1. Билет №???

Single & Sequence

1.1. Воспоминания о Sequence file

Интерфейс sequence file позволяет передавать данные **только из kernel в user**.

Почему нет передачи из user в kernel через интерфейс sequence file?

Потому что **в ядре информации больше** (ещё больше, чем в ф.с. proc), поэтому задача передачи данных из kernel в user считается более важной.

Смысл передачи данных из kernel в user заключается в том, что информация сначала записывается в буфер ядра, и уже из буфера передается в user mode или буфер режима пользователя.

Листинг 1..1: Структуры

```
1  struct seq_operations;
2
3  struct seq_file {
4   char *buf;
5   size_t size;
6   size_t from;
7   size_t count;
8   ...
9   loff_t index;
10  loff_t read_pos;
11  struct mutex lock;
12  const struct seq_operations *op;
13  int poll_event;
14  const struct file *file;
15  void *private;
16 };
17
18 struct seq_operations {
19  void * (*start) (struct seq_file *m, loff_t *pos);
```

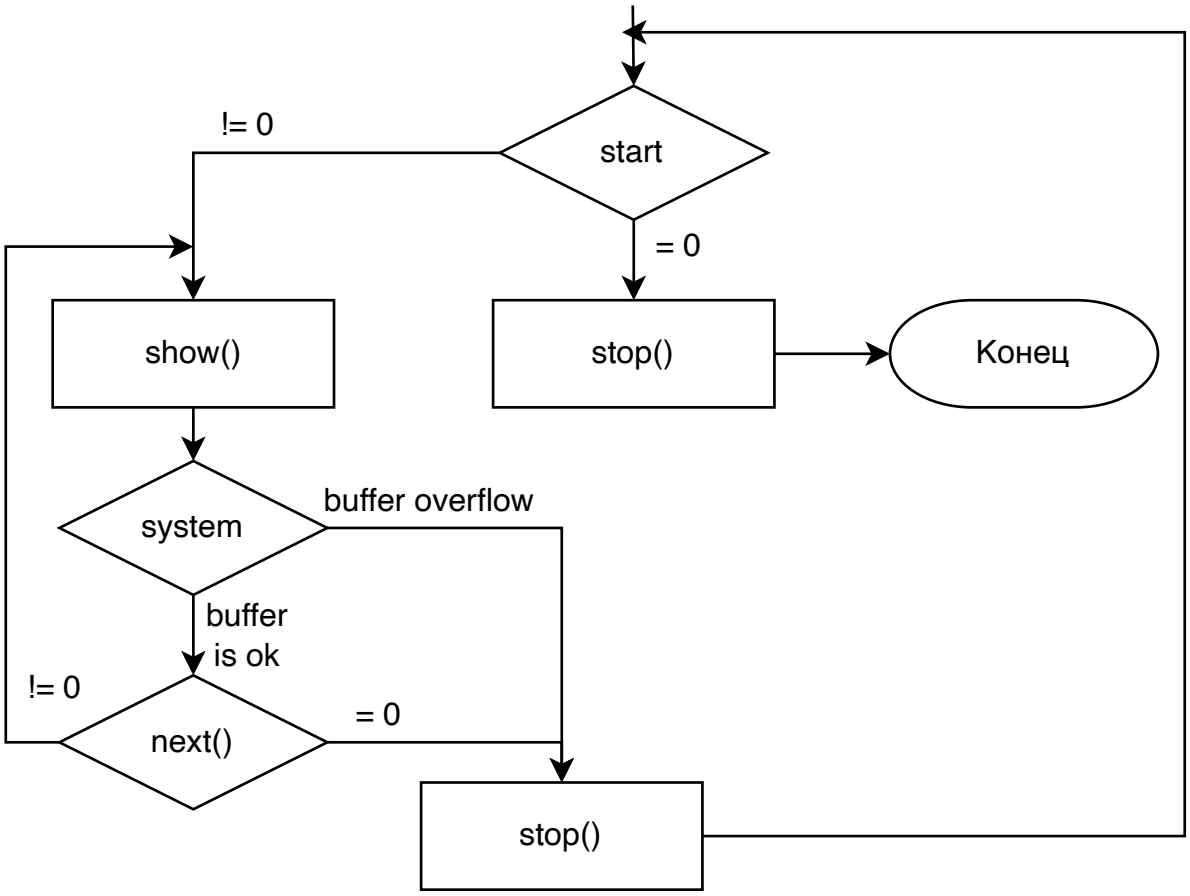
```
20  void (*stop) (struct seq_file *m, void *v);
21  void * (*next) (struct seq_file *m, void *v, loff_t *pos);
22  int (*show) (struct seq_file *m, void *v);
23  };
```

