### 

[Файл md.h](#_f14e88leyzli) 1

[Файл md1.c](#_dyc1tnc68483) 1

[Файл md2.c](#_8y4lvl2gpj2m) 1

[seq\_file](#_oijh83vcvsih) 2

[фортунки (неполный)](#_c62j28e5weyk) 3

[tasklet](#_dypo6nb13uff) 6

[workqueue](#_naraxr897hg4) 7

[селект + af\_inet + sock\_stream](#_jo4sk6vtfs62) 8

### Файл md.h

| **extern** **char**\* md1\_data;**extern** **char**\* **md1\_proc**(**void**); |
| --- |

### Файл md1.c

| **#include "md.h"** MODULE\_LICENSE("GPL"); **char** \*md1\_data = "Hello world!";  **extern** **char** \***md1\_proc**(**void**) { **return** md1\_data;}  EXPORT\_SYMBOL(md1\_data); EXPORT\_SYMBOL(md1\_proc);  **static** **int** \_\_init **md\_init**(**void**) { printk("+ module md1 start!\n"); **return** 0;}  **static** **void** \_\_exit **md\_exit**(**void**) { printk("+ module md1 unloaded!\n");}  module\_init(md\_init); module\_exit(md\_exit); |
| --- |

### Файл md2.c

| **#include "md.h"** MODULE\_LICENSE("GPL"); **static** **int** \_\_init **md\_init**(**void**) { printk("+ module md2 start!\n");  printk("+ data string exported from md1 : %s\n", md1\_data);  printk("+ string returned md1\_proc() is : %s\n", md1\_proc());  **return** 0;}  **static** **void** \_\_exit **md\_exit**(**void**) { printk("+ module md2 unloaded!\n");}  module\_init(md\_init); module\_exit(md\_exit); |
| --- |

### seq\_file

| **#include ///** **static** **int** limit = 10; *//значение по умолчанию, его можно менять здесь* **static** **int**\* even\_ptr; *//усложняющий пример указатель, демонстрирующий, как работатьс динамической памятью*  **static** **void** \***ct\_seq\_start**(struct seq\_file \*s, **loff\_t** \*pos) {  *//Выделяем память для числа, которое будет хранить наше растущее четное значение*  even\_ptr = kmalloc(**sizeof**(**int**), GFP\_KERNEL);  **if** (!even\_ptr) **return** NULL;  printk(KERN\_INFO "In start(), even\_ptr = %pX.\n", even\_ptr);  \*even\_ptr = (\*pos)\*2;  **return** even\_ptr; } **static** **int** **ct\_seq\_show**(struct seq\_file \*s, **void** \*v) {  printk(KERN\_INFO "In show(), even = %d.\n", \*(**int**\*)v);  seq\_printf(s, "The current value of the even number is %d\n", \*(**int**\*)v);  **return** 0;}  **static** **void** \***ct\_seq\_next**(struct seq\_file \*s, **void** \*v, **loff\_t** \*pos) {  (\*pos)++; *//увеличиваем счётчик*  \*(**int**\*)v += 2; *//переходим к следующему чётному*  **return** v; }  **static** **void** **ct\_seq\_stop**(struct seq\_file \*s, **void** \*v) {  printk(KERN\_INFO "In stop(), even\_ptr = %pX.\n", even\_ptr);}  **static** **struct** **seq\_operations** **ct\_seq\_ops** = {  .start = ct\_seq\_start,  .next = ct\_seq\_next,  .stop = ct\_seq\_stop,  .show = ct\_seq\_show };  **static** **int** **ct\_open**(struct inode \*inode, struct file \*file) {  **return** seq\_open(file, &ct\_seq\_ops);};  **static** **struct** **file\_operations** **ct\_file\_ops** = {  .owner = THIS\_MODULE,  .open = ct\_open,  .read = seq\_read,  .llseek = seq\_lseek,  .release = seq\_release };  **static** **int** \_\_init **ct\_init**(**void**) {  proc\_create("evens", 0, NULL, &ct\_file\_ops);  **return** 0; }  **static** **void** \_\_exit **ct\_exit**(**void**) {  remove\_proc\_entry("evens", NULL); }  module\_init(ct\_init); module\_exit(ct\_exit);  MODULE\_LICENSE("GPL"); |
| --- |

