EXTRA

single

```
static struct proc_dir_entry *fortune_file = NULL;
static struct proc_dir_entry *fortune_dir = NULL;
static struct proc_dir_entry *fortune_link = NULL;
static char *cookie_buffer = NULL;
static int write_index = 0;
static int read_index = 0;
static char tmp[BUF_SIZE];
ssize_t write(struct file *file, const char __user *buf, size_t len,
    loff_t *offp)
{
  printk(KERN_INFO "+ seqfile: called write");
  if(len > BUF_SIZE - write_index + 1)
      // error
  if(copy_from_user(&cookie_buffer[write_index], buf, len))
      // error
  write_index += len;
  cookie_buffer[write_index-1] = '\0';
  return len;
}
static int seqfile_show(struct seq_file *m, void *v)
{
  printk(KERN_INFO "+ seqfile: called show");
  int len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
  seq_printf(m, "%s", tmp);
  read_index += len;
  return 0;
}
ssize_t seqfile_read(struct file *file, char __user *buf, size_t count,
    loff_t *offp)
{
  printk(KERN_INFO "+ seqfile: called read");
  return seq_read(file, buf, count, offp);
}
int seqfile_open(struct inode *inode, struct file *file)
```

```
{
  printk(KERN_INFO "+␣seqfile:␣called␣open");
  return single_open(file, seqfile_show, NULL);
}
int seqfile_release(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "+␣seqfile:␣called␣release");
  return single_release(inode, file);
}
static struct proc_ops fops = {
  .proc_open = seqfile_open,
  .proc_read = seqfile_read,
  .proc_write = write,
  .proc_release = seqfile_release,
};
void freemem(void)
{
  if (cookie_buffer)
    vfree(cookie_buffer);
  if (fortune_link)
    remove_proc_entry(SYMLINK, NULL);
  if (fortune_file)
    remove_proc_entry(FILENAME, fortune_dir);
  if (fortune_dir)
    remove_proc_entry(DIRNAME, NULL);
}
static int __init init_seqfile_module(void)
{
  if (!(cookie_buffer = vmalloc(BUF_SIZE)))
    // error
  memset(cookie_buffer, 0, BUF_SIZE);
  if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
    // error
  else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops
    )))
      // error
  else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
      //error
  write_index = 0;
  read_index = 0;
  printk(KERN_INFO "+␣module␣loaded");
  return 0;
}
static void __exit exit_seqfile_module(void)
{
  freemem();
  printk(KERN_INFO "+␣seqfile:␣unloaded");
}
module_init(init_seqfile_module);
module_exit(exit_seqfile_module);
```

```
static struct proc_dir_entry *fortune_file = NULL;
static struct proc_dir_entry *fortune_dir = NULL;
static struct proc_dir_entry *fortune_link = NULL;
static char *cookie_buffer = NULL;
static int write_index = 0;
static int read_index = 0;
static char tmp[BUF_SIZE];
ssize_t seq_file_write(struct file *filep, const char __user *buf, size_t
    len, loff_t *offp)
{
  printk(KERN_INFO "+␣seq_file:␣write");
  if(len > BUF_SIZE - write_index + 1)
      // error
  if(copy_from_user(&cookie_buffer[write_index], buf, len))
      // error
  write_index += len;
  cookie_buffer[write_index-1] = '\0';
  return len;
}
static int seq_file_show(struct seq_file *m, void *v)
{
  printk(KERN_INFO "+␣seq_file:␣show");
  if (!write_index)
    return 0;
  if (read_index >= write_index)
  {
        read_index = 0;
    }
    int len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
    seq_printf(m, "%s", tmp);
    read_index += len;
    return 0;
}
static void *seq_file_start(struct seq_file *m, loff_t *pos)
{
    printk(KERN_INFO "+␣seq_file:␣start");
    static unsigned long counter = 0;
    if (!*pos) {
        return &counter;
    }
    else {
        *pos = 0;
        return NULL;
    }
}
static void *seq_file_next(struct seq_file *m, void *v, loff_t *pos)
{
    printk(KERN_INFO "+␣seq_file:␣next");
    unsigned long *tmp = (unsigned long *) v;
    (*tmp)++;
    (*pos)++;
    return NULL;
}
static void seq_file_stop(struct seq_file *m, void *v)
{
    printk(KERN_INFO "+␣seq_file:␣stop");
}
static struct seq_operations seq_file_ops = {
    .start = seq_file_start,
    .next = seq_file_next,
    .stop = seq_file_stop,
    .show = seq_file_show
};
static int seq_file_open(struct inode *i, struct file * f)
```

```
{
  printk(KERN_DEBUG "+␣seq_file:␣open␣seq_file");
  return seq_open(f, &seq_file_ops);
}
static struct proc_ops fops = {
  .proc_open = seq_file_open,
  .proc_read = seq_read,
  .proc_write = seq_file_write,
  .proc_lseek = seq_lseek,
  .proc_release = seq_release,
};
void freemem(void)
{
  if (cookie_buffer)
    vfree(cookie_buffer);
  if (fortune_link)
    remove_proc_entry(SYMLINK, NULL);
  if (fortune_file)
    remove_proc_entry(FILENAME, fortune_dir);
  if (fortune_dir)
    remove_proc_entry(DIRNAME, NULL);
}
static int __init init_seq_file_module(void)
{
  if (!(cookie_buffer = vmalloc(BUF_SIZE)))
      // error
  memset(cookie_buffer, 0, BUF_SIZE);
  if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
      // error
  else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops)))
      // error
  else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
      // error
  write_index = 0;
  read_index = 0;
  printk(KERN_INFO "+␣module␣loaded");
  return 0;
```

```c
}
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);
```

```c
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);
        }
    }
    close(fd[0]);
    close(fd[1]);
    return EXIT_SUCCESS;
}
```