Функция show выполняет все действия по передаче данных.

start() – начало передачи данных. Указатель pos может быть инициализирован нулем. (начало последовательности передаваемых данных);

next() – увеличение указателя pos;

stop() – освобождение памяти.



Это не нужно писать, чисто коммент

Все это добро вызывается при вызове seq_read. То есть не нужно определять функцию чтения, нужно определить итераторы, и использовать seq_read, которая внутри вызывает эти итераторы.

Почему 2 раза может вызываться read()? Потому что при чтении файла мы вызываем read() до тех пор, пока он не вернет 0. Соответственно, первый

read() читает данные, второй возвращает 0 — признак окончания файла.

buffer overflow — буфер заполнен и следующая запись не может быть выполнена (переполнена может быть только помойка).

Варианты решения проблемы:

1. Система может создать дополнительный буфер (размер равен исходному)
2. Если память выделить невозможно, система может вернуть ошибку

1.1.1. Вызов функции stop()

Функция stop() вызывается в 3 контекстах:

1. сразу после start() – окончание вывода данных;
2. после show() – буфер заполнен и может быть создан дополнительный объем памяти (исходный буфер может быть расширен на тот же объем, который запрашивался до этого). Если памяти нет, будет возвращена ошибка (требуется проверка);
3. после next(), когда закончились данные для вывода (нужно либо завершить работу, либо перейти на вывод другого массива данных).

1.2. Интерфейс single file

Single file интерфейс создан для облегчения написания кода, предназначенного для передачи информации из kernel в user.

Почему single называют упрощенным? Потому что нужно определить только функцию `show`, а `start`, `stop`, `next` определять не нужно. Инициализация эих функций происходит в `single_open`.

Single file subsystem позволяет передавать до 64 Кб. Это ограничение только для буфера передачи из kernel в user, так как обратная передача не реализована.

open — главная точка входа.

open -> show -> seq_printf.

Single file subsystem является промежуточным слоем отказоустойчивости за счет снижения сложности обмена данными между пространством ядра и пространством пользователя "до чего-то подобного функции printf()".

proc_show() и proc_my_open()

```c
#include <linux/module.h>
#include <linux/proc_ds.h>
#include <linux/seq_file.h>

#define PROC_FILE_NAME "myfile"

static struct proc_dir_entry *proc_file;
static char *str;

static int proc_show(struct seq_file *m, void *v)
{
  int error = 0;
  error = seq_printf(m, "%s\n", str);
  return error;
}

static int proc_my_open(struct inode *inode, struct file *file)
{
  return single_open(file, proc_show, NULL); // функция интерфейса single file
}
```

proc_ops

```c
static const struct proc_ops proc_fops =
{
  // Ниже предписанные функции соответствующего интерфейса
  .proc_open = proc_my_open,
  .proc_release = single_release,
  .proc_read = seq_read
}
```

Определение функции single_open()

```c
int single_open(struct file *file, int (*show)(struct seq_file *, void *), void *data)
{
  struct seq_operation *op = kmalloc(sizeof(*op), GFP_KERNEL);
  intres = -ENOMEM;
  if (op)
```

```
6     {
7         op->start = single_start; // функции итератора
8         op->stop = single_stop;
9         op->next = single_next;
10        op->show = show;
11        if (!res) ((struct seq_file *)file->private_data)->private = data;
12        else kfree(op);
13    }
14    return res;
15 }
16 EXPORT_SYMBOL(single_open);
```

1.3. Воспоминания о записи в файл

Чтобы иметь возможность передавать данные из user в kernel, необходимо написать функцию write, из которой вызывать copy_from_user.

