent_stat* get_percentage_stat(struct cpustat*
    cpus_stat, uint64_t possible_cpus, unsigned long long*
    tot_jiffies){
105     struct cpu_percent_stat* new_stat = malloc((possible_cpus + 1) *
       sizeof(struct cpu_percent_stat));
106     if(!new_stat){
107         printf("New cpu stat is NULL");
108         return NULL;
109     }
110     for(int i = 0; i < possible_cpus + 1; i++){
111         new_stat[i].user = (double)(cpus_stat[i].user * 100) /
              tot_jiffies[i];
112         new_stat[i].system = (double)(cpus_stat[i].system * 100) /
              tot_jiffies[i];
113         new_stat[i].nice = (double)(cpus_stat[i].nice * 100) /
              tot_jiffies[i];
114         new_stat[i].idle = (double)(cpus_stat[i].idle * 100) /
              tot_jiffies[i];
115         new_stat[i].iowait = (double)(cpus_stat[i].iowait * 100) /
              tot_jiffies[i];
116         new_stat[i].irq = (double)(cpus_stat[i].irq * 100) /
              tot_jiffies[i];
```

```
117         new_stat[i].softirq = (double)(cpus_stat[i].softirq * 100) /
               tot_jiffies[i];
118         new_stat[i].steal = (double)(cpus_stat[i].steal * 100) /
               tot_jiffies[i];
119         new_stat[i].guest = (double)(cpus_stat[i].guest * 100) /
               tot_jiffies[i];
120         new_stat[i].guest_nice = (double)(cpus_stat[i].guest_nice *
               100) / tot_jiffies[i];
121     }
122     return new_stat;
123 }
124
125
126 void write_os_stat(struct os_stat* os_stat){
127     printf("OS Information:\n");
128     printf("sysname: %s\n", os_stat->sysname);
129     printf("nodename: %s\n", os_stat->nodename);
130     printf("release: %s\n", os_stat->release);
131     printf("version: %s\n", os_stat->version);
132     printf("machine: %s\n", os_stat->machine);
133     printf("domainname: %s\n", os_stat->domainname);
134 };
135
136 void write_header(struct os_stat* os_stat, uint64_t possible_cpus){
137     char date_str[10];
138     get_current_date(date_str);
139     printf("%s %s (%s)  %s      _%s_        (%"PRIu64" CPU)\n",
           os_stat->sysname, os_stat->release, os_stat->nodename,
           date_str,  os_stat->machine, possible_cpus);
140     printf("\n");
141 }
142
143 void write_cpu_percent_stat(struct cpu_percent_stat* cpus_stat,
       uint64_t possible_cpus){
144     char time_str[10];
145     get_current_time(time_str);
146     printf("%s\t", time_str);
147
148     printf("CPU\t%%usr\t%%nice\t%%sys\t%%iowait\t%%irq "
149                "\t%%soft\t%%steal\t%%guest\t%%gnice\t%%idle\n");
150     for(int i = 0; i < possible_cpus + 1; i++){
151         if(!cpu_mask[i]) continue;
152         get_current_time(time_str);
153         printf("%s\t", time_str);
154
155         if(i == 0){
156             printf("all\t");
157         }
158         else printf("%d\t", (i - 1));
159
160         printf("%.2f\t%.2f\t%.2f\t%.2f\t%.2f"
161         "    %.2f\t%.2f\t%.2f\t%.2f\t%.2f\n",
162             cpus_stat[i].user,
```