0.1. AF_UNIX + SOCK_DGRAM + без bind у клиента

client

```c
int main(int argc, char ** argv)
{
    char buf[BUF_SIZE];
    if (argc != 2)
        // error
    sprintf(buf, "%d %s", getpid(), argv[1]);
    int sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockfd == -1)
        // error
    struct sockaddr sa;
    sa.sa_family = AF_UNIX;
    strcpy(sa.sa_data, "socket.soc");
    if (sendto(sockfd, buf, sizeof(buf), 0, &sa, sizeof(sa)) == -1)
        // error
    close(sockfd);
    return EXIT_SUCCESS;
}
```

server

```c
int sockfd;
void signal_handler(int signal)
{
    printf("\nCaught signal = %d\n", signal);
    unlink("socket.soc");
    close(sockfd);
    printf("\nServer exiting.\n");
    exit(0);
}
int main()
{
  if ((signal(SIGINT, signal_handler) == SIG_ERR))
      // error
  char buf[BUF_SIZE];
  sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (sockfd == -1)
      // error
  struct sockaddr sa;
  sa.sa_family = AF_UNIX;
  strcpy(sa.sa_data, "socket.soc");
  if (bind(sockfd, &sa, sizeof(sa)) < 0)
      // error
  int bytes;
  while (1)
  {
    bytes = recvfrom(sockfd, buf, sizeof(buf), 0, NULL, NULL);
    if (bytes == -1)
        // error
    printf("\nreceived: %s\n", buf);
  }
  return EXIT_SUCCESS;
}
```

