EXTRA

single

```
1  static struct proc_dir_entry *fortune_file = NULL;
2  static struct proc_dir_entry *fortune_dir = NULL;
3  static struct proc_dir_entry *fortune_link = NULL;
4  static char *cookie_buffer = NULL;
5  static int write_index = 0;
6  static int read_index = 0;
7  static char tmp[BUF_SIZE];
8  ssize_t write(struct file *file, const char __user *buf, size_t len,
       loff_t *offp)
9  {
10   printk(KERN_INFO "␣seqfile:␣called␣write");
11   if(len > BUF_SIZE - write_index + 1)
12       // error
13   if(copy_from_user(&cookie_buffer[write_index], buf, len))
14       // error
15   write_index += len;
16   cookie_buffer[write_index-1] = '\0';
17   return len;
18 }
19 static int seqfile_show(struct seq_file *m, void *v)
20 {
21   printk(KERN_INFO "␣seqfile:␣called␣show");
22   int len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
23   seq_printf(m, "%s", tmp);
24   read_index += len;
25   return 0;
26 }
27 ssize_t seqfile_read(struct file *file, char __user *buf, size_t count,
       loff_t *offp)
28 {
29   printk(KERN_INFO "␣seqfile:␣called␣read");
30   return seq_read(file, buf, count, offp);
31 }
32 int seqfile_open(struct inode *inode, struct file *file)
```

```
33 {
34   printk(KERN_INFO "+ seqfile: called open");
35   return single_open(file, seqfile_show, NULL);
36 }
37 int seqfile_release(struct inode *inode, struct file *file)
38 {
39   printk(KERN_INFO "+ seqfile: called release");
40   return single_release(inode, file);
41 }
42 static struct proc_ops fops = {
43   .proc_open = seqfile_open,
44   .proc_read = seqfile_read,
45   .proc_write = write,
46   .proc_release = seqfile_release,
47 };
48 void freemem(void)
49 {
50   if (cookie_buffer)
51     vfree(cookie_buffer);
52   if (fortune_link)
53     remove_proc_entry(SYMLINK, NULL);
54   if (fortune_file)
55     remove_proc_entry(FILENAME, fortune_dir);
56   if (fortune_dir)
57     remove_proc_entry(DIRNAME, NULL);
58 }
59 static int __init init_seqfile_module(void)
60 {
61   if (!(cookie_buffer = vmalloc(BUF_SIZE)))
62     // error
63   memset(cookie_buffer, 0, BUF_SIZE);
64   if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
65     // error
66   else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops
       )))
67     // error
68   else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
69     //error
```

```
70    write_index = 0;
71    read_index = 0;
72    printk(KERN_INFO "+ module loaded");
73    return 0;
74  }
75  static void __exit exit_seqfile_module(void)
76  {
77    freemem();
78    printk(KERN_INFO "+ seqfile: unloaded");
79  }
80  module_init(init_seqfile_module);
81  module_exit(exit_seqfile_module);
```

```
1   static struct proc_dir_entry *fortune_file = NULL;
2   static struct proc_dir_entry *fortune_dir = NULL;
3   static struct proc_dir_entry *fortune_link = NULL;
4   static char *cookie_buffer = NULL;
5   static int write_index = 0;
6   static int read_index = 0;
7   static char tmp[BUF_SIZE];
8   ssize_t seq_file_write(struct file *filep, const char __user *buf, size_t
        len, loff_t *offp)
9   {
10    printk(KERN_INFO "+ seq_file: write");
11    if(len > BUF_SIZE - write_index + 1)
12        // error
13    if(copy_from_user(&cookie_buffer[write_index], buf, len))
14        // error
15    write_index += len;
16    cookie_buffer[write_index-1] = '\0';
17    return len;
18  }
19  static int seq_file_show(struct seq_file *m, void *v)
20  {