```
163             cpus_stat[i].nice,
164             cpus_stat[i].system,
165             cpus_stat[i].iowait,
166             cpus_stat[i].irq,
167             cpus_stat[i].softirq,
168             cpus_stat[i].steal,
169             cpus_stat[i].guest,
170             cpus_stat[i].guest_nice,
171             cpus_stat[i].idle
172         );
173     }
174 }
175
176
177 void write_cpu_stat(struct cpustat* cpus_stat, uint64_t
    possible_cpus){
178     char time_str[10];
179     get_current_time(time_str);
180     printf("%s\t", time_str);
181
182
183
184     printf("CPU\t%%usr   %%nice    %%sys %%iowait  %%irq "
185                "%%soft  %%steal  %%guest  %%gnice   %%idle\n");
186     for(int i = 0; i < possible_cpus + 1; i++){
187         get_current_time(time_str);
188         printf("%s\t", time_str);
189         if(i == 0){
190             printf("all\t");
191         }
192         else printf("%d\t", (i - 1));
193
194         printf("%"PRIu64"\t%"PRIu64"\t%"PRIu64"\t%"PRIu64"\t%"PRIu64""
195         "
               %"PRIu64"\t\t%"PRIu64"\t%"PRIu64"\t%"PRIu64"\t%"PRIu64"\n",
196             cpus_stat[i].user,
197             cpus_stat[i].nice,
198             cpus_stat[i].system,
199             cpus_stat[i].iowait,
200             cpus_stat[i].irq,
201             cpus_stat[i].softirq,
202             cpus_stat[i].steal,
203             cpus_stat[i].guest,
204             cpus_stat[i].guest_nice,
205             cpus_stat[i].idle
206         );
207     }
208 }
209
210 struct cpustat* cpu_diff(struct cpustat* cpu_prev, struct cpustat*
    cpu_current, uint64_t possible_cpu){
211     struct cpustat* cpu_diff = malloc((possible_cpu + 1) *
        sizeof(struct cpustat));
```

```
212     if(!cpu_diff){
213         printf("Unable to malloc cpu_diff");
214         return NULL;
215     }
216     for(int i = 0; i < possible_cpu + 1; i++){
217         cpu_diff[i].user = cpu_current[i].user - cpu_prev[i].user;
218         cpu_diff[i].nice = cpu_current[i].nice - cpu_prev[i].nice;
219         cpu_diff[i].system = cpu_current[i].system -
              cpu_prev[i].system;
220         cpu_diff[i].idle = cpu_current[i].idle - cpu_prev[i].idle;
221         cpu_diff[i].iowait = cpu_current[i].iowait -
              cpu_prev[i].iowait;
222         cpu_diff[i].irq = cpu_current[i].irq - cpu_prev[i].irq;
223         cpu_diff[i].softirq = cpu_current[i].softirq -
              cpu_prev[i].softirq;
224         cpu_diff[i].steal = cpu_current[i].steal - cpu_prev[i].steal;
225         cpu_diff[i].guest = cpu_current[i].guest - cpu_prev[i].guest;
226         cpu_diff[i].guest_nice = cpu_current[i].guest_nice -
              cpu_prev[i].guest_nice;
227     }
228     return cpu_diff;
229 }
230
231
232
233 int loop(int driver, int interval, int count, uint64_t possible_cpu){
234     struct cpustat* cpus_stat_p = malloc((possible_cpu + 1) *
          sizeof(struct cpustat));
235     if(cpus_stat_p == NULL){
236         printf("CPUS_STAT is NULL");
237         return -1;
238     }
239     struct cpustat* cpus_stat_c = malloc((possible_cpu + 1) *
          sizeof(struct cpustat));
240     if(cpus_stat_c == NULL){
241         printf("CPUS_STAT is NULL");
242         return -1;
243     }
244     struct os_stat* os_stat = malloc(sizeof(struct os_stat));
245     ioctl(driver, GET_OS_STAT, os_stat);
246     write_header(os_stat, possible_cpu);
247     int i = -1;
248     ioctl(driver, GET_CPU_STAT_ALL, cpus_stat_p);
249     while(1){
250         if(++i == count) break;
251         sleep(interval);
252         ioctl(driver, GET_CPU_STAT_ALL, cpus_stat_c);
253         struct cpustat* diff_stat = cpu_diff(cpus_stat_p,
              cpus_stat_c, possible_cpu);
254         unsigned long long* tot_jiffies =
              get_global_cpu_mpstats(diff_stat, possible_cpu);
255         struct cpu_percent_stat* cpus_percent_stat =
              get_percentage_stat(diff_stat, possible_cpu, tot_jiffies);
```

