1.1 Сокеты + fork

Server.c

```c
#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <netinet/in.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/stat.h>

#include "common.h"

int sfd, semid;

#define READER_QUEUE 0
#define WRITER_QUEUE 1
#define READER 2
#define WRITER 3
#define SHADOW_WRITER 4
struct sembuf start_rd[] = {{READER_QUEUE, 1, 0},
                            {WRITER_QUEUE, 0, 0},
                            {SHADOW_WRITER, 0, 0},
                            {READER, 1, 0},
                            {READER_QUEUE, -1, 0}};

struct sembuf stop_rd[] = {{READER, -1, 0}};

struct sembuf start_wr[] = {{WRITER_QUEUE, 1, 0},
                            {READER, 0, 0},
                            {WRITER, -1, 0},
                            {SHADOW_WRITER, 1, 0},
                            {WRITER_QUEUE, -1, 0}};

struct sembuf stop_wr[] = {{SHADOW_WRITER, -1, 0},
                           {WRITER, 1, 0}};

void process_client(int connfd, char *arr, int semid)
{
    char buf[BUF_SIZE];
    unsigned index;
    int status;

    printf("[%d] || Got new connection!\n", getpid());

    while (1)
    {
        if (recv(connfd, buf, BUF_SIZE, 0) <= 0)
        {
            printf("[%d] || Server finished\n", getpid());
            break;
        }


        switch (buf[0])
        {
        case 'r':
            if (semop(semid, start_rd, 5) == -1)
            {
                perror("semop");
                exit(1);
            }
            for (size_t i = 0; i < ARR_SIZE; i++)
            {
                buf[i] = arr[i];
            }
            if (send(connfd, &buf, sizeof(buf), 0) == -1)
            {
                perror("error send");
                exit(1);
            }

            if (semop(semid, stop_rd, 1) == -1)
            {
                perror("semop");
                exit(1);
            }

            break;

        case 'w':
            index = (int) buf[1];
            if (semop(semid, start_wr, 5) == -1)
            {
                perror("semop");
                exit(1);
            }

            if (index < ARR_SIZE && arr[index] != ' ')
            {
                arr[index] = ' ';
                status = OK;
                printf("[%d] || Client reserved seat %u\n", getpid(),
index);
            }
            else
            {
                if (arr[index] == ' ')
                {
                    status = ALREADY_RESERVED;
                    printf("[%d] || Client failed to reserve seat %u\n",
getpid(), index);
                }
                else
                {
                    status = ERROR;
                    printf("[%d] || Server received invalid seat number
%u\n", getpid(), index);
                }
            }

            int full = 1;
            for (size_t i = 0; i < ARR_SIZE; i++)
            {
                if (arr[i] != ' ')
                {
                    full = 0;
                    break;
                }
            }

            if (full)
            {
                printf("[%d] || All seats reserved. Shutting down server.
\n", getpid());
                kill(getppid(), SIGINT); // Signal parent process to
shutdown
            }

            if (send(connfd, &status, sizeof(status), 0) == -1)
            {
                perror("error send");
                exit(1);
            }

            if (semop(semid, stop_wr, 2) == -1)
            {
                perror("semop");
                exit(1);
            }
            break;
        }
    }
}

void sigint_handler()
{
    close(sfd);
    //semctl(semid, 5, IPC_RMID, NULL);
    exit(0);
}
```

2.1 Сокеты + pthread (только сервер, клиент в 1)

```c
#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <netinet/in.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <pthread.h>

#include "common.h"

int sfd, semid;

char arr[ARR_SIZE];

#define READER_QUEUE 0
#define WRITER_QUEUE 1
#define READER 2
#define WRITER 3
#define SHADOW_WRITER 4
struct sembuf start_rd[] = {{READER_QUEUE, 1, 0},
                            {WRITER_QUEUE, 0, 0},
                            {SHADOW_WRITER, 0, 0},
                            {READER, 1, 0},
                            {READER_QUEUE, -1, 0}};

struct sembuf stop_rd[] = {{READER, -1, 0}};

struct sembuf start_wr[] = {{WRITER_QUEUE, 1, 0},
                            {READER, 0, 0},
                            {WRITER, -1, 0},
                            {SHADOW_WRITER, 1, 0},
                            {WRITER_QUEUE, -1, 0}};

struct sembuf stop_wr[] = {{SHADOW_WRITER, -1, 0},
                           {WRITER, 1, 0}};

void* process_client(void *args)
{
    unsigned index;
    int connfd = *((int *)args);
    char buf[BUF_SIZE];
    int status;
    int full;

    printf("[%d] || Got new connection!\n", getpid());

    while (1)
    {
        if (recv(connfd, buf, BUF_SIZE, 0) <= 0)
        {
            printf("[%d] || Server finished\n", getpid());
            break;
        }


        switch (buf[0])
        {
            case 'r':
                if (semop(semid, start_rd, 5) == -1)
                {
                    perror("semop");
                    exit(1);
                }
                for (size_t i = 0; i < ARR_SIZE; i++)
                {
                    buf[i] = arr[i];
                }
                if (send(connfd, &buf, sizeof(buf), 0) == -1)
                {
                    perror("error send");
                    exit(1);
                }

                if (semop(semid, stop_rd, 1) == -1)
                {
                    perror("semop");
                    exit(1);
                }

                break;

            case 'w':
                index = (int) buf[1];
                if (semop(semid, start_wr, 5) == -1)
                {
                    perror("semop");
                    exit(1);
                }

                if (index < ARR_SIZE && arr[index] != ' ')
                {
                    arr[index] = ' ';
                    status = OK;
                    printf("[%d] || Client reserved seat %u\n",
getpid(), index);
                }
                else
                {
                    if (arr[index] == ' ')
                    {
                        status = ALREADY_RESERVED;
                        printf("[%d] || Client failed to reserve
seat %u\n", getpid(), index);
                    }
                    else
                    {
                        status = ERROR;
                        printf("[%d] || Server received invalid seat
number %u\n", getpid(), index);
                    }
                }

                full = 1;
                for (size_t i = 0; i < ARR_SIZE; i++)
                {
                    if (arr[i] != ' ')
                    {
                        full = 0;
                        break;
                    }
                }

                if (full)
                {
                    printf("[%d] || All seats reserved. Shutting
down server.\n", getpid());
                    kill(getppid(), SIGINT); // Signal parent
process to shutdown
                }

                if (send(connfd, &status, sizeof(status), 0) == -1)
                {
                    perror("error send");
                    exit(1);
                }

                if (semop(semid, stop_wr, 2) == -1)
                {
                    perror("semop");
                    exit(1);
                }
                break;
        }
    }
    return NULL;
}

void sigint_handler()
{
    close(sfd);
    semctl(semid, 5, IPC_RMID, NULL);
    exit(0);
}
```

3.1 Сокеты + epoll (только сервер, клиент в 1)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include "common.h"

#define MAX_EVENTS 10
volatile sig_atomic_t running = 1;

// Неблокирующий режим для сокета
int set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl F_GETFL");
        return -1;
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl F_SETFL O_NONBLOCK");
        return -1;
    }
    return 0;
}

int has_free_seats(char *seats) {
    for (int i = 0; i < SEATS_COUNT; i++) {
        if (seats[i] >= 'a' && seats[i] <= 'z') {
            return 1;
        }
    }
    return 0;
}

void sigint_handler(int sig) {
    printf("\nПолучен сигнал завершения. Завершаем работу
сервера...\n");
    running = 0;
}

void process_client_request(int conn_fd, char *seats) {
    char recv_buffer[MAX_MSG_LEN];
    char send_buffer[MAX_MSG_LEN];
    pid_t client_pid;
    int seat_num;
    MessageType msg_type;

    ssize_t bytes = recv(conn_fd, recv_buffer, MAX_MSG_LEN,
0);
    if (bytes <= 0) {
        if (bytes == 0) {
            printf("Клиент отключился\n");
        } else {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                // Нет доступных данных для чтения, но
соединение по-прежнему активно.
                return;
            }
            perror("recv failed");
        }
        close(conn_fd);
        return;
    }

    if (sscanf(recv_buffer, "%d %d", &msg_type, &client_pid) <
2) {
        fprintf(stderr, "Неверный формат сообщения: %s\n",
recv_buffer);
        close(conn_fd);
        return;
    }

    switch (msg_type) {
        case READER:
            printf("Клиент [%d] запросил информацию о
местах\n", client_pid);

            int free_seats = has_free_seats(seats);

            // Строка ответа
            snprintf(send_buffer, MAX_MSG_LEN, "%d %d %s",
READER_RESPONSE, free_seats, seats);

            send(conn_fd, send_buffer, strlen(send_buffer) +
1, 0);
            break;

        case WRITER:
            if (sscanf(recv_buffer, "%d %d %d", &msg_type,
&client_pid, &seat_num) != 3) {
                fprintf(stderr, "Неверный формат сообщения
бронирования: %s\n", recv_buffer);
                close(conn_fd);
                return;
            }

            printf("Клиент [%d] пытается забронировать место
индекс %d\n", client_pid, seat_num);
            int success = 0;

            if (seat_num < 0 || seat_num >= SEATS_COUNT) {
                success = 0;
            }
            else {
                if (seats[seat_num] >= 'a' && seats[seat_num]
<= 'z') {
                    seats[seat_num] = 'X';
                    success = 1;
                    printf("Клиент [%d] успешно забронировал
место %c (индекс %d)\n",
                           client_pid, 'a' + seat_num,
seat_num);
                }
            }