21    printk(KERN_INFO "+ seq_file: show");
22    if (!write_index)
23      return 0;
24    if (read_index >= write_index)
25    {
```

```
26      read_index = 0;
27    }
28    int len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
29    seq_printf(m, "%s", tmp);
30    read_index += len;
31    return 0;
32  }
33  static void *seq_file_start(struct seq_file *m, loff_t *pos)
34  {
35    printk(KERN_INFO "+ seq_file: start");
36    static unsigned long counter = 0;
37    if (!*pos) {
38      return &counter;
39    }
40    else {
41      *pos = 0;
42      return NULL;
43    }
44  }
45  static void *seq_file_next(struct seq_file *m, void *v, loff_t *pos)
46  {
47    printk(KERN_INFO "+ seq_file: next");
48    unsigned long *tmp = (unsigned long *) v;
49    (*tmp)++;
50    (*pos)++;
51    return NULL;
52  }
53  static void seq_file_stop(struct seq_file *m, void *v)
54  {
55    printk(KERN_INFO "+ seq_file: stop");
56  }
57  static struct seq_operations seq_file_ops = {
58    .start = seq_file_start,
59    .next = seq_file_next,
60    .stop = seq_file_stop,
61    .show = seq_file_show
62  };
63  static int seq_file_open(struct inode *i, struct file * f)
```

```
64 {
65   printk(KERN_DEBUG "+ seq_file: open seq_file");
66   return seq_open(f, &seq_file_ops);
67 }
68 static struct proc_ops fops = {
69   .proc_open = seq_file_open,
70   .proc_read = seq_read,
71   .proc_write = seq_file_write,
72   .proc_lseek = seq_lseek,
73   .proc_release = seq_release,
74 };
75 void freemem(void)
76 {
77   if (cookie_buffer)
78     vfree(cookie_buffer);
79   if (fortune_link)
80     remove_proc_entry(SYMLINK, NULL);
81   if (fortune_file)
82     remove_proc_entry(FILENAME, fortune_dir);
83   if (fortune_dir)
84     remove_proc_entry(DIRNAME, NULL);
85 }
86 static int __init init_seq_file_module(void)
87 {
88   if (!(cookie_buffer = vmalloc(BUF_SIZE)))
89       // error
90   memset(cookie_buffer, 0, BUF_SIZE);
91   if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
92       // error
93   else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops
       )))
94       // error
95   else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
96       // error
97   write_index = 0;
98   read_index = 0;
99   printk(KERN_INFO "+ module loaded");
100  return 0;
```

```
}
static void __exit exit_seq_file_module(void)
{
  freemem();
  printk(KERN_INFO "+ seq_file: unloaded");
}
module_init(init_seq_file_module);
module_exit(exit_seq_file_module);
```

```
int main(int argc, char ** argv)
{
    int fd[2];
    char buf[BUF_SIZE];
    pid_t child_pid[CHILD_NUM];
    if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
        // error
    for (size_t i = 0; i < CHILD_NUM; i++)
    {
        if ((child_pid[i] = fork()) == -1)

        if (child_pid[i] == 0)
        {
            sprintf(buf, "%d", getpid());
            write(fd[0], buf, sizeof(buf));
            printf("Child wrote %s\n", buf);
            //sleep(1);
            read(fd[0], buf, sizeof(buf));
            printf("Child %d read %s\n", getpid(), buf);
            return EXIT_SUCCESS;
        }
        else
        {
            read(fd[1], buf, sizeof(buf));
            printf("Parent read %s\n", buf);
            sprintf(buf, "%s %d", buf, getpid());
            write(fd[1], buf, sizeof(buf));
            printf("Parent wrote %s\n", buf);
        }
    }
```

```
31      close(fd[0]);
32      close(fd[1]);
33      return EXIT_SUCCESS;
34  }
```