```
256         write_cpu_percent_stat(cpus_percent_stat, possible_cpu);
257         free(diff_stat);
258         free(tot_jiffies);
259         free(cpus_percent_stat);
260         memcpy(cpus_stat_p, cpus_stat_c, (possible_cpu + 1) *
               sizeof(struct cpustat));
261     }
262     free(cpus_stat_c);
263     free(cpus_stat_p);
264     free(os_stat);
265     close(driver);
266     return 0;
267 }
268
269
270 int main(int argc, char *argv[]){
271     int driver = open(DEVICE_PATH, O_RDWR);
272     if(driver < 0){
273         printf("Fail to open device.\n");
274         return -1;
275     }
276
277
278
279     struct ioctl_arg umsg;
280     ioctl(driver, GET_POSSIBLE_CPU_NUM, &umsg);
281     uint64_t possible_cpu = umsg.val;
282
283     int opt = 0;
284     int interval = -1;
285     int count = -1;
286     cpu_mask = malloc((possible_cpu + 1));
287     if(!cpu_mask){
288         printf("Fail to malloc cpu_mask");
289         return -1;
290     }
291     memset(cpu_mask, 0, possible_cpu + 1);
292
293     while(++opt < argc){
294         if(!strcmp(argv[opt], "-P")){
295             if (!argv[++opt]) {
296                 usage(argv[0]);
297             }
298             if(parse_values(argv[opt], cpu_mask, possible_cpu ,
                   KEY_all) < 0){
299                 printf("Fail to parse cpu mask");
300                 return -1;
301             }
302             opt_flags |= F_OPT_P;
303         }
304
305         else if(interval < 0){
```

```
306             if (strspn(argv[opt], "0123456789") !=
                   strlen(argv[opt])) {
307                     usage(argv[0]);
308                 }
309             interval = atoi(argv[opt]);
310             if (interval < 0) {
311                 usage(argv[0]);
312             }
313             opt_flags |= F_OPT_INTERVAL;
314         }
315         else if(count < 0){
316             if (strspn(argv[opt], "0123456789") != strlen(argv[opt])
                   || !interval) {
317                     usage(argv[0]);
318                 }
319             count = atoi(argv[opt]);
320             if (count < 1) {
321                 usage(argv[0]);
322             }
323         }
324     }
325     if(!(opt_flags & F_OPT_P)){
326         cpu_mask[0] = 1;
327     }
328     if(interval > 0 && (count > 0 || count == -1)){
329         loop(driver, interval, count, possible_cpu);
330     }
331     else{
332         struct cpustat* cpus_stat = malloc((possible_cpu + 1) *
               sizeof(struct cpustat));
333         if(cpus_stat == NULL){
334             printf("CPUS_STAT is NULL");
335             return -1;
336         }
337         struct os_stat* os_stat = malloc(sizeof(struct os_stat));
338         ioctl(driver, GET_OS_STAT, os_stat);
339         write_header(os_stat, possible_cpu);
340         ioctl(driver, GET_CPU_STAT_ALL, cpus_stat);
341         unsigned long long* tot_jiffies =
               get_global_cpu_mpstats(cpus_stat, possible_cpu);
342         struct cpu_percent_stat* cpus_percent_stat =
               get_percentage_stat(cpus_stat, possible_cpu, tot_jiffies);
343         write_cpu_percent_stat(cpus_percent_stat, possible_cpu);
344         free(tot_jiffies);
345         free(cpus_percent_stat);
346         free(cpus_stat);
347         free(os_stat);
348     }
349     close(driver);
350     return 0;
351 }
```

```
1 #include "common.h"
```

```c
#include <time.h>
#include <stdio.h>
#include <inttypes.h>
#include <string.h>
#include <stdint.h>
#include <stdlib.h>

