Advanced Operating Systems

Multi-flow device file

University of Rome Tor Vergata

Di Battista Mattia (0304938)

April 2, 2022

Contents

Chapter 1

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 2

Operations

2.1 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;
```

It is single for each thread, unlike the *_object_state* unique for each device file.

```
//FILE: info.h
typedef struct _object_state{
   struct mutex operation_synchronizer;
   memory_node *head;
   wait_queue_head_t wq;
```

```
} object_state;
```

__session 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 (expressed as microseconds) 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 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.

2.2 Blocking Operation

In a blocking operation the thread goes to sleep, in the respective wait queue of the object[1], in case it fails to take the lock.

```
//FILE: common.h
static int blocking(unsigned long timeout, struct mutex *mutex,
    wait_queue_head_t *wq){
   //...
   timeout = msecs_to_jiffies(timeout);
   //Returns 0, if the condition evaluated to false after the
      timeout elapsed
   val = wait_event_timeout(*wq, mutex_trylock(mutex), timeout);
   //...
}
```

The thread waits for a certain time, until the lock is released. Time passed to *wait_event_timeout()* is converted from microseconds to jiffies with *msecs_to_jiffies()*. An error code[2] is returned if the timer has expired.

2.3 Write Operation

Unlike the read op. (see Chap. 2.4) the write can be done with low priority (based on delayed work). In this way the operation is performed by system daemons[3].

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

[1]There are two for each object: a low and high priority

[2]EAGAIN: Resource temporarily unavailable.

[3]Unlike tasklets, work queues allow you to queue blocking activities, executed by threads in charge of doing this.

```
}else{
   priority_obj = &the_object[LOW_PRIORITY];
   //...
   ret = put_work(filp, buff, len, off, priority_obj, session,
      minor);
   if (ret != 0) //...
   ret = len;
}
```

To define deferred work:

```
//FILE: write.h
long put_work(struct file *filp,
   const char *buff,
   size_t len,
   loff_t *off,
   object_state *the_object,
   session *session,
   int minor){

   //...

   __INIT_WORK(&(the_task->the_work), (void *)delayed_write,
      (unsigned long)(&(the_task->the_work)));

   schedule_work(&the_task->the_work);

   return 0;
}
```

For each write operation a new node and buffer (both with *kmalloc()*[4] are allocated. The buffer's size is equal to the amount of Bytes the user wants to write. Finally the new node is added to the list (one for each stream).

copy_from_user() may sleep if pagefaults are enabled. That's why it is invoked before requiring the lock on the flow[5].

```
//FILE: write.h
```

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

[5]As an alternative could be invoked the *__copy_to_user_inatomic()* (see https://www.kernel.org/doc/htmldocs/kernel-api/API—copy-to-user-inatomic.html)

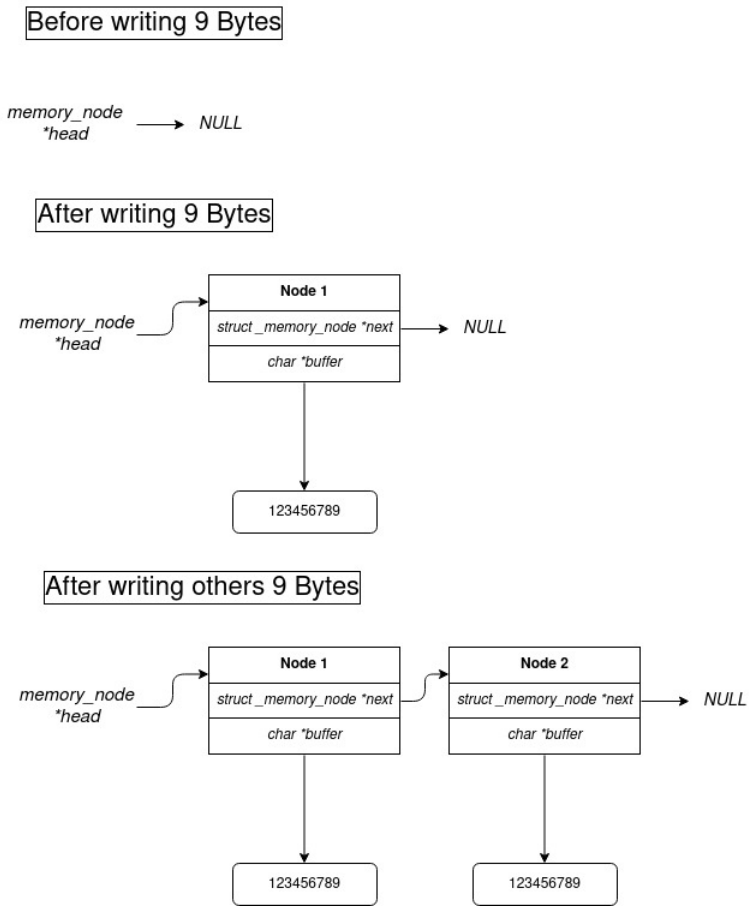

Figure 2.1: Example of write operation