0.2. AF_UNIX + SOCK_DGRAM + bind у клиента

client

```c
int main(int argc, char **argv)
{
    char buf[BUF_SIZE];
    if (argc != 2)
        // error
    sprintf(buf, "%d %s", getpid(), argv[1]);
    int sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockfd == -1)
        // error
    struct sockaddr sa;
    sa.sa_family = AF_UNIX;
    strcpy(sa.sa_data, "socket.soc");
    socklen_t len = sizeof(sa);
    char name[20];
    sprintf(name, "%d.soc", getpid());
    struct sockaddr ca;
    ca.sa_family = AF_UNIX;
    strcpy(ca.sa_data, name);
    if (bind(sockfd, &ca, sizeof(ca)) == -1)
        // error
    if (sendto(sockfd, buf, sizeof(buf), 0, &sa, len) == -1)
        // error
    if (recvfrom(sockfd, buf, sizeof(buf), 0, NULL, NULL) == -1) //&sa, &len
        // error
    printf("\nreceived:␣%s\n", buf);
    unlink(name);
    close(sockfd);
    return EXIT_SUCCESS;
}
```

server

```
int sockfd;
void signal_handler(int signal)
{
    printf("\nCaught␣signal␣=␣%d\n", signal);
    unlink("socket.soc");
    close(sockfd);
    printf("\nServer␣exiting.\n");
    exit(0);
}
int main(int argc, char **argv)
{
```

```c
    if ((signal(SIGINT, signal_handler) == SIG_ERR))
        // error
    char buf[BUF_SIZE];
    sockfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sockfd == -1)
        // error
    struct sockaddr sa;
    sa.sa_family = AF_UNIX;
    strcpy(sa.sa_data, "socket.soc");
    if (bind(sockfd, &sa, sizeof(sa)) == -1)
        // error
    int bytes;
    while (1)
    {
        struct sockaddr ca;
        socklen_t len = sizeof(ca);
        bytes = recvfrom(sockfd, buf, sizeof(buf), 0, &ca, &len);
        if (bytes == -1)
            // error
        printf("\nreceived: %s\n", buf);
        sprintf(buf, "%s %d", buf, getpid());
        if (sendto(sockfd, buf, sizeof(buf), 0, &ca, len) == -1)
            // error
        printf("sent: %s\n", buf);
    }
    return EXIT_SUCCESS;
}
```

```c
MODULE_LICENSE("GPL");
struct tasklet_struct *tasklet;
void tasklet_func(unsigned long data)
{
    printk(KERN_INFO "+: ------------------------------");
    printk(KERN_INFO "+: tasklet began");
    printk(KERN_INFO "+: tasklet count = %u", tasklet->count.counter);
    printk(KERN_INFO "+: tasklet state = %lu", tasklet->state);
    printk(KERN_INFO "+: key code - %d", data);
    if (data < ASCII_LEN)
        printk(KERN_INFO "+: key press - %s", ascii[data]);
    if (data > 128 && data < 128 + ASCII_LEN)
        printk(KERN_INFO "+: key release - %s", ascii[data - 128]);
    printk(KERN_INFO "+: tasklet ended");
    printk(KERN_INFO "+: ----------------------------");
}
static irqreturn_t my_irq_handler(int irq, void *dev_id)
{
  int code;
  printk(KERN_INFO "+: my_irq_handler called\n");
  if (irq != IRQ_NUMBER)
  {
    printk(KERN_INFO "+: irq not handled");
    return IRQ_NONE;
  }
  printk(KERN_INFO "+: tasklet state (before schedule) = %lu",
                tasklet->state);
  code = inb(0x60);
  tasklet->data = code;
  tasklet_schedule(tasklet);
  printk(KERN_INFO "+: tasklet scheduled");