1.4. Пример на SEQFILE (из семинара, лучше не писать, а указать норм пример из лабы)

```
1  // TODO: добавить пример на SEQFILE
2  static int ct_seq_show(struct seq_file *s, void *v)
3  {
4      printk(KERN_INFO "In show() event = %d\n", *(int*)v);
5      seq_printf(s, "The current value of the event number is %d\n", *(int*)
           v);
6      return 0;
7  }
8
9  static void *ct_seq_start(struct seq_file *s, loff_t *pos)
10 {
11     printk(KERN_INFO "Enter start(), pos = %ld, seq_file pos = %lu\n", *
           pos, s->count);
12     if (pos >= limit)
13     {
14         printk(KERN_INFO "Done\n");
```

```
15          return 0;
16      }
17      ...
18  }
19
20  // start() - начало передачи данных
21  // указатель pos может быть инициализирован 0 (начало последовательности п
        ередаваемых данных)
22  // show() - передача данных
23  // next() - увеличение указателя pos
24
25  // Упрощенный вариант
26  static void *ct_seq_next(struct seq_file *s, void *v, loff_t *pos)
27  {
28      printk(KERN_INFO "next()\n"); // ...
29      (*pos)++; // Увеличение указателя
30      ...
31  }
32
33  // Основная задача stop() - освобождение памяти
34  // В stop() не может быть вызвана функция seq_printf, но может быть вызван
        а printk()
35
36  static struct seq_operations ct_seq_ops = {
37      .start = ct_seq_start,
38      .next = ct_seq_next,
39      .stop = ct_seq_stop,
40      .show = ct_seq_show,
41  };
42
43  static int ct_open(struct inode *inode, struct file *file)
44  {
45      return seq_open(file, &ct_seq_ops);
46  }
47
48  static struct proc_ops proc_fops = {
49      .proc_open = ct_open,
50      .proc_read = seq_read,
```

```
51      .proc_lseek = seq_seek,
52      .proc_release = seq_release, // Функции библиотеки seqfile
53  };
```