            // Ответ о результате бронирования
            snprintf(send_buffer, MAX_MSG_LEN, "%d %d",
WRITER_RESPONSE, success);
            send(conn_fd, send_buffer, strlen(send_buffer) +
1, 0);
            break;

        default:
            fprintf(stderr, "Неизвестный тип сообщения: %d\n",
msg_type);
            close(conn_fd);
            return;
    }
}
```

3.2

```
int main() {
    int listen_fd, conn_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);

    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction failed");
        exit(EXIT_FAILURE);
    }

    shmctl(shmget(SHM_KEY, 0, 0), IPC_RMID, NULL);


    int shmid = shmget(SHM_KEY, SEATS_COUNT + 1, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget failed");
        exit(EXIT_FAILURE);
    }

    char *seats = shmat(shmid, NULL, 0);
    if (seats == (void *)-1) {
        perror("shmat failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < SEATS_COUNT; i++) {
        seats[i] = 'a' + i;
    }
    seats[SEATS_COUNT] = '\0';

    // Серверный сокет
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd == -1) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    int opt = 1;
    if (setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
== -1) {
        perror("setsockopt failed");
        exit(EXIT_FAILURE);
    }

    // Устанавливаем неблокирующий режим для серверного сокета
    if (set_nonblocking(listen_fd) == -1) {
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(listen_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(listen_fd, 5) == -1) {
        perror("listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Server started on port %d\n", PORT);

    int epollfd = epoll_create1(0);
    if (epollfd == -1) {
        perror("epoll_create1 failed");
        exit(EXIT_FAILURE);
    }

    // Добавление серверного сокета в epoll
    struct epoll_event ev, events[MAX_EVENTS];
    ev.events = EPOLLIN;
    ev.data.fd = listen_fd;
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) {
        perror("epoll_ctl: listen_fd");
        exit(EXIT_FAILURE);
    }

    while (running) {
        int nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
        if (nfds == -1) {
            //if (errno == EINTR) {
              //  continue;
            //}
            perror("epoll_wait");
            exit(EXIT_FAILURE);
        }

        for (int n = 0; n < nfds; ++n) {
            if (events[n].data.fd == listen_fd) {
                // обработка данных от сервера
                conn_fd = accept(listen_fd, (struct sockaddr *)&client_addr,
&client_len);
                if (conn_fd == -1) {
                    if (errno == EAGAIN || errno == EWOULDBLOCK) {
                        // Нет больше подключений
                        continue;
                    } else {
                        perror("accept failed");
                        exit(EXIT_FAILURE);
                    }
                }

                // Устанавливаем неблокирующий режим для клиентского сокета
                if (set_nonblocking(conn_fd) == -1) {
                    close(conn_fd);
                    exit(EXIT_FAILURE);
                }

                // Добавляем клиентский сокет в epoll (edge-triggered mode)
                ev.events = EPOLLIN | EPOLLET;
                ev.data.fd = conn_fd;
                if (epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_fd, &ev) == -1) {
                    perror("epoll_ctl: conn_fd");
                    close(conn_fd);
                    exit(EXIT_FAILURE);
                }

                printf("Новое подключение принято\n");
            } else {
                // Обработка данных от клиента
                process_client_request(events[n].data.fd, seats);
            }
        }
    }

    shmdt(seats);
    shmctl(shmid, IPC_RMID, NULL);
    close(listen_fd);
    close(epollfd);

    return 0;
}
```

2.2

```
int main(void)
{
    int listenfd, connfd;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;
    pthread_t th;

    if (signal(SIGINT, sigint_handler) == (void *)-1)
    {
        perror("cannot set handler");
        exit(1);
    }

    semid = semget(IPC_PRIVATE, 5, IPC_CREAT | 0666);
    if (semid == -1)
        perror("semget");
    if (semctl(semid, READER_QUEUE, SETVAL, 0) == -1)
        perror("semctl");
    if (semctl(semid, WRITER_QUEUE, SETVAL, 0) == -1)
        perror("semctl");
    if (semctl(semid, READER, SETVAL, 0) == -1)
        perror("semctl");
    if (semctl(semid, WRITER, SETVAL, 1) == -1)
        perror("semctl");
    if (semctl(semid, SHADOW_WRITER, SETVAL, 0) == -1)
        perror("semctl");

    for (size_t i = 0; i < ARR_SIZE; i++)
        arr[i] = (char)('a' + i);

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1)
    {
        perror("error socket");
        exit(1);
    }

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    if (bind(listenfd, (struct sockaddr *)&servaddr,
sizeof(servaddr)) == -1)
    {
        perror("error bind");
        exit(1);
    }

    if (listen(listenfd, 1024) == -1)
    {
        perror("error listen");
        exit(1);
    }

    printf("listening on port %d\n", SERV_PORT);

    while (1)
    {
        clilen = sizeof(cliaddr);
        connfd = accept(listenfd, (struct sockaddr
*)&cliaddr, &clilen);

        if (pthread_create(&th, NULL, process_client,
&connfd) != 0)
        {
            perror("pthread_create");
            exit(1);
        }

        if (pthread_detach(th))
        {
            perror("pthread_detach");
            exit(1);
        }
        while (waitpid(-1, NULL, WNOHANG) > 0);
    }

    if (semctl(semid, 5, IPC_RMID, NULL) == -1)
    {
        perror("semclt");
        exit(1);
    }
}
```

1.2

```
int main(void)
{
    int listenfd, connfd, shmid;
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;

    if (signal(SIGINT, sigint_handler) == (void *)-1)
    {
        perror("cannot set handler");
        exit(1);
    }

    shmid = shmget(IPC_PRIVATE, BUF_SIZE, IPC_CREAT | 0666);
    if (shmid == -1)
    {
        perror("error shmget");
        exit(1);
    }


    semid = semget(IPC_PRIVATE, 5, IPC_CREAT | 0666);
    if (semid == -1)
        perror("semget");
    if (semctl(semid, READER_QUEUE, SETVAL, 0) == -1)
        perror("semctl");
    if (semctl(semid, WRITER_QUEUE, SETVAL, 0) == -1)
        perror("semctl");
    if (semctl(semid, READER, SETVAL, 0) == -1)
        perror("semctl");
    if (semctl(semid, WRITER, SETVAL, 1) == -1)
        perror("semctl");
    if (semctl(semid, SHADOW_WRITER, SETVAL, 0) == -1)
        perror("semctl");

    char *addr = shmat(shmid, NULL, 0);
    if (addr == (void *)-1)
    {
        perror("error shmat");
        exit(1);
    }

    for (size_t i = 0; i < ARR_SIZE; i++)
        addr[i] = (char)('a' + i);

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1)
    {
        perror("error socket");
        exit(1);
    }

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    if (bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr))
== -1)
    {
        perror("error bind");
        exit(1);
    }

    if (listen(listenfd, 1024) == -1)
    {
        perror("error listen");
        exit(1);
    }

    printf("listening on port %d\n", SERV_PORT);

    while (1)
    {
        clilen = sizeof(cliaddr);
        connfd = accept(listenfd, (struct sockaddr *)&cliaddr,
&clilen);

        if (fork() == 0)
        {
            close(listenfd);
            process_client(connfd, addr, semid);
            close(connfd);
            exit(0);
        }
        close(connfd);
    }

    if (semctl(semid, 5, IPC_RMID, NULL) == -1)
    {
        perror("semclt");
        exit(1);
    }
    shmdt(addr);
}
```

Client.c

```
#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "common.h"

void request(int sockfd)
{
    int index;
    char buf[BUF_SIZE];

    while (1)
    {

        buf[0] = 'r';
        if (send(sockfd, &buf, sizeof(buf), 0) == -1)
        {
            perror("error send");
            exit(1);
        }

        if (recv(sockfd, &buf, sizeof(buf), 0) == -1)
        {
            perror("error recieve");
            exit(1);
        }

        printf("Recieved seats: ");
        for (size_t i = 0; i < ARR_SIZE; i++)
            printf("[%c] ", buf[i]);
        puts("");

        index = -1;
        for (int i = 0; i < ARR_SIZE; i++)
        {
            if (buf[i] != ' ')
            {
                index = i;
                break;
            }
        }

        if (index == -1)
        {
            printf("No availible seats left! Disconnecting from
server\n");
            exit(1);
        }

        usleep(500000 + rand() % 500000);

        buf[0] = 'w';
        buf[1] = (char) index;

        printf("Sending index %d to server\n", index);

        if (send(sockfd, &buf, sizeof(buf), 0) == -1)
        {
            perror("error send");
            exit(1);
        }
```

1.3

```
        if (recv(sockfd, &buf, sizeof(buf), 0) == -1)
        {
            perror("error recieve");
            exit(1);
        }

        switch ((int) buf[0])
        {
        case OK:
            puts("Response: OK\n");
            break;
        case ALREADLY_RESERVED:
            puts("Response: already reserved\n");
            break;
        default:
            printf("Disconnected from server\n");
            exit(1);
        }

        usleep(2000000 + rand() % 1500000);
    }
}

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2)
    {
        perror("usage: tcpcli <IPaddress>");
        exit(1);
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if (sockfd == -1)
    {
        perror("error socket");
        exit(1);
    }

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) == -1)
    {
        perror("inet pton");
        exit(1);
    }

    if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))
== -1)
    {
        perror("connection error");
        exit(1);
    }
    else
        printf("Connected\n");

    request(sockfd);
    close(sockfd);

    exit(0);
}
```

Common.h

```
#define SERV_PORT 9877
#define BUF_SIZE 200
#define ARR_SIZE 20

typedef enum {GET_SEATS, RESERVE_SEAT} request_type_t;
typedef enum {OK, ALREADLY_RESERVED, ERROR} response_type_t;

typedef union
{
    struct
    {
        request_type_t action;
        unsigned seat;
    } request;

    struct
    {
        response_type_t status;
        int seats[ARR_SIZE];

