

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 Р33102
Лапин Алексей Александрович
Преподаватель:
Осипов Святослав Владимирович

г. Санкт-Петербург
2023г.

Содержание

1	Текст задания:	3
2	Код модуля ядра:	3
3	Код программы пользователя:	11
4	Результаты работы программы:	20
5	Вывод:	20

1 Текст задания:

Разработать комплекс программ на пользовательском уровне и уровне ядра, 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:

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/init.h>
4
5 #include <linux/fs.h>
6 #include <linux/proc_fs.h>
7 #include <linux/cdev.h>
8 #include <linux/uaccess.h>
9 #include <linux/slab.h>
10 #include <linux/ioctl.h>
11 #include <linux/kernel_stat.h>
12 #include <linux/types.h> /* u64 */
13 #include <linux/cpumask.h>
14 #include <linux/vmalloc.h>
15 #include <linux/tick.h>
16 #include <linux/jiffies.h>
17 #include <linux/time64.h>
18 #include <linux/math64.h>
```

```

19 #include <linux/utsname.h>
20
21 MODULE_LICENSE("GPL");
22 MODULE_AUTHOR("Aleksei Lapin");
23 MODULE_DESCRIPTION("Simple module to start");
24
25 struct ioctl_arg{
26     u64 val;
27 };
28
29 static struct cpustat {
30     u64 user;
31     u64 nice;
32     u64 system;
33     u64 idle;
34     u64 iowait;
35     u64 irq;
36     u64 softirq;
37     u64 steal;
38     u64 guest;
39     u64 guest_nice;
40 };
41
42 struct os_stat{
43     char sysname[__NEW_UTS_LEN + 1];
44     char nodename[__NEW_UTS_LEN + 1];
45     char release[__NEW_UTS_LEN + 1];
46     char version[__NEW_UTS_LEN + 1];
47     char machine[__NEW_UTS_LEN + 1];
48     char domainname[__NEW_UTS_LEN + 1];
49 };
50
51 /* Documentation/ioctl/ioctl-number.txt */
52 #define IOC_MAGIC 'a'
53
54 #define GET_OS_STAT _IOR(IOC_MAGIC, 0, struct os_stat)
55 #define SET_CPU _IOW(IOC_MAGIC, 1, struct ioctl_arg)
56 #define GET_CPU_STAT_BY_NUM _IOR(IOC_MAGIC, 2, struct cpustat)
57 #define GET_ONLINE_CPU_NUM _IOR(IOC_MAGIC, 3, struct ioctl_arg)
58 #define GET_POSSIBLE_CPU_NUM _IOR(IOC_MAGIC, 4, struct ioctl_arg)
59 #define GET_CPU_STAT_ALL _IOR(IOC_MAGIC, 5, struct cpustat*)
60
61 #define DEVICE_NAME "os_lab"
62
63
64 #define NSEC_TO_SEC(nsec) (nsec) / 1000000000u
65 static dev_t major = 0; /* The major number assigned to the device
66     driver */
67 static struct class *dev_class;
68 static struct cdev cs_dev;
69
70 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,
120             sizeof(struct ioctl_arg))) {
121             pr_err("Fail to copy to user space.");
122         }
123         return 0;
124     }
125     case GET_POSSIBLE_CPU_NUM: {
126         cpu_num.val = get_possible_cpu_num();
127         if(copy_to_user((struct ioctl_arg*) arg, &cpu_num,
128             sizeof(struct ioctl_arg))) {
129             pr_err("Fail to copy to user space.");
130         }
131         return 0;
132     }
133     case GET_CPU_STAT_BY_NUM: {
134         struct cpustat* stat = NULL;
135         stat = get_cpustat_by_num(cpu);
136         if(copy_to_user((struct cpustat*) arg, stat,
137             sizeof(struct cpustat))) {
138             pr_err("Fail to copy to user space.");
139         }
140         return 0;
141     }
142     case GET_CPU_STAT_ALL: {
143         struct cpustat* stat = NULL;
144         u64 buffer_size = (get_possible_cpu_num() + 1) *
145             sizeof(struct cpustat);
146         stat = get_cpustat_all();
147         if(copy_to_user((struct cpustat*) arg, stat,
148             buffer_size)) {
149             pr_err("Fail to copy to user space.");
150         }
151         return 0;
152     }
153     case GET_OS_STAT: {
154         struct os_stat os_stat;
155         struct new_utsname *nutsname = utsname();
156         strcpy(os_stat.sysname, nutsname->sysname);
157         strcpy(os_stat.nodename, nutsname->nodename);
158         strcpy(os_stat.release, nutsname->release);
159         strcpy(os_stat.version, nutsname->version);
160         strcpy(os_stat.machine, nutsname->machine);
161         strcpy(os_stat.domainname, nutsname->domainname);
162         if(copy_to_user((struct os_stat*) arg, &os_stat,
163             sizeof(struct os_stat))) {
164             pr_err("Fail to copy to user space.");
165         }
166         return 0;
167     }
168     case SET_CPU: {

```