### фортунки (неполный)

| **#include …** MODULE\_LICENSE("GPL"); **#define FORTUNE\_PATH FORTUNE\_FILENAME** **#define MAX\_COOKIE\_BUF\_SIZE PAGE\_SIZE**  **static** **struct** **proc\_dir\_entry** \***fortune\_dir**; **static** **struct** **proc\_dir\_entry** \***fortune\_file**; **static** **struct** **proc\_dir\_entry** \***fortune\_symlink**;  **static** **char** \*cookie\_buffer = NULL; *// хранилище фортунок* **static** **int** read\_index = 0; **static** **int** write\_index = 0; **char** tmp\_buffer[MAX\_COOKIE\_BUF\_SIZE];  **static** **int** **fortune\_open**(struct inode \*sp\_inode, struct file \*sp\_file){  printk(KERN\_INFO " MY\_FORTUNE: %s called.\n", \_\_func\_\_);  **return** OK;}  **static** **int** **fortune\_release**(struct inode \*sp\_node, struct file \*sp\_file){  printk(KERN\_INFO " MY\_FORTUNE: %s called.\n", \_\_func\_\_);  **return** OK;}  **static** ssize\_t **fortune\_write**(struct file \*file, **const** **char** \_\_user \*buf, **size\_t** len, **loff\_t** \*f\_pos){  ...  **if** (copy\_from\_user(&cookie\_buffer[write\_index], buf, len) != 0) {  **return** -EFAULT; }  ...  **return** len; *// количество символов фактически записанных в cookie\_buffer*}  **static** ssize\_t **fortune\_read**(struct file \*file, **char** \_\_user \*buf, **size\_t** len, **loff\_t** \*f\_pos){  **int** read\_len;  ...  **if** (copy\_to\_user(buf, tmp\_buffer, read\_len) != 0) {  **return** -EFAULT; }  ...  **return** read\_len;}  **static** **const** **struct** **proc\_ops** **fops** = {  proc\_read: fortune\_read,  proc\_write: fortune\_write,  proc\_open: fortune\_open,  proc\_release: fortune\_release,};  **static** **void** **cleanup\_fortune**(**void**){  **if** (fortune\_symlink != NULL) {  remove\_proc\_entry(FORTUNE\_SYMLINK, NULL); }  **if** (fortune\_file != NULL) {  remove\_proc\_entry(FORTUNE\_FILENAME, NULL); }  **if** (fortune\_dir != NULL) {  remove\_proc\_entry(FORTUNE\_DIRNAME, NULL); }  vfree(cookie\_buffer);}  **static** **int** \_\_init **fortune\_init**(**void**){  *// Выделение 'виртуально' непрерывного блока памяти*  **if** ((cookie\_buffer = vmalloc(MAX\_COOKIE\_BUF\_SIZE)) == NULL) {  **return** -ENOMEM; }   memset(cookie\_buffer, 0, MAX\_COOKIE\_BUF\_SIZE);  **if** ((fortune\_dir = proc\_mkdir(FORTUNE\_DIRNAME, NULL)) == NULL) {  cleanup\_fortune();  **return** -ENOMEM; }   **if** ((fortune\_file = proc\_create(FORTUNE\_FILENAME, 0666, NULL, &fops)) == NULL) {  cleanup\_fortune();  **return** -ENOMEM; }   **if** ((fortune\_symlink = proc\_symlink(FORTUNE\_SYMLINK, NULL, FORTUNE\_PATH)) == NULL) {  cleanup\_fortune();  **return** -ENOMEM; }   write\_index = 0;  read\_index = 0;   **return** OK; }  **static** **void** \_\_exit **fortune\_exit**(**void**){  cleanup\_fortune();} module\_init(fortune\_init); module\_exit(fortune\_exit); |
| --- |