0.1. AF_UNIX + SOCK_DGRAM + без bind у клиента

client

```
1   int main(int argc, char ** argv)
2   {
3       char buf[BUF_SIZE];
4       if (argc != 2)
5           // error
6       sprintf(buf, "%d %s", getpid(), argv[1]);
7       int sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
8       if (sockfd == -1)
9           // error
10      struct sockaddr sa;
11      sa.sa_family = AF_UNIX;
12      strcpy(sa.sa_data, "socket.soc");
13      if (sendto(sockfd, buf, sizeof(buf), 0, &sa, sizeof(sa)) == -1)
14          // error
15      close(sockfd);
16      return EXIT_SUCCESS;
17  }
```

server

```
1   int sockfd;
2   void signal_handler(int signal)
3   {
4       printf("\nCaught signal = %d\n", signal);
5       unlink("socket.soc");
6       close(sockfd);
7       printf("\nServer exiting.\n");
```

```
8       exit(0);
9   }
10  int main()
11  {
12    if ((signal(SIGINT, signal_handler) == SIG_ERR))
13        // error
14    char buf[BUF_SIZE];
15    sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
16    if (sockfd == -1)
17        // error
18    struct sockaddr sa;
19    sa.sa_family = AF_UNIX;
20    strcpy(sa.sa_data, "socket.soc");
21    if (bind(sockfd, &sa, sizeof(sa)) < 0)
22        // error
23    int bytes;
24    while (1)
25    {
26      bytes = recvfrom(sockfd, buf, sizeof(buf), 0, NULL, NULL);
27      if (bytes == -1)
28          // error
29      printf("\nreceived: %s\n", buf);
30    }
31    return EXIT_SUCCESS;
32  }
```

0.2. AF_UNIX + SOCK_DGRAM + bind у клиента

client

```
1   int main(int argc, char **argv)
2   {
3       char buf[BUF_SIZE];
4       if (argc != 2)
5           // error
6       sprintf(buf, "%d %s", getpid(), argv[1]);
7       int sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
```

```
8       if (sockfd == -1)
9           // error
10      struct sockaddr sa;
11      sa.sa_family = AF_UNIX;
12      strcpy(sa.sa_data, "socket.soc");
13      socklen_t len = sizeof(sa);
14      char name[20];
15      sprintf(name, "%d.soc", getpid());
16      struct sockaddr ca;
17      ca.sa_family = AF_UNIX;
18      strcpy(ca.sa_data, name);
19      if (bind(sockfd, &ca, sizeof(ca)) == -1)
20          // error
21      if (sendto(sockfd, buf, sizeof(buf), 0, &sa, len) == -1)
22          // error
23      if (recvfrom(sockfd, buf, sizeof(buf), 0, NULL, NULL) == -1) //&sa, &
            len
24          // error
25      printf("\nreceived: %s\n", buf);
26      unlink(name);
27      close(sockfd);
28      return EXIT_SUCCESS;
29  }
```