  printk(KERN_INFO "+: tasklet state (after schedule) = %lu",
                 tasklet->state);

  return IRQ_HANDLED;
}
static int __init my_init(void)
{
  if (request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED, "tasklet_irq_handler", (void *) my_irq_handler))
      // error
  tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
  if (tasklet == NULL)
      // error
  tasklet_init(tasklet, tasklet_func, 0);
  printk(KERN_INFO "+: module loaded\n");
  return 0;
}
static void __exit my_exit(void)
```

```
{
    tasklet_kill(tasklet);
    free_irq(IRQ_NUMBER, my_irq_handler);
    printk(KERN_INFO "+:␣module␣unloaded\n");
}
module_init(my_init);
module_exit(my_exit);
```

```
MODULE_LICENSE("GPL");
typedef struct
{
    struct work_struct work;
    int code;
} my_work_struct_t;
static struct workqueue_struct *my_wq;
static my_work_struct_t *work1;
static struct work_struct *work2;
void work1_func(struct work_struct *work)
{
    my_work_struct_t *my_work = (my_work_struct_t *)work;
    int code = my_work->code;
    printk(KERN_INFO "+:␣--------------------------");
    printk(KERN_INFO "+:␣work1␣began");
    printk(KERN_INFO "+:␣key␣code␣-␣%d", code);
    if (code < ASCII_LEN)
        printk(KERN_INFO "+:␣key␣press␣-␣%s", ascii[code]);
    if (code > 128 && code < 128 + ASCII_LEN)
        printk(KERN_INFO "+:␣key␣release␣-␣%s", ascii[code - 128]);
    printk(KERN_INFO "+:␣work1␣ended");
    printk(KERN_INFO "+:␣--------------------------");
}
void work2_func(struct work_struct *work)
{
    printk(KERN_INFO "+:␣--------------------------");
    printk(KERN_INFO "+:␣work2␣began");
    msleep(10);
    printk(KERN_INFO "+:␣work2␣ended");
    printk(KERN_INFO "+:␣--------------------------");
}
```

```
irqreturn_t my_irq_handler(int irq, void *dev)
{
    int code;
    printk(KERN_INFO "+:␣my_irq_handler␣called\n");
    if (irq != IRQ_NUMBER)
    {
        printk(KERN_INFO "+:␣irq␣not␣handled");
        return IRQ_NONE;
    }
    code = inb(0x60);
    work1->code = code;
    queue_work(my_wq, (struct work_struct *)work1);
    queue_work(my_wq, work2);
    return IRQ_HANDLED;
}
static int __init my_init(void)
{
    int rc = request_irq(IRQ_NUMBER, my_irq_handler, IRQF_SHARED,
                         "work_irq_handler", (void *) my_irq_handler);
    if (rc)
        // error
    my_wq = alloc_workqueue("%s", __WQ_LEGACY | WQ_MEM_RECLAIM, 1, "my_wq"
        );
    if (my_wq == NULL)
        // error
    work1 = kmalloc(sizeof(my_work_struct_t), GFP_KERNEL);
    if (work1 == NULL)
        // error
    work2 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);
    if (work2 == NULL)
        // error
    INIT_WORK((struct work_struct *)work1, work1_func);
    INIT_WORK(work2, work2_func);
    printk(KERN_INFO "+:␣module␣loaded");
    return 0;
}
static void __exit my_exit(void)
{
    synchronize_irq(IRQ_NUMBER);
    free_irq(IRQ_NUMBER, my_irq_handler);
    flush_workqueue(my_wq);
    destroy_workqueue(my_wq);
    kfree(work1);
    kfree(work2);
    printk(KERN_INFO "+:␣module␣unloaded");
}
module_init(my_init);
module_exit(my_exit);
```