    } response;
} req_info_t;
```

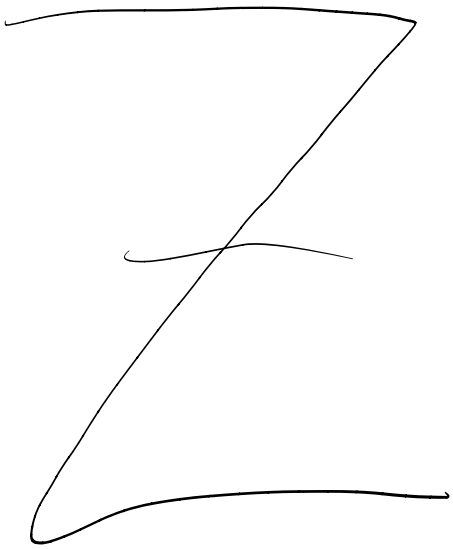
4.1 Дирж

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>
#include <errno.h>
#include <limits.h>

/* Определение типов файлов для передачи в
пользовательскую функцию */
#define FTW_F   1   /* файл, не являющийся каталогом
*/
#define FTW_D   2   /* каталог */
#define FTW_DNR 3   /* каталог, который недоступен
для чтения */
#define FTW_NS  4   /* файл, информацию о котором
невозможно получить с помощью stat */

#define err_quit(msg, ...) { fprintf(stderr, msg,
##__VA_ARGS__); exit(1); }
#define err_sys(msg, ...) { perror(msg); exit(1); }
#define err_ret(msg, ...) { fprintf(stderr, msg,
##__VA_ARGS__); perror(""); }
#define err_dump(msg, ...) { fprintf(stderr, msg,
##__VA_ARGS__); perror(""); abort(); }

/* функция, обрабаывающая каждый файл */
typedef int Myfunc(const char *, const struct stat *,
int, int);

static Myfunc myfunc;
static int dopath(Myfunc *, const char *, int);
static int curr_depth = 0;

/*
 * Спускается по иерархии каталогов, начиная с
указанного каталога.
 * Использует chdir для навигации по каталогам вместо
поддержания полных путей.
 */
static int dopath(Myfunc *func, const char *dirname,
int level) {
    struct stat statbuf;
    struct dirent *dirp;
    DIR *dp;
    int ret = 0;

    /* Получение информации о файле */
    if (lstat(dirname, &statbuf) < 0)
        return func(dirname, &statbuf, FTW_NS,
level);

    /* Если не каталог */
    if (!S_ISDIR(statbuf.st_mode))
        return func(dirname, &statbuf, FTW_F, level);

    /* Это каталог */
    if ((ret = func(dirname, &statbuf, FTW_D,
level)) != 0)
        return ret;

    if (!(dp = opendir(dirname))) {
        return func(dirname, &statbuf, FTW_DNR,
level);
    }

    if (chdir(dirname) < 0) {
        perror("chdir failed");
        return func(dirname, &statbuf, FTW_DNR,
level);
    }

    while (ret == 0 && (dirp = readdir(dp))) {
        /* Пропускаем '.' и '..' */
        if (strcmp(dirp->d_name, ".") != 0 &&
strcmp(dirp->d_name, "..") != 0) {
            /* Получаем информацию о файле */
            if (lstat(dirp->d_name, &statbuf) < 0)
                ret = func(dirp->d_name, &statbuf,
FTW_NS, level + 1);
            else if (S_ISDIR(statbuf.st_mode)) {
                /* Рекурсивно обрабатываем подкаталог
*/
                curr_depth++;

                ret = dopath(func, dirp->d_name,
level + 1);

                curr_depth--;
            } else {
                /* Обычный файл */
                ret = func(dirp->d_name, &statbuf,
FTW_F, level + 1);
            }
        }
    }

    printf("XXX\n");

    /* Закрываем каталог и возвращаемся в
родительский каталог */
    closedir(dp);
    if (chdir("..") < 0)
        err_sys("ошибка вызова chdir(..)");

    return ret;
}
```

5.1 Модули

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/fs_struct.h>
#include <linux/path.h>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ovchinnikov Yaroslav IU7-61B");
void print_task_info(struct task_struct *task)
{
    printk(KERN_INFO "\n\n*** flags: %08x,
__state: %08x, pid - %d, ppid - %d, comm - %s,
policy - %d, migration-disables - %hu, utime -
%llu, stime - %llu,"
                     "prio - %d, static_pro - %d,
normal_prio - %d, rt_priotiry - %u, exit_state -
%d, exit_code - %d, exit_signal - %d, lst-switch-
count - %lu, lst-switch-time - %lu, pwd - %s\n\n",
           task->flags,
           task->__state,
           task->pid,
           task->parent->pid,
           task->comm,
           task->policy,
           task->migration_disabled,
           task->utime,
           task->stime,
           task->prio,
           task->static_prio,
           task->normal_prio,
           task->rt_priority,
           task->exit_state,
           task->exit_code,
           task->exit_signal,
           task->last_switch_count,
           task->last_switch_time,
           task->fs && task->fs->pwd.dentry ?
task->fs->pwd.dentry->d_name.name : (const
unsigned char *)"none"
           );
}

static int __init md_init(void)
{
    int i = 0;
    int kwc = 0;
    struct task_struct *task = &init_task;
    do
    {
        if (!strcmp(task->comm, "kthreadd") || !
strncmp(task->comm, "ksoftirqd/0", 11) || !
strncmp(task->comm, "kworker/0", 9) ||
            !strncmp(task->comm, "migration/0",
11) || (task->prio < 50 && task->prio > 0 && i ==
0))
        {
            if (!strncmp(task->comm, "kworker/0",
9) && i++ > 3)
                continue;
            if (task->prio < 50 && task->prio > 0)
                i++;
            print_task_info(task);
        }

    } while ((task = next_task(task)) !=
&init_task);

    print_task_info(current);

    return 0;
}

static void __exit md_exit(void)
{
    printk(KERN_INFO "***: Goodbye!\n");
}

module_init(md_init);
module_exit(md_exit);
```

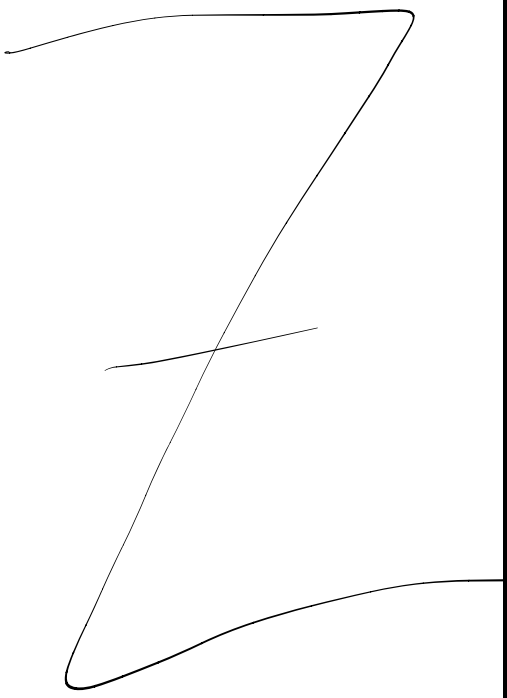
5.2



4.2

```
/*
 * Функция вывода информации о файле.
 * level - уровень вложенности для отображения отступов
 */
static int myfunc(const char *pathname, const struct stat
*statptr, int type, int level) {
    /* Вывод отступов в зависимости от уровня вложенности */
    for (int i = 0; i < level; i++) {
        if (i == level - 1)
            printf("|-- ");
        else
            printf("|   ");
    }

    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:  printf("%s [обычный файл]\n", pathname);
break;
        case S_IFBLK:  printf("%s [блочное устройство]\n",
pathname); break;
        case S_IFCHR:  printf("%s [символьное устройство]\n",
pathname); break;
        case S_IFIFO:  printf("%s [FIFO]\n", pathname); break;
        case S_IFLNK:  printf("%s [символическая ссылка]\n",
pathname); break;
        case S_IFSOCK: printf("%s [сокет]\n", pathname); break;
        default:       printf("%s [неизвестный тип]\n",
pathname); break;
        }
        break;
    case FTW_D:
        printf("%s [каталог]\n", pathname);
        break;
    case FTW_DNR:
        printf("%s [недоступный каталог]\n", pathname);
        break;
    case FTW_NS:
        printf("%s [ошибка stat]\n", pathname);
        break;
    default:
        printf("%s [неизвестный тип %d]\n", pathname, type);
    }

    return 0;
}

int main(int argc, char *argv[]) {
    int ret;

    if (argc != 2)
        err_quit("Использование: %s <начальный_каталог>\n",
argv[0]);

    printf("Дерево каталогов, начиная с %s:\n", argv[1]);

    ret = dopath(myfunc, argv[1], 0);

    exit(ret);
}

static int dopath(Myfunc* func, int level) {
    struct stat statbuf;
    struct dirent *dirp; //информация о записи в каталоге
    DIR *dp; //поток каталога
    int ret;

    if ((dp = opendir(".")) == NULL)
        return -1;

    /* Обрабатываем все файлы в текущем каталоге */
    while ((dirp = readdir(dp)) != NULL) {

        if (strcmp(dirp->d_name, ".") != 0 && strcmp(dirp-
>d_name, "..") != 0) {

            /* Получаем информацию о файле */
            if (lstat(dirp->d_name, &statbuf) < 0) {
               func(dirp->d_name, &statbuf, FTW_NS, level);