server

```
1   int sockfd;
2   void signal_handler(int signal)
3   {
4       printf("\nCaught signal = %d\n", signal);
5       unlink("socket.soc");
6       close(sockfd);
7       printf("\nServer exiting.\n");
8       exit(0);
9   }
10  int main(int argc, char **argv)
11  {
```

```
12      if ((signal(SIGINT, signal_handler) == SIG_ERR))
13          // error
14      char buf[BUF_SIZE];
15      sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
16      if (sockfd == -1)
17          // error
18      struct sockaddr sa;
19      sa.sa_family = AF_UNIX;
20      strcpy(sa.sa_data, "socket.soc");
21      if (bind(sockfd, &sa, sizeof(sa)) == -1)
22          // error
23      int bytes;
24      while (1)
25      {
26          struct sockaddr ca;
27          socklen_t len = sizeof(ca);
28          bytes = recvfrom(sockfd, buf, sizeof(buf), 0, &ca, &len);
29          if (bytes == -1)
30              // error
31          printf("\nreceived: %s\n", buf);
32          sprintf(buf, "%s %d", buf, getpid());
33          if (sendto(sockfd, buf, sizeof(buf), 0, &ca, len) == -1)
34              // error
35          printf("sent: %s\n", buf);
36      }
37      return EXIT_SUCCESS;
38  }
```

```
1   MODULE_LICENSE("GPL");
2   struct tasklet_struct *tasklet;
3   void tasklet_func(unsigned long data)
4   {
5       printk(KERN_INFO "+: ----------");
6       printk(KERN_INFO "+: tasklet began");
7       printk(KERN_INFO "+: tasklet count = %u", tasklet->count.counter);
8       printk(KERN_INFO "+: tasklet state = %lu", tasklet->state);
9       printk(KERN_INFO "+: key code - %d", data);
10      if (data < ASCII_LEN)
11          printk(KERN_INFO "+: key press - %s", ascii[data]);
```

```
12      if (data > 128 && data < 128 + ASCII_LEN)
13          printk(KERN_INFO "+: key release - %s", ascii[data - 128]);
14      printk(KERN_INFO "+: tasklet ended");
15      printk(KERN_INFO "+: ------------------------");
16  }
17  static irqreturn_t my_irq_handler(int irq, void *dev_id)
18  {
19    int code;
20    printk(KERN_INFO "+: my_irq_handler called\n");
21    if (irq != IRQ_NUMBER)
22    {
23      printk(KERN_INFO "+: irq not handled");
24      return IRQ_NONE;
25    }
26    printk(KERN_INFO "+: tasklet state (before schedule) = %lu",
27                     tasklet->state);
28    code = inb(0x60);
29    tasklet->data = code;
30    tasklet_schedule(tasklet);
31    printk(KERN_INFO "+: tasklet scheduled");
32    printk(KERN_INFO "+: tasklet state (after schedule) = %lu",
33                      tasklet->state);
34
35    return IRQ_HANDLED;
36  }
37  static int __init my_init(void)
38  {
39    if (request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED, "
          tasklet_irq_handler", (void *) my_irq_handler))
40        // error
41    tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
42    if (tasklet == NULL)
43        // error
44    tasklet_init(tasklet, tasklet_func, 0);
45    printk(KERN_INFO "+: module loaded\n");
46    return 0;
47  }
48  static void __exit my_exit(void)
```

```
49 {
50    tasklet_kill(tasklet);
51    free_irq(IRQ_NUMBER, my_irq_handler);
52    printk(KERN_INFO "+: module unloaded\n");
53 }
54 module_init(my_init);
55 module_exit(my_exit);
```

```
1  MODULE_LICENSE("GPL");
2  typedef struct
3  {
4      struct work_struct work;
5      int code;
6  } my_work_struct_t;
7  static struct workqueue_struct *my_wq;
8  static my_work_struct_t *work1;
9  static struct work_struct *work2;
10 void work1_func(struct work_struct *work)
11 {
12     my_work_struct_t *my_work = (my_work_struct_t *)work;
13     int code = my_work->code;
14     printk(KERN_INFO "+: ------------------------");
15     printk(KERN_INFO "+: work1 began");
16     printk(KERN_INFO "+: key code - %d", code);
17     if (code < ASCII_LEN)
18         printk(KERN_INFO "+: key press - %s", ascii[code]);
19     if (code > 128 && code < 128 + ASCII_LEN)
20         printk(KERN_INFO "+: key release - %s", ascii[code - 128]);
21     printk(KERN_INFO "+: work1 ended");
22     printk(KERN_INFO "+: ------------------------");
23 }
24 void work2_func(struct work_struct *work)
25 {
26     printk(KERN_INFO "+: ------------------------");
27     printk(KERN_INFO "+: work2 began");
28     msleep(10);
29     printk(KERN_INFO "+: work2 ended");
30     printk(KERN_INFO "+: ------------------------");
31 }
```

```
32 irqreturn_t my_irq_handler(int irq, void *dev)
33 {
34     int code;
35     printk(KERN_INFO "+: my_irq_handler called\n");
36     if (irq != IRQ_NUMBER)
37     {
38         printk(KERN_INFO "+: irq not handled");
39         return IRQ_NONE;
40     }
41     code = inb(0x60);
42     work1->code = code;
43     queue_work(my_wq, (struct work_struct *)work1);
44     queue_work(my_wq, work2);
45     return IRQ_HANDLED;
46 }
47 static int __init my_init(void)
48 {
49     int rc = request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED,
50                          "work_irq_handler", (void *) my_irq_handler);
51     if (rc)
52         // error
53     my_wq = alloc_workqueue("%s", __WQ_LEGACY | WQ_MEM_RECLAIM, 1, "my_wq"
        );
54     if (my_wq == NULL)
55         // error
56     work1 = kmalloc(sizeof(my_work_struct_t), GFP_KERNEL);
57     if (work1 == NULL)
58         // error
59     work2 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);
60     if (work2 == NULL)
61         // error
62     INIT_WORK((struct work_struct *)work1, work1_func);
63     INIT_WORK(work2, work2_func);
64     printk(KERN_INFO "+: module loaded");
65     return 0;
66 }
67 static void __exit my_exit(void)
68 {
```

```
69      synchronize_irq(IRQ_NUMBER);
70      free_irq(IRQ_NUMBER, my_irq_handler);
71      flush_workqueue(my_wq);
72      destroy_workqueue(my_wq);
73      kfree(work1);
74      kfree(work2);
75      printk(KERN_INFO "+: module unloaded");
76  }
77  module_init(my_init);
78  module_exit(my_exit);
```