открываем 2 раза и пишем туда аааааааа потом ббббб

| **#include <fcntl.h> // O\_RDWR** **#include <unistd.h> // write, close** **#include <string.h> // strlen** **#include <stdio.h>**  **int** **main**() {  printf("fff)");  **int** fd1 = open("text.txt", O\_RDWR);  **int** fd2 = open("text.txt", O\_RDWR);   **char** \*data1 = "aaaaaaaaaaaa";  **char** \*data2 = "bbbb";   write(fd1, data1, strlen(data1));  write(fd2, data2, strlen(data2));   close(fd1);  close(fd2);   **return** 0; } |
| --- |

vfs + slab

| **#include ///**  MODULE\_LICENSE("GPL"); **#define MYFS\_MAGIC\_NUMBER 0x13131313** **#define SLABNAME "my\_cache"** **struct** **myfs\_inode** {  **int** i\_mode;  **unsigned** **long** i\_ino; } myfs\_inode;  **static** **int** sco = 0; **static** **int** number = 31; **struct** **kmem\_cache** \***cache** = **NULL**; **static** **void** \*\*line = NULL; **static** **int** size = **sizeof**(struct myfs\_inode);  **static** **void** **myfs\_put\_super**(struct super\_block\* sb){  printk(KERN\_INFO ">>> MYFS super block destroyed!\n");}  **static** **struct** **super\_operations** **const** **myfs\_super\_ops** ={  .put\_super = myfs\_put\_super,  .statfs = simple\_statfs,  .drop\_inode = generic\_delete\_inode, };  **static** struct inode \***myfs\_make\_inode**(struct super\_block \* sb, **int** mode){  **struct** **inode** \***ret** = **new\_inode**(**sb**);  **if** (ret) {  inode\_init\_owner(&init\_user\_ns, ret, NULL, mode);   ret->i\_size = PAGE\_SIZE;  ret->i\_atime = ret->i\_mtime = ret->i\_ctime = current\_time(ret);  ret->i\_private = &myfs\_inode; }  **return** ret;}  **static** **int** **myfs\_fill\_sb**(struct super\_block\* sb, **void**\* data, **int** silent){  **struct** **inode**\* **root** = **NULL**;   sb->s\_blocksize = PAGE\_SIZE;  sb->s\_blocksize\_bits = PAGE\_SHIFT;  sb->s\_magic = MYFS\_MAGIC\_NUMBER;  sb->s\_op = &myfs\_super\_ops;   root = myfs\_make\_inode(sb, S\_IFDIR | 0755);  ...  root->i\_op = &simple\_dir\_inode\_operations;  root->i\_fop = &simple\_dir\_operations;  sb->s\_root = d\_make\_root(root);  ...  **return** 0;}  **static** struct dentry\* **myfs\_mount**(struct file\_system\_type \*type, **int** flags, **char** **const** \*dev, **void** \*data) {  **struct** **dentry**\* **const** **entry** = **mount\_nodev**(**type**, **flags**, **data**, **myfs\_fill\_sb**) ;    **if** (IS\_ERR(entry))  **else**  **return** entry;}   **static** **struct** **file\_system\_type** **myfs\_type** = {  .owner = THIS\_MODULE,  .name = "myfs",  .mount = myfs\_mount,  .kill\_sb = kill\_anon\_super, };  **static** **int** \_\_init **md\_init**(**void**) {  **int** ret;  line = kmalloc(**sizeof**(**void**\*), GFP\_KERNEL);  **if**(!line) // ошибка   cache = kmem\_cache\_create(SLABNAME, size, 0, 0, co);  **if**(!cache) // ошибка   **if** (!((\*line) = kmem\_cache\_alloc(cache, GFP\_KERNEL))) {  kmem\_cache\_free(cache, \*line);  kmem\_cache\_destroy(cache);  kfree(line);  **return** -ENOMEM; }    ret = register\_filesystem(&myfs\_type);  **if** (ret) {  ошибка }     **return** 0; }  **static** **void** \_\_exit **md\_exit**(**void**){  kmem\_cache\_free(cache, \*line);  kmem\_cache\_destroy(cache);  kfree(line);    **if** (unregister\_filesystem(&myfs\_type))  ... } module\_init(md\_init); module\_exit(md\_exit); |
| --- |