            } else {
                if (S_ISDIR(statbuf.st_mode)) {

                    /* Выводим информацию о каталоге */
                    func(dirp->d_name, &statbuf, FTW_D, level);

                    /* Это каталог, переходим в него */
                    if (chdir(dirp->d_name) < 0) {
                        func(dirp->d_name, &statbuf, FTW_DNR,
level);

                    } else {

                        /* Рекурсивно обрабатываем подкаталог */
                        ret = dopath(func, level + 1);

                        /* Возвращаемся в родительский каталог
*/
                        if (chdir("..") < 0)
                            err_sys("ошибка вызова chdir(..)");

                        if (ret != 0) {
                            closedir(dp);
                            return ret;
                        }
                    }
                } else {
                    /* Это не каталог, просто вызываем функцию
*/
                    func(dirp->d_name, &statbuf, FTW_F, level);
                }
            }
        }
    }

    closedir(dp);
    return 0;
}
```

1.4



6.1 Модули 2 (md1, md2, md3)

md.h

```
extern char* md1_data;
extern char* md1_proc(void);

// extern char* md1_local(void);
// extern char *md1_noexport(void);
```

md1.c

```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("");
char* md1_data = "Hello world!";
extern char* md1_proc(void) {
    return md1_data;
}
static char* md1_local(void) {
    return md1_data;
}
extern char* md1_noexport(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);
```

md2.c

```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Yaroslav Ovchinnikov");

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());

    // printk("+ string returned
md1_local() : %s\n", md1_local());
    // printk("+ string returned
md1_noexport() : %s\n", md1_noexport());

    return 0;
}
static void __exit md_exit(void) {
    printk("+ module md2 unloaded!\n");
}
module_init(md_init);
module_exit(md_exit);
```

md3.c

```
#include <linux/init.h>
#include <linux/module.h>
#include "md.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Yaroslav Ovchinnikov");

static int __init md_init(void) {
    printk("+ module md3 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 md3 unloaded!\n");
}
module_init(md_init);
module_exit(md_exit);
```

7.1 Фортунки

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/vmalloc.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/sched/task.h>
#include <linux/pid.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("");

#define BUFFER_SIZE PAGE_SIZE
#define FILENAME "myFile"
#define DIRNAME  "myDir"
#define SYMLINK  "myLink"
#define FILEPATH DIRNAME "/" FILENAME

static struct proc_dir_entry *myFile;
static struct proc_dir_entry *myDir;
static struct proc_dir_entry *myLink;

static char *buffer = NULL;
static int pid_to_show = 0;

static int myOpen(struct inode *spInode, struct file *spFile)
{
    printk(KERN_INFO "myproc: open called\n");
    return 0;
}

static int myRelease(struct inode *spInode, struct file *spFile)
{
    printk(KERN_INFO "myproc: release called\n");
    return 0;
}

static ssize_t myWrite(struct file *file, const char __user *buf,
                       size_t len, loff_t *fPos)
{
    char pid_buf[32];
    int pid;

    printk(KERN_INFO "myproc: write called\n");

    if (len >= sizeof(pid_buf))
        return -EINVAL;

    if (copy_from_user(pid_buf, buf, len) != 0) {
        printk(KERN_ERR "myproc: copy_from_user error\n");
        return -EFAULT;
    }

    pid_buf[len] = '\0';
    if (kstrtoint(pid_buf, 10, &pid) != 0) {
        //printk(KERN_ERR "myproc: Invalid PID format\n");
        return -EINVAL;
    }

    pid_to_show = pid;
    //printk(KERN_INFO "myproc: Set PID to show: %d\n", pid_to_show);

    return len;
}

static ssize_t myRead(struct file *file, char __user *buf,
                      size_t len, loff_t *fPos)
{
    struct task_struct *task = NULL;
    int output_len = 0;

    printk(KERN_INFO "myproc: read called\n");

    if (*fPos > 0)
        return 0;

    if (buffer == NULL)
        return -ENOMEM;

    memset(buffer, 0, BUFFER_SIZE);

    if (pid_to_show <= 0) {
        output_len = snprintf(buffer, BUFFER_SIZE, "No PID specified. Please
write a PID to the file first.\n");
    } else {
        task = get_pid_task(find_get_pid(pid_to_show), PIDTYPE_PID);

        if (task) {
            output_len = snprintf(buffer, BUFFER_SIZE,
                "Process Information for PID %d:\n"
                "  comm:              %s\n"
                "  pid:             %d\n"
                "  ppid:             %d\n"
                "  state:            %08x\n"
                "  flags:            %08x\n"
                "  policy:           %d\n"
                "  prio:             %d\n"
                "  static_prio:      %d\n"
                "  normal_prio:      %d\n"
                "  rt_priority:      %u\n"
                "  utime:            %llu jiffies\n"
                "  stime:            %llu jiffies\n"
                "  migration_disabled:%hu\n"
                "  exit_state:       %d\n"
                "  exit_code:        %d\n"
                "  exit_signal:      %d\n"
                "  last_switch_count: %lu\n"
                "  last_switch_time:  %lu jiffies\n",
                task->pid,
                task->comm,
                task->pid,
                task->parent->pid,
                task->__state,
                task->flags,
                task->policy,
                task->prio,
                task->static_prio,
                task->normal_prio,
                task->rt_priority,
                task->utime,
                task->stime,
                task->migration_disabled,
                task->exit_state,
                task->exit_code,
                task->exit_signal,
                task->last_switch_count,
                task->last_switch_time
            );

            put_task_struct(task);
        } else {
            output_len = snprintf(buffer, BUFFER_SIZE, "Process with PID %d not
found.\n", pid_to_show);
        }
    }
```

8.1 Sequence файлы

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/string.h>
#include <linux/vmalloc.h>
#include <linux/sched.h>
#include <linux/sched/task.h>
#include <linux/pid.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("");

#define DIRNAME "seqDir"
#define FILENAME "seqFile"
#define SYMLINK "seqLink"
#define FILEPATH DIRNAME "/" FILENAME

#define BUFFER_SIZE PAGE_SIZE

static struct proc_dir_entry *proc_file = NULL;
static struct proc_dir_entry *proc_dir = NULL;
static struct proc_dir_entry *proc_link = NULL;

static int pid_to_show = 0;
static struct task_struct *task_to_show = NULL;

ssize_t seq_file_write(struct file *filep, const char __user *buf, size_t len, loff_t *offp);

static int seq_file_show(struct seq_file *m, void *v)
{
    printk(KERN_INFO "seqproc: show called, %p\n", v);

    if (pid_to_show <= 0) {
        seq_printf(m, "No PID specified. Please write a PID to the file first.\n");
        return 0;
    }

    if (!task_to_show) {
        seq_printf(m, "Process with PID %d not found.\n", pid_to_show);
        return 0;
    }

    seq_printf(m, "Process Information for PID %d:\n", pid_to_show);
    seq_printf(m, "  comm:              %s\n", task_to_show->comm);
    seq_printf(m, "  pid:             %d\n", task_to_show->pid);
    seq_printf(m, "  ppid:             %d\n", task_to_show->parent->pid);
    seq_printf(m, "  state:            %08x\n", task_to_show->__state);
    seq_printf(m, "  flags:            %08x\n", task_to_show->flags);
    seq_printf(m, "  policy:           %d\n", task_to_show->policy);
    seq_printf(m, "  prio:             %d\n", task_to_show->prio);
    seq_printf(m, "  static_prio:      %d\n", task_to_show->static_prio);
    seq_printf(m, "  normal_prio:      %d\n", task_to_show->normal_prio);
    seq_printf(m, "  rt_priority:      %u\n", task_to_show->rt_priority);
    seq_printf(m, "  utime:            %llu jiffies\n", task_to_show->utime);
    seq_printf(m, "  stime:            %llu jiffies\n", task_to_show->stime);
    seq_printf(m, "  migration_disabled:%hu\n", task_to_show->migration_disabled);
    seq_printf(m, "  exit_state:       %d\n", task_to_show->exit_state);
    seq_printf(m, "  exit_code:        %d\n", task_to_show->exit_code);
    seq_printf(m, "  exit_signal:      %d\n", task_to_show->exit_signal);
    seq_printf(m, "  last_switch_count: %lu\n", task_to_show->last_switch_count);
    seq_printf(m, "  last_switch_time:  %lu jiffies\n", task_to_show->last_switch_time);

    return 0;
}

static void *seq_file_start(struct seq_file *m, loff_t *pos)
{
    static unsigned long counter = 0;

    printk(KERN_INFO "seqproc: start called\n");

    if (!*pos) {
        /* Обновляем ссылку на task_struct при каждом новом чтении */
        if (task_to_show) {
            put_task_struct(task_to_show);
            task_to_show = NULL;
        }

        if (pid_to_show > 0) {
            task_to_show = get_pid_task(find_get_pid(pid_to_show), PIDTYPE_PID);
        }

        return &counter;
    } else {
        *pos = 0;
        return NULL;
    }
}

static void *seq_file_next(struct seq_file *m, void *v, loff_t *pos)
{
    printk(KERN_INFO "seqproc: next called %p\n", v);
    (*pos)++;
    return NULL;
}
```

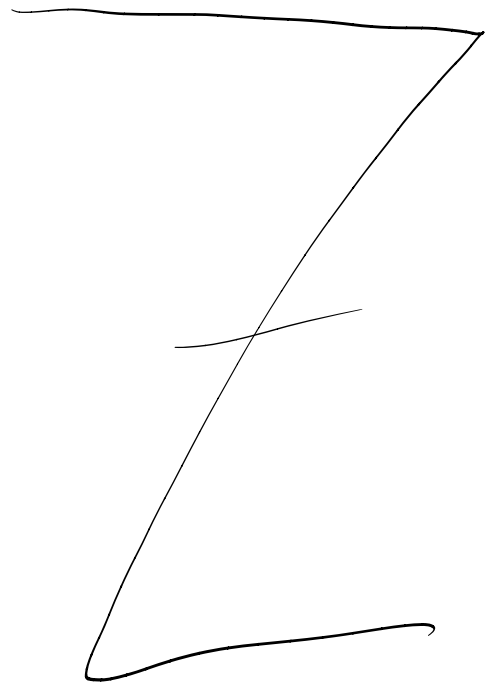
8.2