1.5. Пример на SINGLEFILE

```
1   #include <linux/module.h>
2   #include <linux/kernel.h>
3   #include <linux/proc_fs.h>
4   #include <linux/seq_file.h>
5   #include <linux/string.h>
6   #include <linux/vmalloc.h>
7   //#include <asm/uaccess.h>
8
9   MODULE_LICENSE("GPL");
10  MODULE_AUTHOR("Karpova Ekaterina");
11
12  #define DIRNAME "seqfiles"
13  #define FILENAME "seqfile"
14  #define SYMLINK "seqfile_link"
15  #define FILEPATH DIRNAME "/" FILENAME
16
17  #define BUF_SIZE PAGE_SIZE
18
19  static struct proc_dir_entry *fortune_file = NULL;
20  static struct proc_dir_entry *fortune_dir = NULL;
21  static struct proc_dir_entry *fortune_link = NULL;
22
23  static char *cookie_buffer = NULL;
24  static int write_index = 0;
25  static int read_index = 0;
26
27  static char tmp[BUF_SIZE];
28
29  ssize_t write(struct file *file, const char __user *buf, size_t len,
        loff_t *offp)
```

```
30 {
31   printk(KERN_INFO "+ seqfile: called write");
32   if(len > BUF_SIZE - write_index + 1)
33   {
34     printk(KERN_ERR "+ seqfile: cookie_buffer overflow");
35     return -ENOSPC;
36   }
37   if(copy_from_user(&cookie_buffer[write_index], buf, len))
38   {
39     printk(KERN_ERR "+ seqfile: copy from user error");
40     return -EFAULT;
41   }
42   write_index += len;
43   cookie_buffer[write_index-1] = '\0';
44   return len;
45 }
46
47 static int seqfile_show(struct seq_file *m, void *v)
48 {
49   printk(KERN_INFO "+ seqfile: called show");
50   //if (read_index >= write_index)
51     //return 0;
52
53   int len = snprintf(tmp, BUF_SIZE, "%s\n", &cookie_buffer[read_index]);
54   seq_printf(m, "%s", tmp);
55   read_index += len;
56   return 0;
57 }
58
59 ssize_t seqfile_read(struct file *file, char __user *buf, size_t count,
     loff_t *offp)
60 {
61   printk(KERN_INFO "+ seqfile: called read");
62   return seq_read(file, buf, count, offp);
63 }
64
65 int seqfile_open(struct inode *inode, struct file *file)
66 {
```

```
67    printk(KERN_INFO "+ seqfile: called open");
68    return single_open(file, seqfile_show, NULL);
69  }
70
71  int seqfile_release(struct inode *inode, struct file *file)
72  {
73    printk(KERN_INFO "+ seqfile: called release");
74    return single_release(inode, file);
75  }
76
77  static struct proc_ops fops = {
78    .proc_open = seqfile_open,
79    .proc_read = seqfile_read,
80    .proc_write = write,
81    .proc_release = seqfile_release,
82  };
83
84  void freemem(void)
85  {
86    if (cookie_buffer)
87      vfree(cookie_buffer);
88    if (fortune_link)
89      remove_proc_entry(SYMLINK, NULL);
90    if (fortune_file)
91      remove_proc_entry(FILENAME, fortune_dir);
92    if (fortune_dir)
93      remove_proc_entry(DIRNAME, NULL);
94  }
95
96  static int __init init_seqfile_module(void)
97  {
98    if (!(cookie_buffer = vmalloc(BUF_SIZE)))
99    {
100     freemem();
101     printk(KERN_ERR "+ seqfile: error during vmalloc");
102     return -ENOMEM;
103   }
104   memset(cookie_buffer, 0, BUF_SIZE);
```

```
105     if (!(fortune_dir = proc_mkdir(DIRNAME, NULL)))
106     {
107       freemem();
108       printk(KERN_ERR "+ seqfile: error during directory creation");
109       return -ENOMEM;
110     }
111     else if (!(fortune_file = proc_create(FILENAME, 0666, fortune_dir, &fops
            )))
112     {
113       freemem();
114       printk(KERN_ERR "+ seqfile: error during file creation");
115       return -ENOMEM;
116     }
117     else if (!(fortune_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
118     {
119       freemem();
120       printk(KERN_ERR "+ seqfile: error during symlink creation");
121       return -ENOMEM;
122     }
123     write_index = 0;
124     read_index = 0;
125     printk(KERN_INFO "+ module loaded");
126     return 0;
127 }
128
129 static void __exit exit_seqfile_module(void)
130 {
131     freemem();
132     printk(KERN_INFO "+ seqfile: unloaded");
133 }
134
135 module_init(init_seqfile_module);
136 module_exit(exit_seqfile_module);
```