### tasklet

| **#include ///** **#define IRQ\_NUM 1 *// прерывание от клавиатуры***  MODULE\_LICENSE("GPL"); **char** \*my\_tasklet\_data = "my\_tasklet data"; **struct** **tasklet\_struct**\* **my\_tasklet**;  **void** **my\_tasklet\_function**(**unsigned** **long** data){  **int** code = inb(0x60);  **char** \* ascii[84] = {" ", "Esc", "1", и тд};  **if** (code < 84) // печать символа }  **irqreturn\_t** my\_handler(**int** irq, **void** \*dev) {  printk(">> my\_tasklet: " "(handler) (time=%llu): \n", ktime\_get());  **if** (irq == IRQ\_NUM) {  tasklet\_schedule(my\_tasklet); *// планирование тасклета*  **return** IRQ\_HANDLED; *// прерывание обработано*  }  **return** IRQ\_NONE; *// прерывание не обработано* }  **static** **int** \_\_init **my\_init**(**void**) {  *//гарантированно получаете непрерывные физические блоки памяти*  my\_tasklet = kmalloc(**sizeof**(struct tasklet\_struct), GFP\_KERNEL);   **if** (!my\_tasklet) {  **return** -1;  }  *//Динамическое создание (регистрация) тасклета.*  tasklet\_init(my\_tasklet, my\_tasklet\_function, (**unsigned** **long**)my\_tasklet\_data);   **if** (request\_irq(IRQ\_NUM, my\_handler, IRQF\_SHARED, "my\_dev\_name", &my\_handler)) {  **return** -1; }  printk(">> my\_tasklet: " "module loaded\n");  **return** 0; }  **static** **void** \_\_exit **my\_exit**(**void**){  *//ждет завершения тасклета и удаляет тасклет из очереди на выполнение только в контексте процесса.*   tasklet\_kill(my\_tasklet);  kfree(my\_tasklet);  *//освободит линию irq от указанного обработчика.*   free\_irq(IRQ\_NUM, &my\_handler);}  module\_init(my\_init)  module\_exit(my\_exit) |
| --- |

### workqueue

| **#include ...** MODULE\_LICENSE("GPL"); **#define IRQ 1** **static** **int** devID; **struct** **workqueue\_struct** \***workQueue**; **char** \*simbol;  **void** **queue\_functionF**(struct work\_struct \*work){  **int** code = inb(0x60);  **char** \* ascii[84] = {" ", "Esc", "1", "2", и тд};    **if** (code < 84) // печать символа  printk(KERN\_INFO "<<<my\_Queue: Key was clicked (1 worker) and we sleep\n");  msleep(10);  printk(KERN\_INFO "<<<my\_Queue: Key was clicked (1 worker) return\n"); }  **void** **queue\_functionS**(struct work\_struct \*work){  **atomic64\_t** data64 = work->data;  **long** **long** data = data64.counter;   printk(KERN\_INFO "<<<my\_Queue: queue\_functionS data = %lld\n", data);  printk(KERN\_INFO "<<<my\_Queue: Key was clicked (2 worker)\n");}  **struct** **work\_struct** **fWork**; **struct** **work\_struct** **sWork**;  **irqreturn\_t** handler(**int** irq, **void** \*dev){  printk(KERN\_INFO "<<<my\_Queue: move work to queue...\n");  **if** (irq == IRQ) {  *// Помещаем структуру в очередь работ.*  *// queue\_work назначает работу текущему процессору.*  queue\_work(workQueue, &fWork);  queue\_work(workQueue, &sWork);  **return** IRQ\_HANDLED; }  **return** IRQ\_NONE;}  **static** **int** \_\_init **work\_queue\_init**(**void**){  **int** ret = request\_irq(IRQ, */\* номер irq \*/*  handler, */\* наш обработчик \*/*  IRQF\_SHARED, */\* линия может быть разделена, IRQ (разрешено совместное использование)\*/*  "my\_irq2\_handler", */\* имя устройства (можно потом посмотреть в /proc/interrupts)\*/*  &devID); */\* Последний параметр (идентификатор устройства) irq\_handler нужен  для того, чтобы можно отключить с помощью free\_irq \*/*  **if** (ret) ошибка   **if** (!(workQueue = create\_workqueue("my\_queue"))) *ошибка*   INIT\_WORK(&fWork, queue\_functionF);  INIT\_WORK(&sWork, queue\_functionS);   printk(KERN\_INFO "<<<my\_Queue: module loaded\n");  **return** 0; }  **static** **void** \_\_exit **work\_queue\_exit**(**void**){  *// Принудительно завершаем все работы в очереди.*  *// Вызывающий блок блокируется до тех пор, пока операция не будет завершена.*  flush\_workqueue(workQueue);  destroy\_workqueue(workQueue);  free\_irq(IRQ, &devID);  printk(KERN\_INFO "<<<my\_Queue: module unloaded\n"); }  module\_init(work\_queue\_init)  module\_exit(work\_queue\_exit) |
| --- |