```
static void seq_file_stop(struct seq_file *m, void *v)
{
    printk(KERN_INFO "seqproc: stop called %p\n", v);
}

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 "seqproc: open called\n");
    return seq_open(f, &seq_file_ops);
}

static struct proc_ops ops = {
    .proc_open = seq_file_open,
    .proc_read = seq_read,
    .proc_write = seq_file_write,
    .proc_release = seq_release,
};

void cleanup_seq_module(void);

static int __init init_seq_module(void)
{
    proc_dir = proc_mkdir(DIRNAME, NULL);
    if (!proc_dir) {
        printk(KERN_INFO "seqproc: Couldn't create proc dir.\n");
        cleanup_seq_module();
        return -ENOMEM;
    }

    proc_file = proc_create(FILENAME, 0666, proc_dir, &ops);
    if (!proc_file) {
        printk(KERN_INFO "seqproc: Couldn't create proc file.\n");
        cleanup_seq_module();
        return -ENOMEM;
    }

    proc_link = proc_symlink(SYMLINK, NULL, FILEPATH);
    if (!proc_link) {
        printk(KERN_INFO "seqproc: Couldn't create proc symlink.\n");
        cleanup_seq_module();
        return -ENOMEM;
    }

    pid_to_show = 0;
    task_to_show = NULL;

    printk(KERN_INFO "seqproc: Module loaded.\n");
    return 0;
}

void cleanup_seq_module(void)
{
    if (proc_file)
        remove_proc_entry(FILENAME, proc_dir);
    if (proc_link)
        remove_proc_entry(SYMLINK, NULL);
    if (proc_dir)
        remove_proc_entry(DIRNAME, NULL);
    if (task_to_show)
        put_task_struct(task_to_show);
}

static void __exit exit_seq_module(void)
{
    cleanup_seq_module();
    printk(KERN_INFO "seqproc: unloaded\n");
}

module_init(init_seq_module);
module_exit(exit_seq_module);

ssize_t seq_file_write(struct file *filep, const char __user *buf, size_t
len, loff_t *offp)
{
    char pid_buf[32];
    int pid;

    printk(KERN_INFO "seqproc: write called\n");

    if (len >= sizeof(pid_buf))
        return -EINVAL;

    if (copy_from_user(pid_buf, buf, len) != 0) {
        printk(KERN_ERR "seqproc: copy_from_user error\n");
        return -EFAULT;
    }

    pid_buf[len] = '\0';
    if (kstrtoint(pid_buf, 10, &pid) != 0) {
        printk(KERN_ERR "seqproc: Invalid PID format\n");
        return -EINVAL;
    }

    pid_to_show = pid;
    printk(KERN_INFO "seqproc: Set PID to show: %d\n",
pid_to_show);

    return len;
}
```

7.2

```
    if (output_len > len)
        output_len = len;

    if (copy_to_user(buf, buffer, output_len) != 0) {
        printk(KERN_ERR "myproc: copy_to_user error\n");
        return -EFAULT;
    }

    *fPos += output_len;
    return output_len;
}

static const struct proc_ops fops = {
    .proc_open = myOpen,
    .proc_read = myRead,
    .proc_write = myWrite,
    .proc_release = myRelease
};

static void freeResources(void)
{
    if (myLink != NULL)
        remove_proc_entry(SYMLINK, NULL);

    if (myFile != NULL)
        remove_proc_entry(FILENAME, myDir);

    if (myDir != NULL)
        remove_proc_entry(DIRNAME, NULL);

    if (buffer != NULL)
        vfree(buffer);
}

static int __init md_init(void)
{
    printk(KERN_INFO "myproc: init\n");

    if ((buffer = vmalloc(BUFFER_SIZE)) == NULL) {
        printk(KERN_ERR "myproc: memory error\n");
        return -ENOMEM;
    }

    memset(buffer, 0, BUFFER_SIZE);

    if ((myDir = proc_mkdir(DIRNAME, NULL)) == NULL) {
        printk(KERN_ERR "myproc: create dir err\n");
        freeResources();
        return -ENOMEM;
    }

    if ((myFile = proc_create(FILENAME, 0666, myDir, &fops)) == NULL) {
        printk(KERN_ERR "myproc: create file err\n");
        freeResources();
        return -ENOMEM;
    }

    if ((myLink = proc_symlink(SYMLINK, NULL, FILEPATH)) == NULL) {
        printk(KERN_ERR "myproc: create link err\n");
        freeResources();
        return -ENOMEM;
    }

    pid_to_show = 0;

    //printk(KERN_INFO "myproc: loaded\n");
    return 0;
}

static void __exit md_exit(void)
{
    printk(KERN_INFO "myproc: exit\n");
    freeResources();
}

module_init(md_init);
module_exit(md_exit);
```

6.2



9.1 Single фаилы

```
#include <linux/kernel.h>
#include <linux/vmalloc.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/sched.h>
#include <linux/sched/task.h>
#include <linux/pid.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Yaroslav Ovchinnikov");

#define BUF_SIZE PAGE_SIZE

#define DIRNAME "singleDir"
#define FILENAME "singleFile"
#define SYMLINK "singleLink"
#define FILEPATH DIRNAME "/" FILENAME

static struct proc_dir_entry *dir;
static struct proc_dir_entry *afile;
static struct proc_dir_entry *link;

static int pid_to_show = 0;
static struct task_struct *task_to_show = NULL;

static int single_file_show(struct seq_file *m, void *v)
{
    printk(KERN_INFO "singleproc: show called, %p\n", v);

    if (pid_to_show <= 0) {
        seq_printf(m, "No PID specified. Please write a PID to
the file first.\n");
        return 0;
    }

    if (!task_to_show) {
        seq_printf(m, "Process with PID %d not found.\n",
pid_to_show);
        return 0;
    }

    seq_printf(m, "Process Information for PID %d:\n",
pid_to_show);
    seq_printf(m, "  comm:                %s\n", task_to_show-
>comm);
    seq_printf(m, "  pid:                 %d\n", task_to_show-
>pid);
    seq_printf(m, "  ppid:                %d\n", task_to_show-
>parent->pid);
    seq_printf(m, "  state:               %08lx\n",
task_to_show->__state);
    seq_printf(m, "  flags:               %08x\n", task_to_show-
>flags);
    seq_printf(m, "  policy:              %d\n", task_to_show-
>policy);
    seq_printf(m, "  prio:                %d\n", task_to_show-
>prio);
    seq_printf(m, "  static_prio:         %d\n", task_to_show-
>static_prio);
    seq_printf(m, "  normal_prio:         %d\n", task_to_show-
>normal_prio);
    seq_printf(m, "  rt_priority:         %u\n", task_to_show-
>rt_priority);
    seq_printf(m, "  utime:               %llu jiffies\n",
task_to_show->utime);
    seq_printf(m, "  stime:               %llu jiffies\n",
task_to_show->stime);
    seq_printf(m, "  migration_disabled:%hu\n", task_to_show-
>migration_disabled);
    seq_printf(m, "  exit_state:          %d\n", task_to_show-
>exit_state);
    seq_printf(m, "  exit_code:           %d\n", task_to_show-
>exit_code);
    seq_printf(m, "  exit_signal:         %d\n", task_to_show-
>exit_signal);
    seq_printf(m, "  last_switch_count: %lu\n", task_to_show-
>last_switch_count);
    seq_printf(m, "  last_switch_time:  %lu jiffies\n",
task_to_show->last_switch_time);

    return 0;
}

ssize_t single_file_write(struct file *file, const char __user
*buf, size_t len, loff_t *offp)
{
    char pid_buf[32];
    int pid;

    printk(KERN_INFO "singleproc: write called\n");

    if (len >= sizeof(pid_buf))
        return -EINVAL;

    if (copy_from_user(pid_buf, buf, len) != 0) {
        printk(KERN_ERR "singleproc: copy_from_user error\n");
        return -EFAULT;
    }

    pid_buf[len] = '\0';
    if (kstrtoint(pid_buf, 10, &pid) != 0) {
        printk(KERN_ERR "singleproc: Invalid PID format\n");
        return -EINVAL;
    }

