МИНИСТЕРСТВО науки и высшего ОБРАЗОВАНИЯ РОссИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

Факультет №3 «Системы управления, информатика и электроэнергетика»

Кафедра № 304 «Вычислительные машины, системы и сети»

Проектирование драйверов

Отчет по лабораторной работе № 2

«Написание символьного драйвера»

Выполнил студент группы 3О\_425Бк Наштыков Д.Е.

Проверил доцент 304 каф., к.т.н. Титов Ю.П.

Москва 2022 г.

**Задание**

Реализовать символьный драйвер, имеющий следующий функционал:

* Создает виртуальное устройство
* Каждый пользователь может читать с устройства и писать на него
* Для каждого пользователя создается отдельный циклический буфер, в который пользователь может писать и с которого может читать
* В случае, если в циклическом буфере нет данных, читающий процесс должен быть заблокирован до появления любого кол-ва данных

**Выполнение**

Проектирование:

* Номер устройства выделяется динамически системой
* При переполнении циклического буфера происходит перезапись уже записанных данных
* Буферы хранятся в виде циклического списка
* Для каждого пользователя выделяется структура данных pipe, которая содержит:
  + cbuff –массив символов длиной USHRT\_MAX, циклический буфер
  + head – беззнаковое целое, индекс для начала чтения
  + tail – беззнаковое целое, индекс для начала записи
  + bitesAwaible – беззнаковое целое, кол-во доступных для чтения байт
  + mutex – мьютекс
  + numOfDescriptors – целое, кол-во файловых дескрипторов, указывающих на данный пайп
  + user\_id – uid\_t, id пользователя-владельца буфера
  + queue – wait\_queue\_head\_t, очередь для блокировки читающих процессов
  + list – list\_head, структура данных для хранения буферов в циклическом списке

Реализация:

Полный код лабораторной работы может быть найден по адресу: https://github.com/SHTYKOVYCH/diploma-practice/tree/main/lab2

Инициализация устройства:

static int \_\_init lab2\_init(void) {

// allocated number

int error = alloc\_chrdev\_region(&device, 0, 1, "dmitrii\_pipe");

if (error < 0) {

pr\_info(MSG\_PREFIX "could't register device %d\n", error);

return error;

}

// initing the device

cdev\_init(&dev, &fops);

int err = cdev\_add(&dev, device, 1);

if (err) {

pr\_info(MSG\_PREFIX "coulnd't init cdev\n");

return err;

}

// allocating memory for list mutex

if (!tryAllocateMemory((void\*\*)&pipesMutex, sizeof(struct mutex))) {

unregister\_chrdev\_region(device, 1);

return -1;

}

// initing mutex

mutex\_init(pipesMutex);

// initing list mutex

pipes = NULL;

pr\_info(MSG\_PREFIX "dmitrii\_pipe loaded\n");

return 0;

}

Деинициализация устройства:

static void \_\_exit lab2\_exit(void) {

mutex\_lock(pipesMutex);

kfree(pipesMutex);

if (pipes) {

if (pipes->list.next != &pipes->list) {

struct pipe\* needle;

struct pipe\* n;

for (needle = list\_entry(pipes->list.next, struct pipe, list); needle != &pipes->list;) {

n = list\_entry(needle->list.next, struct pipe, list);

pipeDelete(needle);

needle = n;

}

}

pipeDelete(pipes);

}

cdev\_del(&dev);

unregister\_chrdev\_region(device, 1);

pr\_info(MSG\_PREFIX "dmitrii\_pipe unloaded\n");

}

Регистрация нового буфера устройства:

int device\_open(struct inode\* iocl, struct file\* f) {

// pointer for choosed pipe

struct pipe\* pipeToRemember;

// locking list mutex

mutex\_lock(pipesMutex);

// if there is no pipes

if (!pipes) {

// trying to init new one

pr\_info(MSG\_PREFIX "allocating memory for list\n");

pipes = pipeInit();

if (!pipes) {

// if faild then releasing mutex and leaving with error

mutex\_unlock(pipesMutex);

return -1;

}

// if inited then the pipe for user is the head of pipes, so remembering it

pipeToRemember = pipes;

} else {

// if there are some pipes

struct pipe\* t\_pipe = tryFindPipeWithUserId(pipes, current\_uid().val);

// trying to find pipe with the same user id

if (!t\_pipe) {

// if there is no such pipe then creating new one

pr\_info(MSG\_PREFIX "allocating memory for new pipe\n");

if (!(t\_pipe = pipeInit())) {

mutex\_unlock(pipesMutex);

return -1;

}

// adding new pipe to the list

list\_add(&t\_pipe->list, &pipes->list);

}

// Pipe for user is either a founded one or the new one

// in both cases remembering it and setting num of file's descriptors to 1

pipeToRemember = t\_pipe;

}

mutex\_lock(pipeToRemember->mutex);

pipeToRemember->numOfDescriptors += 1;

mutex\_unlock(pipeToRemember->mutex);

// as we don't do anything with list, we can release it

mutex\_unlock(pipesMutex);