```

164         if(copy_from_user(&cpu_num, (struct iocctl_arg*) arg,
165             sizeof(struct iocctl_arg))){
166             pr_err("Fail to copy to kernel space.");
167         }
168         cpu = cpu_num.val;
169         return 0;
170     }
171     default:
172         pr_err("Unknown command\n");
173     }
174     return 0;
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 <=
195          * 1024,
196          * overflow after 64.99 years.
197          * exact for HZ=60, 72, 90, 120, 144, 180, 300, 600, 900, ...
198          */
199         return div_u64(x * 9, (9ull * NSEC_PER_SEC + (USER_HZ / 2)) /
200             USER_HZ);
201     #endif
202 }
203
204 #ifdef arch_idle_time
205 static u64 cs_get_idle_time(struct kernel_cpustat *kcs, int cpu)
206 {
207     u64 idle;
208
209     idle = kcs->cpustat[CPUTIME_IDLE];
210     if (cpu_online(cpu) && !nr_iowait_cpu(cpu))
211         idle += arch_idle_time(cpu);
212     return idle;
213 }
214
215 static u64 cs_get_iowait_time(struct kernel_cpustat *kcs, int cpu)

```

```

214 {
215     u64 iowait;
216
217     iowait = kcs->cpustat[CPUTIME_IOWAIT];
218     if (cpu_online(cpu) && nr_iowait_cpu(cpu))
219         iowait += arch_idle_time(cpu);
220     return iowait;
221 }
222
223 #else
224
225 static u64 cs_get_idle_time(struct kernel_cpustat *kcs, int cpu)
226 {
227     u64 idle, idle_usecs = -1ULL;
228
229     if (cpu_online(cpu))
230         idle_usecs = get_cpu_idle_time_us(cpu, NULL);
231
232     if (idle_usecs == -1ULL)
233         /* !NO_HZ or cpu offline so we can rely on cpustat.idle */
234         idle = kcs->cpustat[CPUTIME_IDLE];
235     else
236         idle = idle_usecs * NSEC_PER_USEC;
237
238     return idle;
239 }
240
241 static u64 cs_get_iowait_time(struct kernel_cpustat *kcs, int cpu)
242 {
243     u64 iowait, iowait_usecs = -1ULL;
244
245     if (cpu_online(cpu))
246         iowait_usecs = get_cpu_iowait_time_us(cpu, NULL);
247
248     if (iowait_usecs == -1ULL)
249         /* !NO_HZ or cpu offline so we can rely on cpustat.iowait */
250         iowait = kcs->cpustat[CPUTIME_IOWAIT];
251     else
252         iowait = iowait_usecs * NSEC_PER_USEC;
253
254     return iowait;
255 }
256
257 #endif
258
259 static struct cpustat* get_cpustat_by_num(u64 cpu_num){
260     struct kernel_cpustat *kcs = &kcpustat_cpu(cpu_num);
261     struct cpustat* stat = vmalloc(sizeof(struct cpustat));
262     if (!stat) {
263         pr_err("vmalloc failed.\n");
264         return NULL;
265     }
266     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     {
342         pr_err("Cannot create the structure class.\n");
343         goto rm_major;
344     }
345     if(device_create(dev_class, NULL, major, NULL, DEVICE_NAME) < 0){
346         pr_err("Cannot create the device");
347         goto rm_class;

```

```

348     }
349
350     pr_info("Device created on /dev/%s\n", DEVICE_NAME);
351
352     return 0;
353
354 rm_class:
355     class_destroy(dev_class);
356 rm_major:
357     unregister_chrdev_region(major, 1);
358
359     return -1;
360 }
361
362 static void __exit cpu_stat_exit(void){
363     device_destroy(dev_class, major);
364     class_destroy(dev_class);
365     cdev_del(&cs_dev);
366     unregister_chrdev_region(major, 1);
367     pr_info("cpu_stat: Module unloaded.\n");
368 }
369
370 module_init(cpu_stat_init);
371 module_exit(cpu_stat_exit);

```

3 Код программы пользователя:

