Содержание

1 Семинар 1

1.1 Демоны

Информацию о процессах, включающую информацию о процессах демонах, можно получить с помощью консольной команды ps -axj. SMP-архитектура — линукс переписан для поддержки данной архитектруы.

У демона нет управляющего терминала. При этом идентификаторы демона PID (process id), PGID (process group id) и SID (seance id) имеют одинаковое значение.

S - interrupalbe sleep. D - uninterruptable sleep (usually I/O - ожидания завершения ввода-вывода).

Already_running - иногда требуется, чтобы была запущена только одна копия демона

2 Семинар 13.04.2022

Почему для передачи юзеру нужны функции ядра? - разные уровни привелегии, kernel - нулевой, user - 3. Виртуальные адресные пространства имеют процессы, а не юзер, а приложения выполняются на уровне юзера (3). Ядро загружено в физическую память, то есть у него физическое адресное пространство. При этом модули ядра оперируют физическими адресами.

Однако ядро неоднородно. Также как в досе есть резидентая и транзитная часть. Есть часть кода, постоянно находящаяся в памяти, а остальная загружается по мере надобности (в современных системах paging/nonpaging). Ядро не оперирует виртуальными адресами, не смотря на то, что оно использует те же механизмы, что и процессы. Наличие таблиц не подразумевает наличие виртуального адресного пространства. Механизм преобразований, если он реализован аппаратно, отключить нельзя, поскольку начнётся путанница.

Система большую часть времени оперирует виртуальными адресными пространствами. Это означает, что загрузка таблицы в памяти происходит в случае крайней необходимости (когда процессор обращается к данным). Если данные отсутствуют в физической памяти возникает страничное прерывание. В результате обработки табличного прерывания нужная виртуальная страница загружается в физическую память. Т.е. когда ядро обращается к буферу процесса (буферу юзермода) этот буфер может быть не загружен в физическую память. Значит система должна будет проделать все действия по загрузке в физическую память.

Важно то, что к буферу обращается код ядра.

Вторая часть лабораторной по proc связана с написанием загружаемого модуля ядра. Делается передача данных в этом загружаемом модуле из kernel в user и из user в kernel. Существует два способа. Первый - функция ядра copy to user, copy from user и sequence. На sequence определена только передача данных из ядра в user mode. Это организовано потом, что не смотря на то, что proc в режиме пользователя получает много информации о ресурсах, однако разработчику может быть этого не достаточно. Для того, чтобы обратится к расширенной информации нужно написать модуль ядра. Основная задача - передать из ядра в user mode - сиквенсы.

Но нельзя исключать необходимость передачи информации из режима пользователя в режим ядра (передача драйверу настроек).

Sequence-файл (файл последовательностей) - специально написаный интерфейс и библиотека (seq_file.h).

Структура для версии ядра 5.16. Single-функии - интерфейс single-файлов, описанный в ядре.

```
1 <linux/seq_file.h>
2 struct seq_file
3 {
4     char * buf;
5     size_t size;
6     size_t frow;
7     size_t count;
8     size_t pad_until;
9     loft_t index; — смещение
```

```
    loft_t read_pos;
    struct mutex lock;
    const struct seq_operations * op;
    int poll_event;
    const struct file * file;
    void * private;
}

struct seq_operations
{
    void * ( * start)(struct  seq_file * m, loft_t * pos);
    void ( * stop)(struct seq_file * m, void * v);
    void * ( * next)(struct seq_file * m, void * v, loft_t * pos);
    int ( * show)(struct seq_file * m, void * v);
}
```

Данная структура не документирована, вся информация - в процессе изучения структур.
struct file - струткура, описывающая открытые файлы. Определяет одну единственную таблицу открытых файлов в ядре. Все открытые файлы определяются структурой struct file.
Главная задача sequence file - передача данных из ядра в приложение. Пример - терминал. Это процесс режима пользователя, следовательно у него виртуальное адресное пространство.
Сначала вызывается start, затем выполняется итератор next, пока не возвращается null и тогда вызывется stop.

```
int single_open(struct file * , int ( * )(struct seq_file * , void * ), void * ); - с
   оответствует функция show
int single_release(struct inode *, struct file * );
```

single open - открывает файл один раз. Однако в режиме пользователя файл может быть открыт множетсво раз. Если посмотреть на struct file operations, там две структуры - struct inode и struct file, тут inode нету. По сути это опосредование обращение к файлу, открытому в proc и в proc создаются inode. Однако у inode из proc другая нумерация. При этом inode всегда есть, потому, что если создаётся файл у него всегда должен быть inode.

```
#include <linux/seq_file.h>
#define PROC_FILE_NAME "Hello"

static struct proc_dir_entry * proc_size;
static char * out_str;
static int proc_hello_show(struct seq_file * m, void  * v)
{
    int error = 0;
    error = seq_print(m, "%s\n", out_str);
    return error;
}
static int proc_hello_open(struct inode * inode, struct file * file)
```

```
13 {
14     return single_open(file, proc_hello_show, NULL);
15 }
16
17 static const file_operations proc_hello_fops =
18 {
19     .owner = THIS_MODULE,
20     .open = proc_hello_open,
21     .release = single_release,
22     .read = seq_read;
23 };
24
25 static int __init proc_hello_init(void)
26 {
27     out_str = "Hello";
28     proc_file = proc_create_data(PROC_FILE_NAME, S_IRUGO, NULL, proc_hello_fops,
          NULL); S_IRUGO – макрос прав доступа
29     if (!proc_file)
30         return -ENOMEM;
31     return 0;
32 }
33
34 static void __exit proc_hello_exit(void)
35 {
36     if (proc_file)
37         remove_proc_entry(PROC_FILE_NAME, NULL);
38 }
```

struct file operations - важная структура, работающая с драйверами, поэтому она не переписывается.

Разработчики ядра построили промежуточный слой отказоустойчивости, который снижает сложность обмена данными между ядром и режимом пользователя до чего-то вроде принтов. Базовая идея промежуточного слоя заключается в том, что разработчик модуля записывает данные в достаточно большую область памяти.

Для передачи данных в оперативную память используется seq_printf. Подсистема single_file копирует соответствующие данные в режим пользователя.

При этом singe_fileы имеют ограничения. Объем данных, которые может быть передан с помощью этой подсистемы ограничен 64кб. При этом seq_printf котролирует переполнение буфера. Если нужно записать большее количество данных, система создаст такой же буфер, если позволяет память. Если нам нужно реализовать функцию write, нужно писать такую же функцию как в фортунках - copy on user.