Код клиента

```
1   int main(void)
2   {
3     setbuf(stdout, NULL);
4     struct sockaddr_in serv_addr =
5     {
6       .sin_family = AF_INET,
7       .sin_addr.s_addr = INADDR_ANY,
8       .sin_port = htons(SERVER_PORT)
9     };
10    socklen_t serv_len;
11    char buf[MSG_LEN];
12    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
13    // error handling
14    if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <
          0)
15      // error handling
16    char input_msg[MSG_LEN], output_msg[MSG_LEN];
17    sprintf(output_msg, "%d", getpid());
18    if (write(sock_fd, output_msg, strlen(output_msg) + 1) == -1)
19      // error handling
20    if (read(sock_fd, input_msg, MSG_LEN) == -1)
21      // error handling
22    printf("Client receive: %s \n", input_msg);
23    close(sock_fd);
24    return EXIT_SUCCESS;
25  }
```

Код сервера

```
1  int main()
2  {
3    int epoll_fd;
4    if ((epoll_fd = epoll_create(CLIENTS_MAX)) == -1)
5        // error
6    int sock;
7    if ((sock = socket(AF_INET, SOCK_STREAM | O_NONBLOCK, 0)) == -1)
8        // error
9    int sopt = 1;
10   if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sopt, sizeof(sopt)) ==
       -1)
11       // error
12   struct sockaddr_in addr;
13   addr.sin_family = AF_INET;
14   addr.sin_addr.s_addr = INADDR_ANY;
15   addr.sin_port = htons(PORT);
16   if (bind(sock, (struct sockaddr *) &addr, sizeof(addr)) == -1)
17       // error
18   if (listen(sock, CLIENTS_MAX) == -1)
19       // error
20   struct epoll_event epev;
21   epev.events = EPOLLIN;
22   epev.data.fd = sock;
23   if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sock, &epev) == -1)
24       // error
25   while (1)
26   {
27     struct epoll_event epev[CLIENTS_MAX + 1];
28     int num;
29     if ((num = epoll_wait(epoll_fd, &epev, CLIENTS_MAX + 1, -1)) == -1)
30         // error
31     for (int i = 0; i < num; i++)
32     {
33       if (epev[i].data.fd == sock)
34       {
35         int conn;
36         if ((conn = accept(sock, NULL, NULL)) == -1)
```

```
37              // error
38          struct epoll_event epev;
39          int flags = fcntl(conn, F_GETFL, 0);
40          fcntl(conn, F_SETFL, flags | O_NONBLOCK);
41          epev.events = EPOLLIN | EPOLLET ;
42          epev.data.fd = conn;
43          if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn, &epev) == -1)
44              // error
45        }
46        else
47        {
48          int conn = epev[i].data.fd;
49          char received_msg[BUF_SIZE], send_msg[BUF_SIZE];
50          if (recv(conn, received_msg, sizeof(received_msg), 0) == -1)
51              // error
52          printf("Server %d received message: %s\n", getpid(), received_msg)
                ;
53          sprintf(send_msg, "%s from server with pid %d", received_msg,
                getpid());
54          if (send(conn, send_msg, sizeof(send_msg), 0) == -1)
55              // error
56          printf("Server %d send message: %s\n", getpid(), send_msg);
57          close(conn);
58        }
59      }
60    }
61    return 0;
62  }
```