// putting pipe pointer so we don't need to search it every time

f->private\_data = pipeToRemember;

pr\_info(MSG\_PREFIX "pipe opened for %d\n", pipeToRemember->userId);

return 0;

}

Удаление буфера устройства:

int device\_close(struct inode\* iocl, struct file\* f) {

// getting pointer to the user's pipe

struct pipe\* currentPipe = f->private\_data;

// locking list as we gonna modify it

mutex\_lock(pipesMutex);

// also blocking pipe as we gonna work with it

mutex\_lock(currentPipe->mutex);

// decreasing num of descriptors

currentPipe->numOfDescriptors -= 1;

// if there is no other users of this pipe - delete it

if (currentPipe->numOfDescriptors == 0) {

pr\_info(MSG\_PREFIX "deleting pipe\n");

// if we deleting head of the list then choosing

if (pipes == currentPipe) {

// if the list contains only one element - deinit it

if (pipes->list.next == &pipes->list) {

pr\_info(MSG\_PREFIX "now there is no list\n");

pipes = NULL;

} else {

// else just move head one element further and removing element from it

pipes = list\_entry(pipes->list.next, struct pipe, list);

pr\_info(MSG\_PREFIX "now the head is pipe for user %d\n", pipes->userId);

list\_del(&currentPipe->list);

}

} else {

list\_del(&currentPipe->list);

}

pr\_info(MSG\_PREFIX "deleted pipe for %d\n", currentPipe->userId);

// then just deleting pipe

pipeDelete(currentPipe);

mutex\_unlock(pipesMutex);

return 0;

}

// unlocking list and pipe

mutex\_unlock(currentPipe->mutex);

mutex\_unlock(pipesMutex);

pr\_info(MSG\_PREFIX "closed pipe for %d\n", currentPipe->userId);

return 0;

}

Запись в буфер устройства:

static ssize\_t device\_write(struct file\* f, const char \_\_user\* buff, size\_t numOfBytes, loff\_t\* offset) {

// getting user's pipe

struct pipe\* writePipe = f->private\_data;

// locking it

mutex\_lock(writePipe->mutex);

// we gonna write data by chunks

char t\_buff[RW\_CHUNK\_SIZE] = {0};

// smarty way to divide by 2^n value

unsigned int numOfIteations = numOfBytes >> RW\_SHIFT\_CONSTANT;

// writing to pipe by chunks

unsigned int i = 0;

while(i < numOfIteations) {

copy\_from\_user(t\_buff, buff + RW\_CHUNK\_SIZE\*i, RW\_CHUNK\_SIZE);

write\_to\_cbuff(writePipe, t\_buff, RW\_CHUNK\_SIZE);

i += 1;

}

// as we divide by chunk size there might be some remains

if (numOfBytes & RW\_LAST\_TREE\_BYTES) {

copy\_from\_user(t\_buff, buff + RW\_CHUNK\_SIZE\*numOfIteations, numOfBytes & RW\_LAST\_TREE\_BYTES);

write\_to\_cbuff(writePipe, t\_buff, numOfBytes & RW\_LAST\_TREE\_BYTES);

}

pr\_info(MSG\_PREFIX"recieved message\n");

// unlocking the pipe

mutex\_unlock(writePipe->mutex);

wake\_up\_interruptible(&writePipe->queue);

return numOfBytes;

}

Чтение из буфера устройства:

static ssize\_t device\_read(struct file\* f, char \_\_user\* buff, size\_t numOfBytes, loff\_t\* offset) {

// getting user's pipe

struct pipe\* readPipe = f->private\_data;

while (1) {

// locking it

mutex\_lock(readPipe->mutex);

if (readPipe->bitesAwaible) {

// we're gonna read data by chunks

char t\_buff[RW\_CHUNK\_SIZE] = {0};

int bytesToWrite = numOfBytes;

// we might not have enought data so choose lessier number

if (readPipe->bitesAwaible < numOfBytes) {

bytesToWrite = readPipe->bitesAwaible;

}

// the same as for writing

unsigned int numOfIteations = bytesToWrite >> RW\_SHIFT\_CONSTANT;

unsigned int i = 0;

while (i < numOfIteations) {

read\_from\_cbuff(readPipe, t\_buff, RW\_CHUNK\_SIZE);

copy\_to\_user(buff + RW\_CHUNK\_SIZE \* i, t\_buff, RW\_CHUNK\_SIZE);

i += 1;

}

if (numOfBytes & RW\_LAST\_TREE\_BYTES) {

read\_from\_cbuff(readPipe, t\_buff, bytesToWrite & RW\_LAST\_TREE\_BYTES);

copy\_to\_user(buff + (RW\_CHUNK\_SIZE \* numOfIteations), t\_buff, bytesToWrite & RW\_LAST\_TREE\_BYTES);

}

//

// unlocking pipe

mutex\_unlock(readPipe->mutex);

pr\_info(MSG\_PREFIX "sent message of %d bytes\n", bytesToWrite);

return numOfBytes;

}

// unlocking pipe

mutex\_unlock(readPipe->mutex);

wait\_event\_interruptible(readPipe->queue, readPipe->bitesAwaible);

}

}