    /* Обновляем информацию о процессе при записи PID */
    if (task_to_show) {
        put_task_struct(task_to_show);
        task_to_show = NULL;
    }

    pid_to_show = pid;
    task_to_show = get_pid_task(find_get_pid(pid_to_show),
PIDTYPE_PID);

    printk(KERN_INFO "singleproc: Set PID to show: %d\n",
pid_to_show);

    return len;
}

int single_file_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "singleproc: open called\n");
    return single_open(file, single_file_show, NULL);
}

static ssize_t single_file_read(struct file *file, char __user
*buf, size_t size, loff_t *ppos)
{
    printk(KERN_INFO "singleproc: read called\n");
    return seq_read(file, buf, size, ppos);
}

static struct proc_ops fops = {
    .proc_read = single_file_read,
    .proc_write = single_file_write,
    .proc_open = single_file_open,
    .proc_release = single_release
};
```

10.1 VFS

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

#include <asm/atomic.h>
#include <asm/uaccess.h>
#include <linux/slab.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ovchinnikov Yaroslav");

#define MYVFS_MAGIC_NUMBER 0x21212121 //адрес файловой системы
#define SLAB_NAME "MyVFSCache"

static void **cache_mem_area = NULL;
struct kmem_cache * cache = NULL;


#define OBJECTS_PER_PAGE 256
#define TOTAL_PAGES 2
#define TOTAL_OBJECTS (OBJECTS_PER_PAGE * TOTAL_PAGES)


static struct my_vfs_inode
{
    int i_mode;
    unsigned long i_ino;
} my_vfs_inode;

static int size = sizeof(struct my_vfs_inode);

static struct inode *my_vfs_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 = &my_vfs_inode;
        ret->i_ino = 1;
    }

    printk(KERN_INFO "MyVFS: struct inode created\n");

    return ret;
}

static void my_vfs_put_super(struct super_block *sb)
{
    printk(KERN_INFO "MyVFS: super block destroyed!\n");
}

static struct super_operations const my_vfs_sup_ops = {
    .put_super = my_vfs_put_super,
    .statfs = simple_statfs,
    .drop_inode = generic_delete_inode
};

static int my_vfs_fill_sb(struct super_block *sb, void *data,
int silent)
{
    struct inode *root_inode;
    sb->s_blocksize = PAGE_SIZE;
    sb->s_blocksize_bits = PAGE_SHIFT;
    sb->s_magic = MYVFS_MAGIC_NUMBER;
    sb->s_op = &my_vfs_sup_ops;

    root_inode = my_vfs_make_inode(sb, S_IFDIR | 0755);
    if (!root_inode)
    {
        printk(KERN_ERR "MyVFS: my_vfs_make_inode error\n");
        return -ENOMEM;
    }

    root_inode->i_atime = root_inode->i_mtime = root_inode-
>i_ctime = current_time(root_inode);
    root_inode->i_op = &simple_dir_inode_operations;
    root_inode->i_fop = &simple_dir_operations;

    sb->s_root = d_make_root(root_inode);
    if (!sb->s_root)
    {
        printk(KERN_ERR "MyVFS: d_make_root error\n");
        iput(root_inode);
        return -ENOMEM;
    }
    else
        printk(KERN_INFO "MyVFS: VFS root created\n");

    printk(KERN_INFO "MyVFS: super_block init");

    return 0;
}

static struct dentry *
my_vfs_mount(struct file_system_type *type, int flags,
             const char *dev, void *data)
{
    struct dentry *const root = mount_nodev(type, flags, data,
my_vfs_fill_sb);

    if (IS_ERR(root))
        printk(KERN_ERR "MyVFS: mounting failed\n");
    else
        printk(KERN_ERR "MyVFS: mounted\n");

    return root;
}

static void func_init(void *p)
{
    *(int *)p = (int)p;
}

void my_kill_litter_super(struct super_block *sb)
{
    printk(KERN_INFO "MyVFS: my_kill_litter_super");

    return kill_litter_super(sb);
}

static struct file_system_type my_vfs_type = {
    .owner   = THIS_MODULE,
    .name    = "MyVFS",
    .mount   = my_vfs_mount,
    .kill_sb = my_kill_litter_super
};
```

11.1 Tasklet

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <asm/atomic.h>
#include <linux/sched.h>
#include <linux/fs_struct.h>
#include <linux/seq_file.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/time.h>
#include <asm/io.h>
#include <linux/proc_fs.h>
#include <linux/jiffies.h>

MODULE_LICENSE("GPL");

#define IRQ_NO 1
#define BUF_SIZE 128

static const char *lowercase[] = {
    "None", "Esc", "1", "2", "3", "4", "5", "6", "7", "8",
"9", "0", "-", "=",
    "Backspace", "Tab", "q", "w", "e", "r", "t", "y", "u",
"i", "o", "p", "[", "]",
    "Enter", "Left Ctrl", "a", "s", "d", "f", "g", "h", "j",
"k", "l", ";", "'", "`",
    "Left Shift", "\\", "z", "x", "c", "v", "b", "n", "m",
",", ".", "/",
    "Right Shift", "Keypad *", "Left Alt", "Space", "Caps
Lock",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9",
"F10"};

static const char *uppercase[] = {
    "None", "Esc", "!", "@", "#", "$", "%", "^", "&", "*",
"(", ")", "_", "+",
    "Backspace", "Tab", "Q", "W", "E", "R", "T", "Y", "U",
"I", "O", "P", "{", "}",
    "Enter", "Left Ctrl", "A", "S", "D", "F", "G", "H", "J",
"K", "L", ":", "\"", "~",
    "Left Shift", "|", "Z", "X", "C", "V", "B", "N", "M", "<",
">", "?",
    "Right Shift", "Keypad *", "Left Alt", "Space", "Caps
Lock",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9",
"F10"};

static struct tasklet_struct *tasklet = NULL;
static char buffer[BUF_SIZE];
static ktime_t start_time;
static u64 total_time = 0;

static int left_shift_pressed = 0;
static int right_shift_pressed = 0;
static int caps_lock_active = 0;

static int my_proc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "Last pressed key: %s\n", buffer);
    seq_printf(m, "Time: %llu ns\n", total_time);

    return 0;
}

static int my_proc_open(struct inode *inode, struct file
*file)
{
    return single_open(file, my_proc_show, NULL);
}

static const struct proc_ops proc_fops = {
    .proc_open = my_proc_open,
    .proc_read = seq_read,
    .proc_release = single_release,
};

static int should_use_uppercase(void)
{
    return (left_shift_pressed || right_shift_pressed) ^
caps_lock_active;
}

void my_tasklet_fun(unsigned long data)
{
    int code = inb(0x60);
    char *key_event;
    const char **current_layout = should_use_uppercase() ?
uppercase : lowercase;

    switch (code)
    {
    case 0x2A:
        left_shift_pressed = 1;
        return;
    case 0xAA:
        left_shift_pressed = 0;
        return;
    case 0x36:
        right_shift_pressed = 1;
        return;
    case 0xB6:
        right_shift_pressed = 0;
        return;
    }

    if ((code >= 0x47 && code <= 0x53) || code == 0x1C)
    {
        total_time = ktime_to_ns(ktime_sub(ktime_get(),
start_time));
    }
    else if (code & 0x80)
    {
        total_time = ktime_to_ns(ktime_sub(ktime_get(),
start_time));
    }
    else
    {
        key_event = "pressed";
        code &= 0x7F;
        if (code >= 0 && code < sizeof(lowercase) /
sizeof(lowercase[0]))
        {
            printk(KERN_INFO "Tasklet : Key %s: %s
(code=0x%02x), Time=%lld ns\n",
                   key_event, current_layout[code], code,
ktime_to_ns(ktime_sub(ktime_get(), start_time)));
            snprintf(buffer, sizeof(buffer), "%s
(code=0x%02x)", current_layout[code], code);
        }

        total_time = ktime_to_ns(ktime_sub(ktime_get(),
start_time));
    }
}
```

11.2

```
static irqreturn_t interrupt_handler(int irq, void *dev_id)
{
    if (irq == IRQ_NO)
    {
        start_time = ktime_get();
        tasklet_schedule(tasklet);
        return IRQ_HANDLED;
    }
    return IRQ_NONE;
}
static int __init my_init(void)
{
    if (proc_create("keyboard", 0, NULL, &proc_fops) == NULL)
    {
        return -ENOMEM;
    }
    int ret = request_irq(IRQ_NO, interrupt_handler, IRQF_SHARED,
"interrupt_handler_tasklet", (void *)(interrupt_handler));
    if (ret)
    {

        return ret;
    }

    buffer[0] = '\0';
    tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
    if (!tasklet)
    {


        free_irq(IRQ_NO, (void *)(interrupt_handler));
        return -ENOMEM;
    }
    tasklet_init(tasklet, my_tasklet_fun, 0);

    return 0;
}

static void __exit my_exit(void)
{
    remove_proc_entry("keyboard", NULL);
    tasklet_kill(tasklet);
    kfree(tasklet);
    free_irq(IRQ_NO, (void *)(interrupt_handler));
}

module_init(my_init);
module_exit(my_exit);
```

10.2

```
static int __init my_vfs_init(void)
{
    int ret = register_filesystem(&my_vfs_type);
    int i;

    if (ret != 0)
    {
        printk(KERN_ERR "MyVFS: register_filesystem error\n");
        return ret;
    }

    cache_mem_area = kmalloc(sizeof(void *) *
OBJECTS_PER_PAGE, GFP_KERNEL);
    if (!cache_mem_area)
    {
        printk(KERN_ERR "MyVFS: cache_mem_area allocation
error\n");
        unregister_filesystem(&my_vfs_type);
        return -ENOMEM;
    }

    cache = kmem_cache_create(SLAB_NAME, sizeof(void *), 0,
                              SLAB_HWCACHE_ALIGN, func_init);
    if (!cache)
    {
        printk(KERN_ERR "MyVFS: kmem_cache_create error\n");
        kfree(cache_mem_area);
        unregister_filesystem(&my_vfs_type);
        return -ENOMEM;
    }

    printk(KERN_INFO "MyVFS: Slab created, allocating %d
objects (%d pages)\n",
           OBJECTS_PER_PAGE, TOTAL_PAGES);

    for (i = 0; i < TOTAL_OBJECTS; i++)
    {
        cache_mem_area[i] = kmem_cache_alloc(cache,
GFP_KERNEL);
        if (!cache_mem_area[i])
        {
            printk(KERN_ERR "MyVFS: kmem_cache_alloc failed at
object %d\n", i);

            while (--i >= 0)
            {
                kmem_cache_free(cache, cache_mem_area[i]);
            }

            kmem_cache_destroy(cache);
            kfree(cache_mem_area);
            unregister_filesystem(&my_vfs_type);
            return -ENOMEM;
        }