fortune

```
1  MODULE_LICENSE("GPL");
2  #define BUF_SIZE PAGE_SIZE
3  #define DIRNAME "fortunes"
4  #define FILENAME "fortune"
5  #define SYMLINK "fortune_link"
6  #define FILEPATH DIRNAME "/" FILENAME
7  static struct proc_dir_entry *fortune_dir = NULL;
```

```
8  static struct proc_dir_entry *fortune_file = NULL;
9  static struct proc_dir_entry *fortune_link = NULL;
10 static char *cookie_buffer;
11 static int write_index;
12 static int read_index;
13 static char tmp[BUF_SIZE];
14 ssize_t fortune_read(struct file *filp, char __user *buf, size_t count,
       loff_t *offp)
15 {
16   int len;
17   printk(KERN_INFO "␣fortune:␣read␣called");
18   if (*offp > 0 || !write_index)
19   {
20     printk(KERN_INFO "␣fortune:␣empty");
21     return 0;
22   }
23   if (read_index >= write_index)
24     read_index = 0;
25   len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
26   if (copy_to_user(buf, tmp, len))
27       // error
28   read_index += len;
29   *offp += len;
30   return len;
31 }
32 ssize_t fortune_write(struct file *filp, const char __user *buf, size_t
       len, loff_t *offp)
33 {
34   printk(KERN_INFO "␣fortune:␣write␣called");
35   if (len > BUF_SIZE - write_index + 1)
36       // error
37   if (copy_from_user(&cookie_buffer[write_index], buf, len))
38       // error
39   write_index += len;
40   cookie_buffer[write_index - 1] = '\0';
41   return len;
42 }
43 int fortune_open(struct inode *inode, struct file *file)
```

```
44 {
45   printk(KERN_INFO "+ fortune: called open");
46   return 0;
47 }
48 int fortune_release(struct inode *inode, struct file *file)
49 {
50   printk(KERN_INFO "+ fortune: called release");
51   return 0;
52 }
53 static const struct proc_ops fops = {
54   proc_read: fortune_read,
55   proc_write: fortune_write,
56   proc_open: fortune_open,
57   proc_release: fortune_release
58 };
59 static void freemem(void)
60 {
61   if (fortune_link)
62     remove_proc_entry(SYMLINK, NULL);
63   if (fortune_file)
64     remove_proc_entry(FILENAME, fortune_dir);
65   if (fortune_dir)
66     remove_proc_entry(DIRNAME, NULL);
67   if (cookie_buffer)
68     vfree(cookie_buffer);
69 }
70 static int __init fortune_init(void)
71 {
72   if (!(cookie_buffer = vmalloc(BUF_SIZE)))
73       // error
74   memset(cookie_buffer, 0, BUF_SIZE);
75   if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
76       // error
77   else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops
       )))
78       // error
79   else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
80       // error
```

```
81    write_index = 0;
82    read_index = 0;
83    printk(KERN_INFO "+ fortune: module loaded");
84    return 0;
85  }
86  static void __exit fortune_exit(void)
87  {
88    freemem();
89    printk(KERN_INFO "+ fortune: module unloaded");
90  }
91  module_init(fortune_init)
92  module_exit(fortune_exit)
```