Код клиента

```
int main(void)
{
  setbuf(stdout, NULL);
  struct sockaddr_in serv_addr =
  {
    .sin_family = AF_INET,
    .sin_addr.s_addr = INADDR_ANY,
    .sin_port = htons(SERVER_PORT)
  };
  socklen_t serv_len;
  char buf[MSG_LEN];
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  // error handling
  if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    // error handling
  char input_msg[MSG_LEN], output_msg[MSG_LEN];
  sprintf(output_msg, "%d", getpid());
  if (write(sock_fd, output_msg, strlen(output_msg) + 1) == -1)
    // error handling
  if (read(sock_fd, input_msg, MSG_LEN) == -1)
    // error handling
  printf("Client␣receive:␣%s␣\n", input_msg);
  close(sock_fd);
  return EXIT_SUCCESS;
}
```

Код сервера

```
int main()
{
  int epoll_fd;
  if ((epoll_fd = epoll_create(CLIENTS_MAX)) == -1)
      // error
  int sock;
  if ((sock = socket(AF_INET, SOCK_STREAM | O_NONBLOCK, 0)) == -1)
      // error
  int sopt = 1;
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sopt, sizeof(sopt)) == -1)
      // error
  struct sockaddr_in addr;
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;
  addr.sin_port = htons(PORT);
  if (bind(sock, (struct sockaddr *) &addr, sizeof(addr)) == -1)
      // error
  if (listen(sock, CLIENTS_MAX) == -1)
      // error
  struct epoll_event epev;
  epev.events = EPOLLIN;
  epev.data.fd = sock;
  if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sock, &epev) == -1)
      // error
  while (1)
  {
    struct epoll_event epev[CLIENTS_MAX + 1];
    int num;
    if ((num = epoll_wait(epoll_fd, &epev, CLIENTS_MAX + 1, -1)) == -1)
        // error
    for (int i = 0; i < num; i++)
    {
      if (epev[i].data.fd == sock)
      {
        int conn;
        if ((conn = accept(sock, NULL, NULL)) == -1)
```

```
                    // error
                struct epoll_event epev;
                int flags = fcntl(conn, F_GETFL, 0);
                fcntl(conn, F_SETFL, flags | O_NONBLOCK);
                epev.events = EPOLLIN | EPOLLET ;
                epev.data.fd = conn;
                if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn, &epev) == -1)
                    // error
            }
            else
            {
                int conn = epev[i].data.fd;
                char received_msg[BUF_SIZE], send_msg[BUF_SIZE];
                if (recv(conn, received_msg, sizeof(received_msg), 0) == -1)
                    // error
                printf("Server %d received message: %s\n", getpid(), received_msg);
                sprintf(send_msg, "%s from server with pid %d", received_msg, getpid());
                if (send(conn, send_msg, sizeof(send_msg), 0) == -1)
                    // error
                printf("Server %d send message: %s\n", getpid(), send_msg);
                close(conn);
            }
        }
    }
    return 0;
}
```

fortune

```
MODULE_LICENSE("GPL");
#define BUF_SIZE PAGE_SIZE
#define DIRNAME "fortunes"
#define FILENAME "fortune"
#define SYMLINK "fortune_link"
#define FILEPATH DIRNAME "/" FILENAME
static struct proc_dir_entry *fortune_dir = NULL;
static struct proc_dir_entry *fortune_file = NULL;
static struct proc_dir_entry *fortune_link = NULL;
static char *cookie_buffer;
static int write_index;
static int read_index;
static char tmp[BUF_SIZE];
ssize_t fortune_read(struct file *filp, char __user *buf, size_t count,
    loff_t *offp)
{
  int len;
  printk(KERN_INFO "+␣fortune:␣read␣called");
  if (*offp > 0 || !write_index)
  {
    printk(KERN_INFO "+␣fortune:␣empty");
    return 0;
  }
  if (read_index >= write_index)
    read_index = 0;
  len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
  if (copy_to_user(buf, tmp, len))
      // error
  read_index += len;
  *offp += len;
  return len;
}
ssize_t fortune_write(struct file *filp, const char __user *buf, size_t
    len, loff_t *offp)
{
  printk(KERN_INFO "+␣fortune:␣write␣called");
  if (len > BUF_SIZE - write_index + 1)
      // error
  if (copy_from_user(&cookie_buffer[write_index], buf, len))
      // error
  write_index += len;
  cookie_buffer[write_index - 1] = '\0';
  return len;
}
int fortune_open(struct inode *inode, struct file *file)
```

```
{
  printk(KERN_INFO "+␣fortune:␣called␣open");
  return 0;
}
int fortune_release(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "+␣fortune:␣called␣release");
  return 0;
}
static const struct proc_ops fops = {
  proc_read: fortune_read,
  proc_write: fortune_write,
  proc_open: fortune_open,
  proc_release: fortune_release
};
static void freemem(void)
{
  if (fortune_link)
    remove_proc_entry(SYMLINK, NULL);
  if (fortune_file)
    remove_proc_entry(FILENAME, fortune_dir);
  if (fortune_dir)
    remove_proc_entry(DIRNAME, NULL);
  if (cookie_buffer)
    vfree(cookie_buffer);
}
static int __init fortune_init(void)
{
  if (!(cookie_buffer = vmalloc(BUF_SIZE)))
      // error
  memset(cookie_buffer, 0, BUF_SIZE);
  if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
      // error
  else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops)))
      // error
  else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
      // error
  write_index = 0;
  read_index = 0;
  printk(KERN_INFO "+ fortune: module loaded");
  return 0;
}
static void __exit fortune_exit(void)
{
  freemem();
  printk(KERN_INFO "+ fortune: module unloaded");
}
module_init(fortune_init)
module_exit(fortune_exit)
```