        *(int *)cache_mem_area[i] = i * 10;
    }

    printk(KERN_INFO "MyVFS: Successfully allocated %d
objects\n", TOTAL_OBJECTS);
    printk(KERN_INFO "MyVFS: loaded\n");

    return 0;
}


static void __exit my_vfs_exit(void)
{
    int i;

    if (cache_mem_area)
    {
        for (i = 0; i < TOTAL_OBJECTS; i++)
        {
            if (cache_mem_area[i])
            {
                kmem_cache_free(cache, cache_mem_area[i]);
            }
        }
        kfree(cache_mem_area);
    }

    if (cache)
    {
        kmem_cache_destroy(cache);
    }

    if (unregister_filesystem(&my_vfs_type) != 0)
        printk(KERN_ERR "MyVFS: unregister_filesystem
error\n");

    printk(KERN_INFO "MyVFS: exit\n");
}


module_init(my_vfs_init);
module_exit(my_vfs_exit);
```

9.2

```
static void freemem(void)
{
    if (link)
        remove_proc_entry(SYMLINK, NULL);
    if (afile)
        remove_proc_entry(FILENAME, dir);
    if (dir)
        remove_proc_entry(DIRNAME, NULL);
    if (task_to_show)
        put_task_struct(task_to_show);
}



static int __init mod_init(void)
{
    dir = proc_mkdir(DIRNAME, NULL);
    if (!dir) {
        printk(KERN_ERR "singleproc: mkdir failed!\n");
        freemem();
        return -ENOMEM;
    }

    afile = proc_create(FILENAME, 0666, dir, &fops);
    if (!afile) {
        printk(KERN_ERR "singleproc: file create failed!
\n");
        freemem();
        return -ENOMEM;
    }

    link = proc_symlink(SYMLINK, NULL, FILEPATH);
    if (!link) {
        printk(KERN_ERR "singleproc: failed to create
symlink!\n");
        freemem();
        return -ENOMEM;
    }

    pid_to_show = 0;
    task_to_show = NULL;

    printk(KERN_INFO "singleproc: module loaded!\n");
    return 0;
}

static void __exit mod_exit(void)
{
    freemem();
    printk(KERN_INFO "singleproc: module unloaded!\n");
}

module_init(mod_init);
module_exit(mod_exit);
```

12.1 Workqueue

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <asm/atomic.h>
#include <linux/sched.h>
#include <linux/fs_struct.h>
#include <linux/seq_file.h>
#include <linux/vmalloc.h>
#include <linux/proc_fs.h>
#include <linux/version.h>
#include <linux/time.h>
#include <asm/io.h>
#include <linux/workqueue.h>
#include <linux/delay.h>

MODULE_LICENSE("GPL");

#define IRQ_NUM 1
#define DIR_NAME "key_buf"

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

typedef struct
{
    struct work_struct work;
    ktime_t start_time;
    int code;
} key_work_t;

static struct workqueue_struct *key_wq;
static key_work_t *work1, *work2;

static int left_shift_pressed = 0;
static int right_shift_pressed = 0;
static int caps_lock_active = 0;

static char *key_names_lowercase[84] = {
    " ", "Esc", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0", "-",
"=", "Backspace",
    "Tab", "q", "w", "e", "r", "t", "y", "u", "i", "o", "p", "[", "]",
"Enter", "Ctrl",
    "a", "s", "d", "f", "g", "h", "j", "k", "l", ";", "'", "`", "Shift
(left)", "\\",
    "z", "x", "c", "v", "b", "n", "m", ",", ".", "/", "Shift (right)",
    "*", "Alt", "Space", "CapsLock",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10",
    "NumLock", "ScrollLock", "Home", "Up", "Page-Up", "-", "Left",
    " ", "Right", "+", "End", "Down", "Page-Down", "Insert", "Delete"};

static char *key_names_uppercase[84] = {
    " ", "Esc", "!", "@", "#", "$", "%", "^", "&", "*", "(", ")", "_",
"+", "Backspace",
    "Tab", "Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P", "{", "}",
"Enter", "Ctrl",
    "A", "S", "D", "F", "G", "H", "J", "K", "L", ":", "\"", "~", "Shift
(left)", "|",
    "Z", "X", "C", "V", "B", "N", "M", "<", ">", "?", "Shift (right)",
    "*", "Alt", "Space", "CapsLock",
    "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10",
    "NumLock", "ScrollLock", "Home", "Up", "Page-Up", "-", "Left",
    " ", "Right", "+", "End", "Down", "Page-Down", "Insert", "Delete"};

static struct proc_dir_entry *proc_file;

static int key_code = -1;
static char *key_name;

static int should_use_uppercase(void)
{
    return (left_shift_pressed || right_shift_pressed) ^
caps_lock_active;
}

static int my_show(struct seq_file *m, void *v)
{
    if (key_code != -1)
        seq_printf(m, "Key: %s Code: %d\n", key_name, key_code);
    return 0;
}

static int my_open(struct inode *inode, struct file *file)
{
    return single_open(file, my_show, NULL);
}

static struct proc_ops key_fops = {
    .proc_read = seq_read,
    .proc_open = my_open,
    .proc_release = single_release};

static void work_fun2(struct work_struct *work)
{

    key_work_t *key_work = (key_work_t *)work;
    int code = key_work->code;
    int is_release = (code & 0x80) ? 1 : 0;
    int press_code = code & 0x7F;

    ktime_t end_time;
    s64 diff_ns;

    switch (code)
    {
    case 0x2A: // Left Shift press
        left_shift_pressed = 1;
        return;
    case 0xAA: // Left Shift release
        left_shift_pressed = 0;
        return;
    case 0x36: // Right Shift press
        right_shift_pressed = 1;
        return;
    case 0xB6: // Right Shift release
        right_shift_pressed = 0;
        return;
    case 0x3A: // Caps Lock press
        caps_lock_active = !caps_lock_active;
        return;
    }

    if (!is_release)
    {
        return;
    }

      if (press_code < 84 && press_code != 28)
    {
        printk(KERN_INFO "work 1 : begin");

        key_code = press_code;
        end_time = ktime_get();
        diff_ns = ktime_to_ns(ktime_sub(end_time, key_work-
>start_time));

        if (should_use_uppercase())
        {
            key_name = key_names_uppercase[press_code];
        }
        else
        {
            key_name = key_names_lowercase[press_code];
        }

        printk(KERN_INFO "work 1 : Key: %s Code: %d Time: %lld\n",
key_name, key_code, diff_ns);
    }
    printk(KERN_INFO "work 1 : end");
}
```

13.1 Buf-unbuf

```
#include <stdio.h>
#include <fcntl.h>
int main()
{
    // have kernel open connection to file alphabet.txt
    int fd = open("alphabet.txt", O_RDONLY);

    // create two a C I/O buffered streams using the above
connection
    FILE* fs1 = fdopen(fd, "r");
    char buff1[20];
    setvbuf(fs1, buff1, _IOFBF, 20);

    FILE* fs2 = fdopen(fd, "r");
    char buff2[20];
    setvbuf(fs2, buff2, _IOFBF, 20);

    // read a char & write it alternatingly from fs1 and fs2
    int flag1 = 1, flag2 = 2;
    while (flag1 == 1 || flag2 == 1)
    {
        char c;
        flag1 = fscanf(fs1, "%c", &c);
        if (flag1 == 1) {
            fprintf(stdout, "%c", c);
        }
        flag2 = fscanf(fs2, "%c", &c);
        if (flag2 == 1) {
            fprintf(stdout, "%c", c);
        }
    }

    return 0;
}
```

Aubvcwdxeyfzghijklmnopqrst$

```
#include <fcntl.h>
int main()
{
    char c;
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    int fl = 1;
    while (fl)
    {
        if (read(fd1, &c, 1) != 1) fl = 0;
        else
        {
            write(1, &c, 1);
            if (read(fd2, &c, 1) != 1) fl = 0;
            else write(1, &c, 1);
        }
    }
    return 0;
}
```

// AAbbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz$

```
#include <fcntl.h>
#include <pthread.h>
void* other_thread(void* data)
{
    int fl = 1;
    char c;
    int fd = (int)data;
    char message[] = "2:_\t";
    while (fl)
    {
        if (read(fd, message + 2, 1) != 1) fl = 0;
        else write(1, message, sizeof(message));
    }
}
int main()
{
    char c;
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    pthread_t thread;
    pthread_attr_t attrs;
    pthread_attr_setdetachstate(&attrs,
PTHREAD_CREATE_DETACHED);
    if (pthread_create(&thread, &attrs, other_thread,
(void*)fd2) == -1)
    {
        printf("pthread");
        exit(-1);
    }
    int fl = 1;
    char message[] = "1:_\t";
    while (fl)
    {
        if (read(fd1, message + 2, 1) != 1) fl = 0;
        else write(1, message, sizeof(message));
    }
    return 0;
}
```

