Advanced Operating Systems

Multi-flow device file

University of Rome Tor Vergata

Di Battista Mattia (0304938)

March 30, 2022

Contents

Chapter 1

How To Use

Chapter 2

Specification

This specification is related to a Linux device driver implementing low and high priority flows of data. Through an open session to the device file a thread can read/write data segments. The data delivery follows a First-in-First-out policy along each of the two different data flows (low and high priority). After read operations, the read data disappear from the flow. Also, the high priority data flow must offer synchronous write operations while the low priority data flow must offer an asynchronous execution (based on delayed work) of write operations, while still keeping the interface able to synchronously notify the outcome. Read operations are all executed synchronously. The device driver should support 128 devices corresponding to the same amount of minor numbers.

The device driver should implement the support for the *ioctl(..)* service in order to manage the I/O session as follows:

- setup of the priority level (high or low) for the operations
- blocking vs non-blocking read and write operations
- setup of a timeout regulating the awake of blocking operations

A few Linux module parameters and functions should be implemented in order to enable or disable the device file, in terms of a specific minor number. If it is disabled, any attempt to open a session should fail (but already open sessions will be still managed). Further additional parameters exposed via VFS should provide a picture of the current state of the device according to the following information:

- enabled or disabled
- number of bytes currently present in the two flows (high vs low priority)
- number of threads currently waiting for data along the two flows (high vs low priority)

Chapter 3

Operations

3.1 Setting Operation

The user can set four modes of operation for his working session:

1. low priority
2. high priority
3. blocking
4. non blocking

The data structure to store these parameters is:

```
//FILE: info.h
typedef struct _session{
   bool priority;
   bool blocking;
   unsigned long timeout;
} session;
```

and is instantiated in *dev_open()*:

```
//FILE: multi_flow.c
session = kmalloc(sizeof(session), GFP_KERNEL);
//...
if (session == NULL) //...
session->priority = HIGH_PRIORITY;
```

```
session->blocking = NON_BLOCKING;
session->timeout = 0;
file->private_data = session;
```

The field *unsigned long timeout* indicates the maximum time in blocking operations (i.e. read, write), for which the user waits.

Parameter setting is done within *dev_ioctl()*:

```
//FILE: multi_flow.c
static long dev_ioctl(struct file *filp, unsigned int command,
    unsigned long param){
   session *session;
   session = filp->private_data;

   switch (command){
   case 3:
      session->priority = LOW_PRIORITY;
      //...
      break;
   case 4:
      session->priority = HIGH_PRIORITY;
      //...
      break;
   case 5:
      session->blocking = BLOCKING;
      //...
      break;
   case 6:
      session->blocking = NON_BLOCKING;
      //...
      break;
   case 7:
      session->timeout = param;
      //...
      break;
   default:
      //...
   }
return 0;
}
```

To associate the session to a user, the field *private_data* of *struct file *filp* has been used.

3.2 Write Operation

Unlike the read op. (see Chap. 3.3) the write can be done with low priority (based on delayed work). In this way the operation is added to a work queue[1] and performed by system daemons.

```
//FILE: multi_flow.c
if (session->priority == HIGH_PRIORITY){
   priority_obj = &the_object[HIGH_PRIORITY];
   //...
   ret = write(priority_obj, buff, off, len, session, minor);
}else{
   priority_obj = &the_object[LOW_PRIORITY];
   //...
   ret = put_work(filp, buff, len, off, priority_obj, session,
      minor);
   if (ret != 0) //...
   ret = len;
}
```

The procedures for doing so are:

```
//FILE: write.h
int write(object_state *,
        const char *,
        loff_t *,
        size_t,
        session *,
        int);
void delayed_write(unsigned long);
long put_work(struct file *,
        const char *,
        size_t,
        loff_t *,
        object_state *,
```

[1] One is defined for each flow in the procedure *init_module()*, with *init_waitqueue_head()*.

```
        session *,
        int);
```

For each write operation a new node and a buffer (both with *kmalloc()*[2] of size equal to the Bytes the user wants to write are allocated. Finally the new node is added to the list.