1.6. Пример на SEQFILE

```
1 #include <linux/module.h>
```

```
2  #include <linux/kernel.h>
3  #include <linux/proc_fs.h>
4  #include <linux/seq_file.h>
5  #include <linux/string.h>
6  #include <linux/vmalloc.h>
7
8  MODULE_LICENSE("GPL");
9  MODULE_AUTHOR("Aleksey Knyazhev");
10
11 #define DIRNAME "seqfiles"
12 #define FILENAME "seqfile"
13 #define SYMLINK "seqfile_link"
14 #define FILEPATH DIRNAME "/" FILENAME
15
16 #define BUF_SIZE PAGE_SIZE
17
18 static struct proc_dir_entry *seqfile_file = NULL;
19 static struct proc_dir_entry *seqfile_dir = NULL;
20 static struct proc_dir_entry *seqfile_link = NULL;
21
22 static char *cookie_buffer = NULL;
23 static int index = 0;
24
25 ssize_t custom_write(struct file *file, const char __user *buf, size_t len
       , loff_t *offp)
26 {
27  printk(KERN_INFO "+ seqfile: called write");
28  if(len > BUF_SIZE - index + 1)
29  {
30   printk(KERN_ERR "+ seqfile: cookie_buffer overflow");
31   return -ENOSPC;
32  }
33  if(copy_from_user(&cookie_buffer[index], buf, len))
34  {
35   printk(KERN_ERR "+ seqfile: copy from user error");
36   return -EFAULT;
37  }
38  index += len;
```

```
39  cookie_buffer[index-1] = '\n';
40  return len;
41 }
42
43 static int seqfile_show(struct seq_file *m, void *v)
44 {
45  printk(KERN_INFO "+ seqfile: called show");
46  seq_printf(m, "%s", cookie_buffer);
47  return 0;
48 }
49
50 ssize_t seqfile_read(struct file *file, char __user *buf, size_t count,
       loff_t *offp)
51 {
52  printk(KERN_INFO "+ seqfile: called read");
53  return seq_read(file, buf, count, offp);
54 }
55
56 int seqfile_open(struct inode *inode, struct file *file)
57 {
58  printk(KERN_INFO "+ seqfile: called open");
59  return single_open(file, seqfile_show, NULL);
60 }
61
62 int seqfile_release(struct inode *inode, struct file *file)
63 {
64  printk(KERN_INFO "+ seqfile: called release");
65  return single_release(inode, file);
66 }
67
68 static struct proc_ops fops = {
69   .proc_open = seqfile_open,
70   .proc_read = seqfile_read,
71   .proc_write = custom_write,
72   .proc_release = seqfile_release,
73 };
74
75 void freemem(void)
```

```
76 {
77  if (cookie_buffer)
78   vfree(cookie_buffer);
79  if (seqfile_link)
80   remove_proc_entry(SYMLINK, NULL);
81  if (seqfile_file)
82   remove_proc_entry(FILENAME, seqfile_dir);
83  if (seqfile_dir)
84   remove_proc_entry(DIRNAME, NULL);
85 }
86
87 static int __init init_seqfile_module(void)
88 {
89  if (!(cookie_buffer = vmalloc(BUF_SIZE)))
90  {
91   freemem();
92   printk(KERN_ERR "␣seqfile:␣error␣during␣vmalloc");
93   return -ENOMEM;
94  }
95  memset(cookie_buffer, 0, BUF_SIZE);
96  if (!(seqfile_dir = proc_mkdir(DIRNAME, NULL)))
97  {
98   freemem();
99   printk(KERN_ERR "␣seqfile:␣error␣during␣directory␣creation");
100  return -ENOMEM;
101  }
102  else if (!(seqfile_file = proc_create(FILENAME, 0666, seqfile_dir, &fops)
        ))
103  {
104   freemem();
105   printk(KERN_ERR "␣seqfile:␣error␣during␣file␣creation");
106   return -ENOMEM;
107  }
108  else if (!(seqfile_link = proc_symlink(SYMLINK, NULL, FILEPATH)))
109  {
110   freemem();
111   printk(KERN_ERR "␣seqfile:␣error␣during␣symlink␣creation");
112   return -ENOMEM;
```

```
113   }
114   index = 0;
115   printk(KERN_INFO "+ module loaded");
116   return 0;
117  }
118
119  static void __exit exit_seqfile_module(void)
120  {
121   freemem();
122   printk(KERN_INFO "+ seqfile: unloaded");
123  }
124
125  module_init(init_seqfile_module);
126  module_exit(exit_seqfile_module);
```