#define KEY_ALL "ALL"

void get_current_time(char* time_str){
    time_t rawtime;
    struct tm * timeinfo;

    time(&rawtime);
    timeinfo = localtime(&rawtime);

    sprintf(time_str, "%02d:%02d:%02d", timeinfo->tm_hour,
       timeinfo->tm_min, timeinfo->tm_sec);
}

void get_current_date(char* date_str){
    time_t rawtime;
    struct tm * timeinfo;

    time(&rawtime);
    timeinfo = localtime(&rawtime);

    sprintf(date_str, "%02d.%02d.%04d", timeinfo->tm_mday,
       timeinfo->tm_mon + 1, timeinfo->tm_year + 1900);
}

int parse_values_range(char* t, int max_value, int* val_low, int*
   val ){
    char *s, *valstr, range[16];

    strncpy(range, t, 16);
    range[15] = '\0';
    s = strchr(range, '-');
    valstr = t;
    if (s) {
        *s = '\0';
        valstr = s + 1;
        *val_low = atoi(range);
        *val = atoi(valstr);
        if (*val_low < 0 || *val_low >= max_value || *val < 0 ||
           *val >= max_value) {
            return -1;
        }
    } else {
        *val_low = *val = atoi(range);
        if (*val_low < 0 || *val_low >= max_value) {
            return -1;
        }
```

```
    }
}

int parse_values(char* strargv, uint8_t* cpu_mask, int max_value,
    const char * KEY_WORD ){
    int i, val_low, val;
    char *t;

    if (!strcmp(strargv, KEY_ALL)) {
        memset(cpu_mask, ~0, max_value);
        return 0;
    }
    for (t = strtok(strargv, ","); t; t = strtok(NULL, ",")){
        if(!strcmp(t, KEY_WORD)){
            cpu_mask[0] = 1;
        }
        else {
            if (parse_values_range(t, max_value, &val_low, &val) <
                0) {
                return -1;
            }
            for(i = val_low; i <= val; i++){
                cpu_mask[i + 1] = 1;
            }
        }
    }
    return 0;

}
```

4 Результаты работы программы:

```
└─ sudo ./my_mpstat -P 1-4 3 3
Linux 6.2.0-39-generic (alexey-ubuntu)  18.12.2023      _x86_64_        (8 CPU)

10:59:25        CPU     %usr    %nice   %sys    %iowait %irq    %soft   %steal  %guest  %gnice  %idle
10:59:25        1       2.02    0.00    2.69    0.00    0.00    0.00    0.00    0.00    0.00    95.29
10:59:25        2       3.01    0.00    2.01    0.00    0.00    0.00    0.00    0.00    0.00    94.98
10:59:25        3       2.34    0.00    2.01    0.00    0.00    0.00    0.00    0.00    0.00    95.65
10:59:25        4       2.00    0.00    1.67    0.00    0.00    0.00    0.00    0.00    0.00    96.33
10:59:28        CPU     %usr    %nice   %sys    %iowait %irq    %soft   %steal  %guest  %gnice  %idle
10:59:28        1       1.99    0.00    1.66    0.00    0.00    0.33    0.00    0.00    0.00    96.01
10:59:28        2       2.68    0.00    1.34    0.00    0.00    0.00    0.00    0.00    0.00    95.99
10:59:28        3       1.72    0.00    2.07    0.00    0.00    0.00    0.00    0.00    0.00    96.21
10:59:28        4       1.34    0.00    1.01    0.00    0.00    0.00    0.00    0.00    0.00    97.65
10:59:31        CPU     %usr    %nice   %sys    %iowait %irq    %soft   %steal  %guest  %gnice  %idle
10:59:31        1       1.35    0.00    1.35    0.00    0.00    0.00    0.00    0.00    0.00    97.30
10:59:31        2       1.68    0.00    1.68    0.00    0.00    0.00    0.00    0.00    0.00    96.64
10:59:31        3       0.68    0.00    1.69    0.00    0.00    0.00    0.00    0.00    0.00    97.64
10:59:31        4       1.36    0.00    1.02    0.00    0.00    0.00    0.00    0.00    0.00    97.62
```

5 Вывод:

В ходе выполнения лабораторной работы я получил практические навыки по разработке комплекса программ на пользовательском уровне и уровне ядра. Этот комплекс программ собирает информацию на стороне ядра, передает ее на уровень пользователя и выводит в удобном для чтения виде. Для этого был разработан загружаемый модуль

ядра и программа на уровне пользователя, которые взаимодействуют через интерфейс передачи **ioctl**. В результате выполнения лабораторной работы я получил практический опыт работы с ядром операционной системы.