```

1 #include "common.h"
2
3 #include <sys/ioctl.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <inttypes.h>
8 #include <stdlib.h>
9 #include <time.h>
10 #include <string.h>
11
12 #define IOC_MAGIC 'a'
13
14 #define GET_OS_STAT _IOR(IOC_MAGIC, 0, struct os_stat)
15 #define SET_CPU _IOW(IOC_MAGIC, 1, struct ioctl_arg)
16 #define GET_CPU_STAT_BY_NUM _IOR(IOC_MAGIC, 2, struct cpustat)
17 #define GET_ONLINE_CPU_NUM _IOR(IOC_MAGIC, 3, struct ioctl_arg)
18 #define GET_POSSIBLE_CPU_NUM _IOR(IOC_MAGIC, 4, struct ioctl_arg)
19 #define GET_CPU_STAT_ALL _IOR(IOC_MAGIC, 5, struct cpustat*)
20
21
22 #define DEVICE_PATH "/dev/os_lab"
23 #define F_OPT_P 0x01
24 #define F_OPT_INTERVAL 0x02
25

```

```

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 *progrname)

```

```

79 {
80     fprintf(stderr, "Usage: %s [ options ]\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               "
196               %"PRIu64"\t\t%"PRIu64"\t%"PRIu64"\t%"PRIu64"\t%"PRIu64"\n",
197               cpus_stat[i].user,
198               cpus_stat[i].nice,
199               cpus_stat[i].system,
200               cpus_stat[i].iowait,
201               cpus_stat[i].irq,
202               cpus_stat[i].softirq,
203               cpus_stat[i].steal,
204               cpus_stat[i].guest,
205               cpus_stat[i].guest_nice,
206               cpus_stat[i].idle
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 -
220             cpu_prev[i].system;
221         cpu_diff[i].idle = cpu_current[i].idle - cpu_prev[i].idle;
222         cpu_diff[i].iowait = cpu_current[i].iowait -
223             cpu_prev[i].iowait;
224         cpu_diff[i].irq = cpu_current[i].irq - cpu_prev[i].irq;
225         cpu_diff[i].softirq = cpu_current[i].softirq -
226             cpu_prev[i].softirq;
227         cpu_diff[i].steal = cpu_current[i].steal - cpu_prev[i].steal;
228         cpu_diff[i].guest = cpu_current[i].guest - cpu_prev[i].guest;
229         cpu_diff[i].guest_nice = cpu_current[i].guest_nice -
230             cpu_prev[i].guest_nice;
231     }
232     return cpu_diff;
233 }
234
235 int loop(int driver, int interval, int count, uint64_t possible_cpu){
236     struct cpustat* cpus_stat_p = malloc((possible_cpu + 1) *
237         sizeof(struct cpustat));
238     if(cpus_stat_p == NULL){
239         printf("CPUS_STAT is NULL");
240         return -1;
241     }
242     struct cpustat* cpus_stat_c = malloc((possible_cpu + 1) *
243         sizeof(struct cpustat));
244     if(cpus_stat_c == NULL){
245         printf("CPUS_STAT is NULL");
246         return -1;
247     }
248     struct os_stat* os_stat = malloc(sizeof(struct os_stat));
249     ioctl(driver, GET_OS_STAT, os_stat);
250     write_header(os_stat, possible_cpu);
251     int i = -1;
252     ioctl(driver, GET_CPU_STAT_ALL, cpus_stat_p);
253     while(1){
254         if(++i == count) break;
255         sleep(interval);
256         ioctl(driver, GET_CPU_STAT_ALL, cpus_stat_c);
257         struct cpustat* diff_stat = cpu_diff(cpus_stat_p,
258             cpus_stat_c, possible_cpu);
259         unsigned long long* tot_jiffies =
260             get_global_cpu_mpstats(diff_stat, possible_cpu);
261         struct cpu_percent_stat* cpus_percent_stat =
262             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 ,
299                 KEY_all) < 0){
300                 printf("Fail to parse cpu mask");
301                 return -1;
302             }
303             opt_flags |= F_OPT_P;
304         }
305         else if(interval < 0){

```