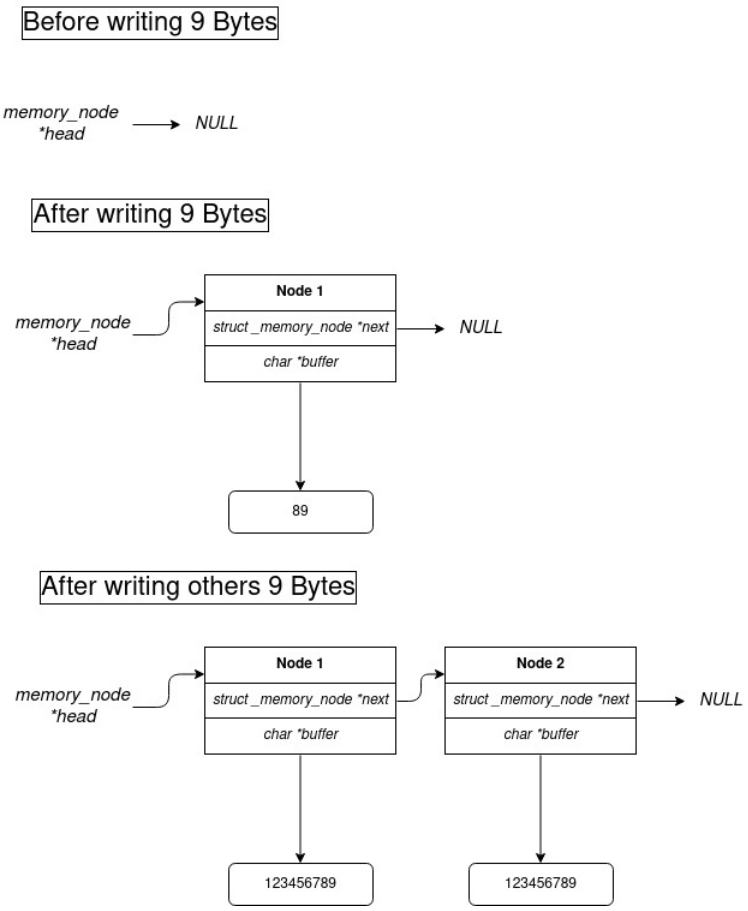

Figure 3.1: Example of write operation

[2]This API allows to reserve Bytes and not entire pages like *get_free_page()*.

3.3 Read Operation

Read operations are all performed synchronously (without delayed work as for write op.), when an invocation occurs the data structure for one of the two flows is set:

```
//FILE: multi_flow.c
if (session->priority == HIGH_PRIORITY){
   priority_obj = &the_object[HIGH_PRIORITY];
   //...
}else{
   priority_obj = &the_object[LOW_PRIORITY];
   //...
}
ret = read(priority_obj, buff, off, len, session, minor);
```

In addition, they involved deleting the bytes read by the user, on the current flow. To do this a linked list was used, in which each node keeps the following fields:

```
//FILE: info.h
typedef struct _memory_node{
   char *buffer;
   struct _memory_node *next;
} memory_node;
```

Each node points to its own buffer, the size of which can then be different for all items in the linked list. Buffers are allocated in write operations (see Chap.3.2). If the user requests the reading of a smaller number of bytes, than those written on all nodes, the deletion happens for the read bytes only and not on the entire buffer (see Fig.3.2). Reading begins from the first written node (**First-in-First-out policy**).

The procedures for doing this are:

```
//FILE: read.h
memory_node *shift_buffer(int, int, memory_node *);
int read(object_state *, const char *, loff_t *, size_t, session
    *, int);
```

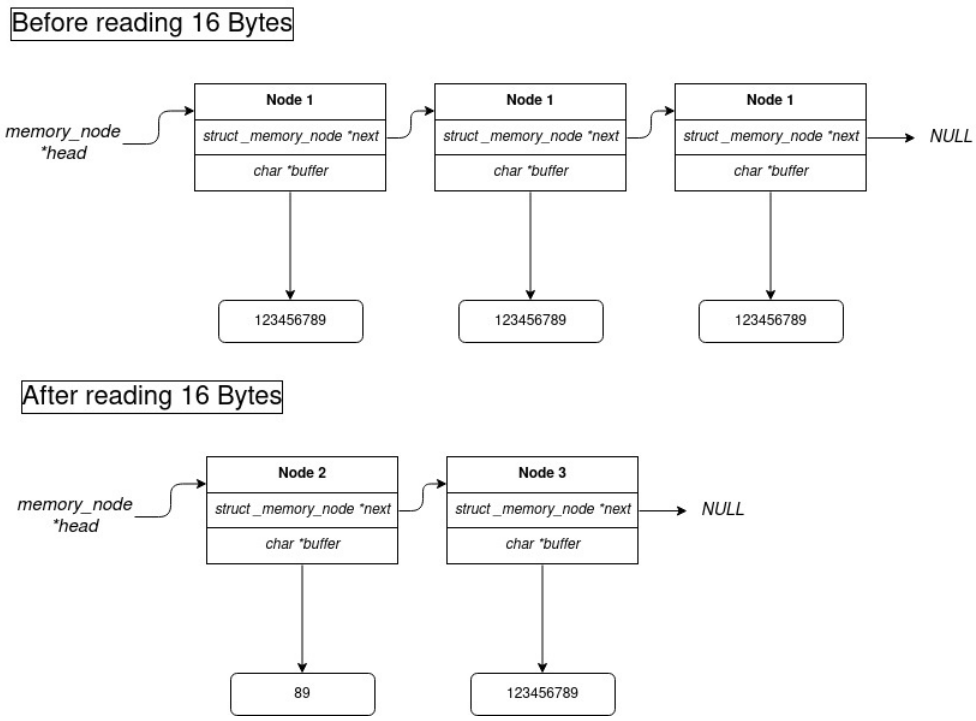

Figure 3.2: Example of read operation

Chapter 4

Parameters

The required module parameters to deploy are:

1. enable or disable the device file
2. number of bytes currently present in the two flows (high vs low priority)
3. number of threads currently waiting for data along the two flows (high vs low priority)

implemented with:

```
//FILE: info.h
static bool enabled_device[MINORS];
module_param_array(enabled_device, bool, NULL, 0744);
MODULE_PARM_DESC(...);

static int hp_bytes[MINORS];
module_param_array(hp_bytes, int, NULL, 0744);
MODULE_PARM_DESC(...);

static int lp_bytes[MINORS];
module_param_array(lp_bytes, int, NULL, 0744);
MODULE_PARM_DESC(...);

static int hp_threads[MINORS];
module_param_array(hp_threads, int, NULL, 0744);
MODULE_PARM_DESC(...);
```

```
static int lp_threads[MINORS];
module_param_array(lp_threads, int, NULL, 0744);
MODULE_PARM_DESC(...);
```

Chapter 5

User

Chapter 6

Reference