```
int write(object_state *the_object,
   const char *buff,
   loff_t *off,
   size_t len,
   session *session,
   int minor){

   //...
   node = kmalloc(sizeof(memory_node), GFP_ATOMIC);
   buffer = kmalloc(len, GFP_ATOMIC);
   if (node == NULL || buffer == NULL)
```

```
   {
   printk("%s: unable to allocate a memory\n", MODNAME);
   return -ENOMEM;
   }

   ret = copy_from_user(buffer, buff, len);

   wq = get_lock(the_object, session, minor);
   if (wq == NULL) return -EAGAIN;
   //...
```

2.4 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.2.3). 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.2.2). Reading begins from the first written node (**First-in-First-out policy**).

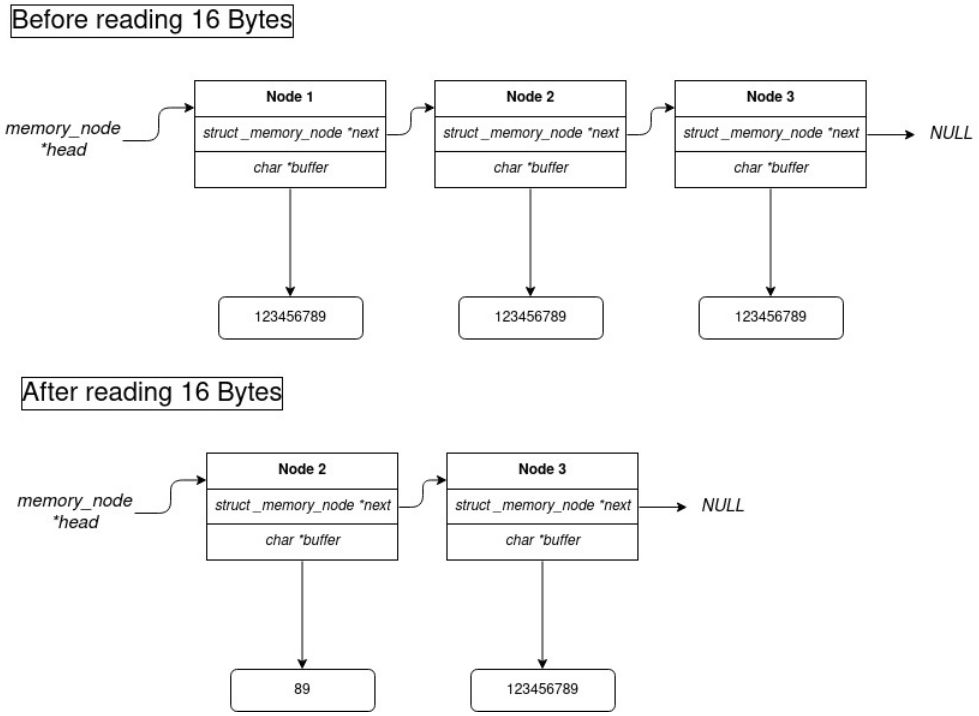

Figure 2.2: Example of read operation

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);
```

Chapter 3

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 int enabled_device[MINORS];
module_param_array(enabled_device, int, NULL, 0660);
MODULE_PARM_DESC(...);

static int hp_bytes[MINORS];
module_param_array(hp_bytes, int, NULL, 0660);
MODULE_PARM_DESC(...);

static int lp_bytes[MINORS];
module_param_array(lp_bytes, int, NULL, 0660);
MODULE_PARM_DESC(...);

static int hp_threads[MINORS];
module_param_array(hp_threads, int, NULL, 0660);
MODULE_PARM_DESC(...);
```

```
static int lp_threads[MINORS];
module_param_array(lp_threads, int, NULL, 0660);
MODULE_PARM_DESC(...);
```

Chapter 4

How To Use

Commands for kernel module (in */code/*):

```
#Compile kernel module (on /multi_flow/code/)
make all

#Clean objects file
make clean

#Insertion kernel module
sudo insmod multi_flow.ko

#Remove kernel module
sudo rmmod multi_flow.ko
```

Commands for parameters (in */code/*):

```
#Show kernel module's parameters (i.e. enabled_device, hp_bytes,
    lp_bytes, hp_threads, lp_threads)
make show-device
make show-hp_bytes
make show-lp_bytes
make show-hp_threads
make show-lp_threads
```

Commands for user (in */code/user*):

```
#Compile user (on /multi_flow/code/user/)
```

```
make

#Run user
sudo ./user /dev/test 240 1

#Disable a list of devices by index (file in /code/user/)
sudo python3 script.py 1 2 45 127
```