```

306         if (strspn(argv[opt], "0123456789") !=
307             strlen(argv[opt])) {
308             usage(argv[0]);
309         }
310         interval = atoi(argv[opt]);
311         if (interval < 0) {
312             usage(argv[0]);
313         }
314         opt_flags |= F_OPT_INTERVAL;
315     }
316     else if(count < 0){
317         if (strspn(argv[opt], "0123456789") != strlen(argv[opt])
318             || !interval) {
319             usage(argv[0]);
320         }
321         count = atoi(argv[opt]);
322         if (count < 1) {
323             usage(argv[0]);
324         }
325     }
326     if(!(opt_flags & F_OPT_P)){
327         cpu_mask[0] = 1;
328     }
329     if(interval > 0 && (count > 0 || count == -1)){
330         loop(driver, interval, count, possible_cpu);
331     }
332     else{
333         struct cpustat* cpus_stat = malloc((possible_cpu + 1) *
334             sizeof(struct cpustat));
335         if(cpus_stat == NULL){
336             printf("CPUS_STAT is NULL");
337             return -1;
338         }
339         struct os_stat* os_stat = malloc(sizeof(struct os_stat));
340         ioctl(driver, GET_OS_STAT, os_stat);
341         write_header(os_stat, possible_cpu);
342         ioctl(driver, GET_CPU_STAT_ALL, cpus_stat);
343         unsigned long long* tot_jiffies =
344             get_global_cpu_mpstats(cpus_stat, possible_cpu);
345         struct cpu_percent_stat* cpus_percent_stat =
346             get_percentage_stat(cpus_stat, possible_cpu, tot_jiffies);
347         write_cpu_percent_stat(cpus_percent_stat, possible_cpu);
348         free(tot_jiffies);
349         free(cpus_percent_stat);
350         free(cpus_stat);
351         free(os_stat);
352     }
353     close(driver);
354     return 0;
355 }

```

```

1 #include "common.h"

```

```

2 #include <time.h>
3 #include <stdio.h>
4 #include <inttypes.h>
5 #include <string.h>
6 #include <stdint.h>
7 #include <stdlib.h>
8
9 #define KEY_ALL "ALL"
10
11 void get_current_time(char* time_str){
12     time_t rawtime;
13     struct tm * timeinfo;
14
15     time(&rawtime);
16     timeinfo = localtime(&rawtime);
17
18     sprintf(time_str, "%02d:%02d:%02d", timeinfo->tm_hour,
19         timeinfo->tm_min, timeinfo->tm_sec);
20 }
21
22 void get_current_date(char* date_str){
23     time_t rawtime;
24     struct tm * timeinfo;
25
26     time(&rawtime);
27     timeinfo = localtime(&rawtime);
28
29     sprintf(date_str, "%02d.%02d.%04d", timeinfo->tm_mday,
30         timeinfo->tm_mon + 1, timeinfo->tm_year + 1900);
31 }
32
33 int parse_values_range(char* t, int max_value, int* val_low, int*
34     val ){
35     char *s, *valstr, range[16];
36
37     strncpy(range, t, 16);
38     range[15] = '\0';
39     s = strchr(range, '-');
40     valstr = t;
41     if (s) {
42         *s = '\0';
43         valstr = s + 1;
44         *val_low = atoi(range);
45         *val = atoi(valstr);
46         if (*val_low < 0 || *val_low >= max_value || *val < 0 ||
47             *val >= max_value) {
48             return -1;
49         }
50     } else {
51         *val_low = *val = atoi(range);
52         if (*val_low < 0 || *val_low >= max_value) {
53             return -1;
54         }
55     }
56 }

```

```

51     }
52 }
53
54 int parse_values(char* strargv, uint8_t* cpu_mask, int max_value,
55     const char * KEY_WORD ){
56     int i, val_low, val;
57     char *t;
58
59     if (!strcmp(strargv, KEY_ALL)) {
60         memset(cpu_mask, ~0, max_value);
61         return 0;
62     }
63     for (t = strtok(strargv, ","); t; t = strtok(NULL, ",")){
64         if(!strcmp(t, KEY_WORD)){
65             cpu_mask[0] = 1;
66         }
67         else {
68             if (parse_values_range(t, max_value, &val_low, &val) <
69                 0) {
70                 return -1;
71             }
72             for(i = val_low; i <= val; i++){
73                 cpu_mask[i + 1] = 1;
74             }
75         }
76     }
77     return 0;
78 }

```

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)

```

	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.