// не успел дочитать нужен Джоин 1:A 1:b 1:c 1:d 1:e 1:f 1:g 1:h 1:i 1:j 1:k 1:l 1:m 1:n 1:o 1:p 1:q 1:r 1:s 1:t 1:u 1:v 1:w 1:x 1:y 1:z $

```
#include <fcntl.h>
#include <pthread.h>

void* other_thread(void* data)
{
    int fl = 1;
    char c;
    int fd = (int)data;
    char message[] = "2:_\t";
    while (fl)
    {
        if (read(fd, message + 2, 1) != 1) fl = 0;
        else write(1, message, sizeof(message));
    }
}

int main()
{
    char c;
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    pthread_t thread;
    if (pthread_create(&thread, NULL, other_thread,
(void*)fd2) == -1)
    {
        printf("pthread");
        exit(-1);
    }
    int fl = 1;
    char message[] = "1:_\t";
    while (fl)
    {
        if (read(fd1, message + 2, 1) != 1) fl = 0;
        else write(1, message, sizeof(message));
    }
    pthread_join(thread, NULL);
    return 0;
}
```

```
1:A 1:b 1:c 1:d 1:e 1:f 1:g 1:h 1:i 1:j 1:k 1:l 1:m 1:n 1:o
1:p 1:q 1:r 1:s 1:t 1:u 1:v 1:w 1:x 1:y 1:z 2:A 2:b 2:c 2:d
2:e 2:f 2:g 2:h 2:i 2:j 2:k 2:l 2:m 2:n 2:o 2:p 2:q 2:r 2:s
2:t 2:u 2:v 2:w 2:x 2:y 2:z
```

13.2

```
#include <fcntl.h>
#include <pthread.h>
void* other_thread(void* data)
{
    int fl = 1;
    int fd = (int)data;
    char message[] = "_:_\t";
    message[0] = '0' + fd - 2;
    while (fl)
    {
        if (read(fd, message + 2, 1) != 1) fl = 0;
        else write(1, message,sizeof(message));
    }
}
int main()
{
    char c;
    int fd1 = open("alphabet.txt", O_RDONLY);
    int fd2 = open("alphabet.txt", O_RDONLY);
    pthread_t thread_1, thread_2;
if (pthread_create(&thread_1, NULL, other_thread, (void*)fd1)
== -1)
    {
        printf("pthread");
        exit(-1);
    }
    if (pthread_create(&thread_2, NULL, other_thread,
(void*)fd2) == -1)
    {
        printf("pthread");
        exit(-1);
    }
    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);
    return 0;
}
```

```
1:A  2:A  1:b  2:b  1:c  2:c  1:d  2:d  1:e  2:e  1:f  2:f
1:g  2:g  1:h  2:h 1:i  2:i  1:j  2:j  1:k  2:k  1:l  2:l  1:m
2:m  1:n  2:n  1:o  1:p  1:q  2:o 1:r  2:p  1:s  2:q  1:t  2:r
1:u  2:s  1:v  2:t  1:w  2:u  1:x  2:v  1:y  2:w 1:z  2:x  2:y
2:z $
```

```
#include <stdio.h>
#include <sys/stat.h>
int main()
{
  struct stat statbuf;
  FILE* fs1 = fopen("out.txt", "w");
  stat("out.txt", &statbuf);
  int fd1 = fileno(fs1);
  printf("fopen fd1=%d; inode=%ld size=%ld\n", fd1,
statbuf.st_ino, statbuf.st_size);
  FILE* fs2 = fopen("out.txt", "w");
  int fd2 = fileno(fs2);
  printf("fopen fd2=%d; inode=%ld size=%ld\n", fd2,
statbuf.st_ino, statbuf.st_size);
  for (char c = 'a'; c <= 'z';)
  {
    fprintf(fs1, "%c", c++);
    stat("out.txt", &statbuf);
    printf("stat inode=%ld size=%ld\n", statbuf.st_ino,
statbuf.st_size);
    fprintf(fs2, "%c", c++);
    stat("out.txt", &statbuf);
    printf("stat inode=%ld size=%ld\n", statbuf.st_ino,
statbuf.st_size);
  }
  fclose(fs1);
  stat("out.txt", &statbuf);
  printf("fclose fd1=%d; inode=%ld size=%ld\n", fd1,
statbuf.st_ino, statbuf.st_size);
  fclose(fs2);
  stat("out.txt", &statbuf);
  printf("fclose fd2=%d; inode=%ld size=%ld\n", fd2,
statbuf.st_ino, statbuf.st_size);
  return 0;
}
```

в конце size становится дважды 13

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
  struct stat statbuf;
  int fd1 = open("out.txt", O_WRONLY | O_APPEND);
  int fd2 = open("out.txt", O_WRONLY | O_APPEND);
  stat("out.txt", &statbuf);
  printf("open fd1=%d fd2=%d inode=%ld size=%ld\n", fd1, fd2,
statbuf.st_ino, statbuf.st_size);
  for (char c = 'a'; c <= 'z';)
  {
    write(fd1, &c, 1);
    stat("out.txt", &statbuf);
    printf("stat inode=%ld size=%ld\n", statbuf.st_ino,
statbuf.st_size);
    c++;
    write(fd2, &c, 1);
    stat("out.txt", &statbuf);
    printf("stat inode=%ld size=%ld\n", statbuf.st_ino,
statbuf.st_size);
    c++;
  }
  close(fd1);
  close(fd2);
  stat("out.txt", &statbuf);
  printf("close fd1=%d fd2=%d inode=%ld size=%ld\n", fd1,fd2,
statbuf.st_ino, statbuf.st_size);
  return 0;
}
```

размер увелич по 1, итого 26

12.2

```
static void work_fun1(struct work_struct *work)
{
    key_work_t *key_work = (key_work_t *)work;
    int code = key_work->code;
    int is_release = (code & 0x80) ? 1 : 0;

    switch (code)
    {
    case 0x2A: // Left Shift press
        left_shift_pressed = 1;
        return;
    case 0xAA: // Left Shift release
        left_shift_pressed = 0;
        return;
    case 0x36: // Right Shift press
        right_shift_pressed = 1;
        return;
    case 0xB6: // Right Shift release
        right_shift_pressed = 0;
        return;
    case 0x3A: // Caps Lock press
        caps_lock_active = !caps_lock_active;
        return;
    }

    if (!is_release)
    {
        return;
    }

    printk(KERN_INFO "work 2 : sleep start");
    msleep(10);
    printk(KERN_INFO "work 2 : sleep end");
    printk(KERN_INFO "work 2 : begin");

    int press_code = code & 0x7F;
    ktime_t end_time;
    s64 diff_ns;

    if (press_code < 84 && press_code != 28)
    {
        key_code = press_code;
        end_time = ktime_get();
        diff_ns = ktime_to_ns(ktime_sub(end_time, key_work->start_time));

        if (should_use_uppercase())
        {
            key_name = key_names_uppercase[press_code];
        }
        else
        {
            key_name = key_names_lowercase[press_code];
        }

        printk(KERN_INFO "work 2 : Key: %s Code: %d Time: %lld\n", key_name, key_code,
diff_ns);
    }
    printk(KERN_INFO "work 2 : end");
}

static irqreturn_t interrupt_handler(int irq, void *dev_id)
{
    if (irq == IRQ_NUM)
    {
        int code = inb(0x60);

        work1->start_time = ktime_get();
        work1->code = code;
        work2->code = code;

        queue_work(key_wq, (struct work_struct *)work1);
        queue_work(key_wq, (struct work_struct *)work2);

        return IRQ_HANDLED;
    }
    return IRQ_NONE;
}

static int __init my_init(void)
{
    int ret = request_irq(IRQ_NUM, interrupt_handler, IRQF_SHARED, "interrupt_handler_wq",
(void *)(interrupt_handler));
    if (ret)
    {
        printk(KERN_ERR "Error: request_irq: %d\n", ret);
        return ret;
    }

    key_wq = alloc_workqueue("my_workqueue", __WQ_LEGACY | WQ_MEM_RECLAIM, 1);
    if (!key_wq)
    {
        free_irq(IRQ_NUM, (void *)(interrupt_handler));
        printk(KERN_ERR "Error: Failed to create workqueue\n");
        return -ENOMEM;
    }

    work1 = (key_work_t *)kmalloc(sizeof(key_work_t), GFP_KERNEL);
    if (!work1)
    {
        free_irq(IRQ_NUM, (void *)(interrupt_handler));
        destroy_workqueue(key_wq);
        printk(KERN_ERR "Error: Failed to allocate immediate work\n");
        return -ENOMEM;
    }
    INIT_WORK((struct work_struct *)work1, work_fun2);

    work2 = (key_work_t *)kmalloc(sizeof(key_work_t), GFP_KERNEL);
    if (!work2)
    {
        free_irq(IRQ_NUM, (void *)(interrupt_handler));
        destroy_workqueue(key_wq);
        kfree(work1);
        printk(KERN_ERR "ERROR: Failed to allocate delayed work\n");
        return -ENOMEM;
    }
    INIT_WORK((struct work_struct *)work2, work_fun1);

    proc_file = proc_create("key_buf_wq", 0444, NULL, &key_fops);
    if (!proc_file)
    {
        printk(KERN_ERR "ERROR: Failed to create proc file\n");
        free_irq(IRQ_NUM, (void *)(interrupt_handler));
        destroy_workqueue(key_wq);
        kfree(work1);
        kfree(work2);
        return -ENOMEM;
    }

    printk(KERN_INFO "Keyboard module loaded\n");
    return 0;
}

static void __exit my_exit(void)
{
    proc_remove(proc_file);

    flush_workqueue(key_wq);
    destroy_workqueue(key_wq);

    free_irq(IRQ_NUM, (void *)(interrupt_handler));

    kfree(work1);
    kfree(work2);

    printk(KERN_INFO "INFO: Keyboard module unloaded\n");
}

module_init(my_init);
module_exit(my_exit);
```