Вопрос 6

```
#define CACHE_SIZE 1024
#define CACHE_NAME "kittyfs_cache"
static struct kmem_cache *cache = NULL;
static struct kittyfs_inode **inode_cache = NULL;
static size_t cache_index = 0;
static struct kittyfs_inode
{
    int i_mode;
    unsigned long i_ino;
} kittyfs_inode;
static void kittyfs_kill_sb(struct super_block *sb){
    printk(KERN_INFO "+ kittyfs: kill super block");
    kill_anon_super(sb);
}
static void kittyfs_put_sb(struct super_block *sb)
{
    printk(KERN_INFO "+ kittyfs: superblock destroy called");
}
static struct super_operations const kittyfs_sb_ops = {
    .put_super = kittyfs_put_sb,
    .statfs = simple_statfs,
    .drop_inode = generic_delete_inode,
};
static struct inode *kittyfs_new_inode(struct super_block *sb, int ino,
   int mode)
```

```c
{
  struct inode *res;
  res = new_inode(sb);
  if (!res)
    return NULL;
  res->i_ino = ino;
    res->i_mode = mode;
    res->i_atime = res->i_mtime = res->i_ctime = current_time(res);
    res->i_op = &simple_dir_inode_operations;
    res->i_fop = &simple_dir_operations;
    res->i_private = &kittyfs_inode;
    if (cache_index >= CACHE_SIZE)
      return NULL;
    inode_cache[cache_index] = kmem_cache_alloc(cache, GFP_KERNEL);
    if (inode_cache[cache_index])
    {
      inode_cache[cache_index]->i_ino = res->i_ino;
      inode_cache[cache_index]->i_mode = res->i_mode;
      cache_index++;
    }
    return res;
}
static int kittyfs_fill_sb(struct super_block *sb, void *data, int silent)
{
    struct dentry *root_dentry;
    struct inode *root_inode;
    sb->s_blocksize = PAGE_SIZE;
    sb->s_blocksize_bits = PAGE_SHIFT;
    sb->s_magic = MAGIC_NUM;
    sb->s_op = &kittyfs_sb_ops;
    root_inode = kittyfs_new_inode(sb, 1, S_IFDIR | 0755);
    if (!root_inode)
        // error
    root_dentry = d_make_root(root_inode);
    if (!root_dentry)
        // error
    sb->s_root = root_dentry;
    return 0;
}
static struct dentry *kittyfs_mount(struct file_system_type *type, int flags, const char *dev, void *data)
{
    struct dentry *const root_dentry = mount_nodev(type, flags, data, kittyfs_fill_sb);
    if (IS_ERR(root_dentry))
        printk(KERN_ERR "+␣kittyfs:␣cannot␣mount");
    else
        printk(KERN_INFO "+␣kittyfs:␣mount␣successful");
    return root_dentry;
}
static void kittyfs_slab_constructor(void *addr)
{
    memset(addr, 0, sizeof(struct kittyfs_inode));
}
static struct file_system_type kittyfs_type = {
    .owner = THIS_MODULE,
    .name = "kittyfs",
    .mount = kittyfs_mount,
    .kill_sb = kittyfs_kill_sb,
};
static int __init kittyfs_init(void)
{
    int err = register_filesystem(&kittyfs_type);
    if (err != 0)
        // error
    if ((inode_cache = kmalloc(sizeof(struct kittyfs_inode*)*CACHE_SIZE, GFP_KERNEL)) == NULL)
        // error

    if ((cache = kmem_cache_create(CACHE_NAME, sizeof(struct kittyfs_inode), 0, SLAB_HWCACHE_ALIGN, kittyfs_slab_constructor)) == NULL)
        // error
    printk(KERN_INFO "+␣kittyfs:␣module␣loaded");
    return 0;
}
static void __exit kittyfs_exit(void)
```

```
{
    int err;
    int i;
    for (i = 0; i < cache_index; i++)
      kmem_cache_free(cache, inode_cache[i]);
    kmem_cache_destroy(cache);
    kfree(inode_cache);
    err = unregister_filesystem(&kittyfs_type);
    if (err != 0)
        printk(KERN_ERR "+ kittyfs: cannot unregister filesystem");
    else
        printk(KERN_INFO "+ kittyfs: module is unloaded");
}
module_init(kittyfs_init);
module_exit(kittyfs_exit);
```

```
MODULE_LICENSE("GPL");
static int __init mod_init(void)
{
    printk(KERN_INFO " + module is loaded.\n");
    struct task_struct *task = &init_task;
    do
    {
        printk(KERN_INFO " + %s (%d) (%d - state, %d - prio, flags - %d, policy - %d), parent %s (%d), d_name %s",
            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);
    } while ((task = next_task(task)) != &init_task);
    printk(KERN_INFO " + %s (%d) (%d - state, %d - prio, flags - %d, policy - %d), parent %s (%d), d_name %s",
        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);
    return 0;
}
static void __exit mod_exit(void)
{
    printk(KERN_INFO " + %s - %d, parent %s - %d\n", current->comm,
            current->pid, current->parent->comm, current->parent->pid);
    printk(KERN_INFO "␣+␣module␣is␣unloaded.\n");
}
module_init(mod_init);
module_exit(mod_exit);
```