### селект + af\_inet + sock\_stream

| **#include ///** **#define BUF\_SIZE 256** **#define PORT 3425** **#define MAX\_CLIENTS\_NUMB 5** **int** sock\_fd; **void** **del\_socket**(**void**) { close(sock\_fd); }  **void** **sigtstp\_handler**(**int** signum) {  del\_socket();  exit(0); }  **void** **receive\_from\_clients**(**int** \*clients, fd\_set \*fd\_readset) {  **char** buf[BUF\_SIZE];  **int** bytes;   **for** (**int** i = 0; i < MAX\_CLIENTS\_NUMB; i++) {  *// FD\_ISSET проверяет, является ли описатель частью набора*   **if** (FD\_ISSET(clients[i], fd\_readset)) {  bytes = recv(clients[i], buf, BUF\_SIZE, 0);  **if** (bytes <= 0) {  *// Соединение разорвано, удаляем сокет из множества*  }  **else** **if** (bytes > 0) {  buf[bytes] = 0;  memset(buf, 0, BUF\_SIZE);  sprintf(buf, "Message recieved\n");  send(clients[i], buf, **sizeof**(buf), 0);  } } }}  **int** **main**(**void**) {  **struct** **sockaddr\_in** **serv\_addr**;  fd\_set fd\_readset;  **int** clients[MAX\_CLIENTS\_NUMB] = {0};    sock\_fd = socket(AF\_INET, SOCK\_STREAM, 0);  **if** (sock\_fd == -1) // ошибка    signal(SIGTSTP, sigtstp\_handler); *//изменение обработчика сигнала*    fcntl(sock\_fd, F\_SETFL, O\_NONBLOCK);   serv\_addr.sin\_family = AF\_INET;  serv\_addr.sin\_addr.s\_addr = INADDR\_ANY; *//сервер зарег на всех адресах той машины, на которой она выполняется.*  serv\_addr.sin\_port = htons(PORT);   **if** (bind(sock\_fd, (struct sockaddr \*) &serv\_addr, **sizeof**(serv\_addr)) == -1) // ошибка   **if** (listen(sock\_fd, MAX\_CLIENTS\_NUMB) == -1) // ошибка  printf("Listening.\nPress Ctrl + Z to stop...\n");   **while**(1) {  FD\_ZERO(&fd\_readset);  FD\_SET(sock\_fd, &fd\_readset);    **int** max\_fd = sock\_fd;  **for** (**int** i = 0; i < MAX\_CLIENTS\_NUMB; i++) {  **if** (clients[i]) {  FD\_SET(clients[i], &fd\_readset); }    **if** (max\_fd < clients[i]) { max\_fd = clients[i];}}  *// Ждём события в одном из сокетов*  **if** (select(max\_fd + 1, &fd\_readset, NULL, NULL, NULL) <= 0) // ошибка   **if** (FD\_ISSET(sock\_fd, &fd\_readset)) {  *// Поступил новый запрос на соединение (sock\_fd - слушающий сокет)*  *// блокируется, пока не появится "свисток" connect (NULL - адрес и длина адреса запрос, неважны)*  **int** new\_sock = accept(sock\_fd, NULL, NULL);  **if** (new\_sock == -1) // ошибка   fcntl(new\_sock, F\_SETFL, O\_NONBLOCK);  **int** place\_not\_found = 1;  **for** (**int** i = 0; i < MAX\_CLIENTS\_NUMB && place\_not\_found; i++) {  **if** (!clients[i]) {  clients[i] = new\_sock;  printf("New connection %d\n", i);  place\_not\_found = 0;  }}}  receive\_from\_clients(clients, &fd\_readset); }  **return** EXIT\_SUCCESS;} |
| --- |