Вопрос 6

```
1   #define CACHE_SIZE 1024
2   #define CACHE_NAME "kittyfs_cache"
3   static struct kmem_cache *cache = NULL;
4   static struct kittyfs_inode **inode_cache = NULL;
5   static size_t cache_index = 0;
6   static struct kittyfs_inode
7   {
8       int i_mode;
9       unsigned long i_ino;
10  } kittyfs_inode;
11  static void kittyfs_kill_sb(struct super_block *sb){
12      printk(KERN_INFO "+ kittyfs: kill super block");
13      kill_anon_super(sb);
14  }
15  static void kittyfs_put_sb(struct super_block *sb)
16  {
17      printk(KERN_INFO "+ kittyfs: superblock destroy called");
18  }
19  static struct super_operations const kittyfs_sb_ops = {
20      .put_super = kittyfs_put_sb,
21      .statfs = simple_statfs,
22      .drop_inode = generic_delete_inode,
23  };
24  static struct inode *kittyfs_new_inode(struct super_block *sb, int ino,
      int mode)
```

```
25 {
26   struct inode *res;
27   res = new_inode(sb);
28   if (!res)
29     return NULL;
30   res->i_ino = ino;
31     res->i_mode = mode;
32     res->i_atime = res->i_mtime = res->i_ctime = current_time(res);
33     res->i_op = &simple_dir_inode_operations;
34     res->i_fop = &simple_dir_operations;
35     res->i_private = &kittyfs_inode;
36     if (cache_index >= CACHE_SIZE)
37       return NULL;
38     inode_cache[cache_index] = kmem_cache_alloc(cache, GFP_KERNEL);
39     if (inode_cache[cache_index])
40     {
41       inode_cache[cache_index]->i_ino = res->i_ino;
42       inode_cache[cache_index]->i_mode = res->i_mode;
43       cache_index++;
44     }
45     return res;
46 }
47 static int kittyfs_fill_sb(struct super_block *sb, void *data, int silent)
48 {
49     struct dentry *root_dentry;
50     struct inode *root_inode;
51     sb->s_blocksize = PAGE_SIZE;
52     sb->s_blocksize_bits = PAGE_SHIFT;
53     sb->s_magic = MAGIC_NUM;
54     sb->s_op = &kittyfs_sb_ops;
55     root_inode = kittyfs_new_inode(sb, 1, S_IFDIR | 0755);
56     if (!root_inode)
57         // error
58     root_dentry = d_make_root(root_inode);
59     if (!root_dentry)
60         // error
61     sb->s_root = root_dentry;
62     return 0;
```

```
63 }
64 static struct dentry *kittyfs_mount(struct file_system_type *type, int
       flags, const char *dev, void *data)
65 {
66     struct dentry *const root_dentry = mount_nodev(type, flags, data,
          kittyfs_fill_sb);
67     if (IS_ERR(root_dentry))
68         printk(KERN_ERR "␣kittyfs:␣cannot␣mount");
69     else
70         printk(KERN_INFO "␣kittyfs:␣mount␣successful");
71     return root_dentry;
72 }
73 static void kittyfs_slab_constructor(void *addr)
74 {
75     memset(addr, 0, sizeof(struct kittyfs_inode));
76 }
77 static struct file_system_type kittyfs_type = {
78     .owner = THIS_MODULE,
79     .name = "kittyfs",
80     .mount = kittyfs_mount,
81     .kill_sb = kittyfs_kill_sb,
82 };
83 static int __init kittyfs_init(void)
84 {
85     int err = register_filesystem(&kittyfs_type);
86     if (err != 0)
87         // error
88     if ((inode_cache = kmalloc(sizeof(struct kittyfs_inode*)*CACHE_SIZE,
          GFP_KERNEL)) == NULL)
89         // error
90
91     if ((cache = kmem_cache_create(CACHE_NAME, sizeof(struct kittyfs_inode
          ), 0, SLAB_HWCACHE_ALIGN, kittyfs_slab_constructor)) == NULL)
92         // error
93     printk(KERN_INFO "␣kittyfs:␣module␣loaded");
94     return 0;
95 }
96 static void __exit kittyfs_exit(void)
```

```
97  {
98      int err;
99      int i;
100     for (i = 0; i < cache_index; i++)
101       kmem_cache_free(cache, inode_cache[i]);
102     kmem_cache_destroy(cache);
103     kfree(inode_cache);
104     err = unregister_filesystem(&kittyfs_type);
105     if (err != 0)
106         printk(KERN_ERR "+ kittyfs: cannot unregister filesystem");
107     else
108         printk(KERN_INFO "+ kittyfs: module is unloaded");
109 }
110 module_init(kittyfs_init);
111 module_exit(kittyfs_exit);
```

```
1   MODULE_LICENSE("GPL");
2   static int __init mod_init(void)
3   {
4       printk(KERN_INFO " + module is loaded.\n");
5       struct task_struct *task = &init_task;
6       do
7       {
8           printk(KERN_INFO " + %s (%d) (%d - state , %d - prio , flags - %d,
                policy - %d), parent %s (%d), d_name %s",
9               task->comm, task->pid, task->__state, task->prio, task->flags,
                    task->policy, task->parent->comm, task->parent->pid, task
                   ->fs->root.dentry->d_name.name);
10      } while ((task = next_task(task)) != &init_task);
11      printk(KERN_INFO " + %s (%d) (%d - state , %d - prio , flags - %d,
            policy - %d), parent %s (%d), d_name %s",
12          current->comm, current->pid, current->__state, current->prio,
                current->flags, current->policy, current->parent->comm, current
               ->parent->pid, current->fs->root.dentry->d_name.name);
13      return 0;
14  }
15  static void __exit mod_exit(void)
16  {
17      printk(KERN_INFO " + %s - %d, parent %s - %d\n", current->comm,
```

```
18              current->pid, current->parent->comm, current->parent->pid);
19      printk(KERN_INFO "+ module is unloaded.\n");
20  }
21  module_init(mod_init);
22  module_exit(mod_exit);
```