kernel_module.h

```
extern char *module_1_data;
extern char *module_1_proc(void);
extern char *module_1_noexport(void);
```

module_1.c

```
#include "kernel_module.h"
MODULE_LICENSE("GPL");
char *module_1_data = "ABCDE";
extern char *module_1_proc(void) { return module_1_data; }
static char *module_1_local(void) { return module_1_data; }
extern char *module_1_noexport(void) { return module_1_data; }
EXPORT_SYMBOL(module_1_data);
EXPORT_SYMBOL(module_1_proc);
static int __init module_1_init(void)
{
    printk("+␣module_1␣started.\n");
    printk("+␣module_1␣use␣local␣from␣module_1:␣%s\n", module_1_local());
    printk("+␣module_1␣use␣noexport␣from␣module_1:␣%s\n",
        module_1_noexport());
    return 0;
}
static void __exit module_1_exit(void) { printk("+␣module␣module_1␣
    unloaded.\n"); }
module_init(module_1_init);
module_exit(module_1_exit);
```

module_2.c

```
#include "kernel_module.h"
MODULE_LICENSE("GPL");
static int __init module_2_init(void)
{
    printk("+␣module␣module_2␣started.\n");
    printk("+␣data␣string␣exported␣from␣module_1:␣%s\n", module_1_data);
    printk("+␣string␣returned␣module_1_proc():␣%s\n", module_1_proc());
    //printk( "+ module_2 use local from module_1: %s\n", module_1_local());
    //printk( "+ module_2 use noexport from module_1: %s\n", module_1_noexport());
    return 0;
}
static void __exit module_2_exit(void)
{
    printk("+␣module_2␣unloaded.\n");
}
module_init(module_2_init);
module_exit(module_2_exit);
```

module_3.c

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