kernel_module.h

```
1  extern char *module_1_data;
2  extern char *module_1_proc(void);
3  extern char *module_1_noexport(void);
```

module_1.c

```
1  #include "kernel_module.h"
2  MODULE_LICENSE("GPL");
3  char *module_1_data = "ABCDE";
4  extern char *module_1_proc(void) { return module_1_data; }
5  static char *module_1_local(void) { return module_1_data; }
6  extern char *module_1_noexport(void) { return module_1_data; }
7  EXPORT_SYMBOL(module_1_data);
8  EXPORT_SYMBOL(module_1_proc);
9  static int __init module_1_init(void)
10 {
11      printk("+ module_1 started.\n");
12      printk("+ module_1 use local from module_1: %s\n", module_1_local());
13      printk("+ module_1 use noexport from module_1: %s\n",
            module_1_noexport());
14      return 0;
15 }
16 static void __exit module_1_exit(void) { printk("+ module module_1
        unloaded.\n"); }
17 module_init(module_1_init);
18 module_exit(module_1_exit);
```

module_2.c

```
1  #include "kernel_module.h"
2  MODULE_LICENSE("GPL");
3  static int __init module_2_init(void)
4  {
5      printk("+ module module_2 started.\n");
```

```
6       printk("+ data string exported from module_1: %s\n", module_1_data);
7       printk("+ string returned module_1_proc(): %s\n", module_1_proc());
8       //printk( "+ module_2 use local from module_1: %s\n", module_1_local()
            );
9       //printk( "+ module_2 use noexport from module_1: %s\n",
            module_1_noexport());
10      return 0;
11  }
12  static void __exit module_2_exit(void)
13  {
14      printk("+ module_2 unloaded.\n");
15  }
16  module_init(module_2_init);
17  module_exit(module_2_exit);
```

module_3.c

```
1   #include "kernel_module.h"
2   MODULE_LICENSE("GPL");
3   static int __init module_2_init(void)
4   {
5       printk("+ module_3 started.\n");
6       printk("+ data string exported from module_1: %s\n", module_1_data);
7       printk("+ string returned module_1_proc() is: %s\n", module_1_proc());
8       return -1;
9   }
10  module_init(module_2_init);
```