Advanced Operating Systems

Multi-flow device file

University of Rome Tor Vergata

Di Battista Mattia (0304938)

April 10, 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{
   int priority;
   int blocking;
   unsigned long timeout;
} session;
```

It is unique for each thread, unlike the *_object_state* defined for every 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 = kzalloc(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) awaited in blocking operations (see Sec. 2.2).

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;
   //...
}
```

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

2.2 Blocking Operation

For blocking operations 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);
   //...
}
```

To wake all threads sleeping on the given wait queue, *wake_up()* is invoked at the end of every operations (after having released the lock).

Threads wait 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 read operations (see Sec. 2.4), 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);
}else{
   priority_obj = &the_object[LOW_PRIORITY];
```

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

[2]EAGAIN: Resource temporarily unavailable.

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

```
   //...
   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,
   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;
}
```

Moreover in the case the user is notified that his bytes have been written. This implies that low priority write operation cannot fail, so deamon waits until the lock is released.

```
// FILE: write.h
if (session->priority == LOW_PRIORITY){
   __sync_add_and_fetch (&lp_threads[minor], 1);
   mutex_lock(&(the_object->operation_synchronizer));
   __sync_add_and_fetch (&lp_threads[minor], -1);
   wq = &the_object->wq;
}else{
   wq = get_lock(the_object, session, minor);
   if (wq == NULL)
```

```
    return -EAGAIN;
}
```

For each write operation a new *memory_node* (see Sec. 2.4) and buffer (both with *kzalloc()*[4]) are allocated. The buffer's size is equal to the amount of Bytes written from the user. Finally the new node is added to a list (one for every stream). See Image 2.1.

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

   //...
   node = kzalloc(sizeof(memory_node), GFP_ATOMIC);
   buffer = kzalloc(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;
   //...
```

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

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

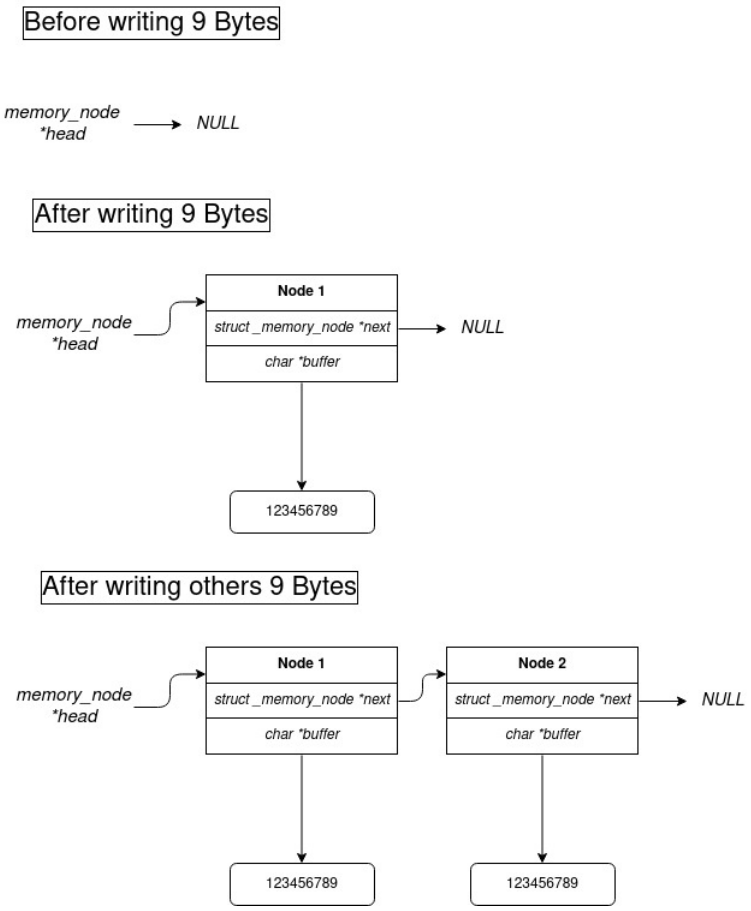

Figure 2.1: Example of write operation

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 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. Buffers are allocated in write operations (see Sec. 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 only read bytes, not on the entire buffer (see Fig. 2.2). Reading and deletion begin from the first 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);
```

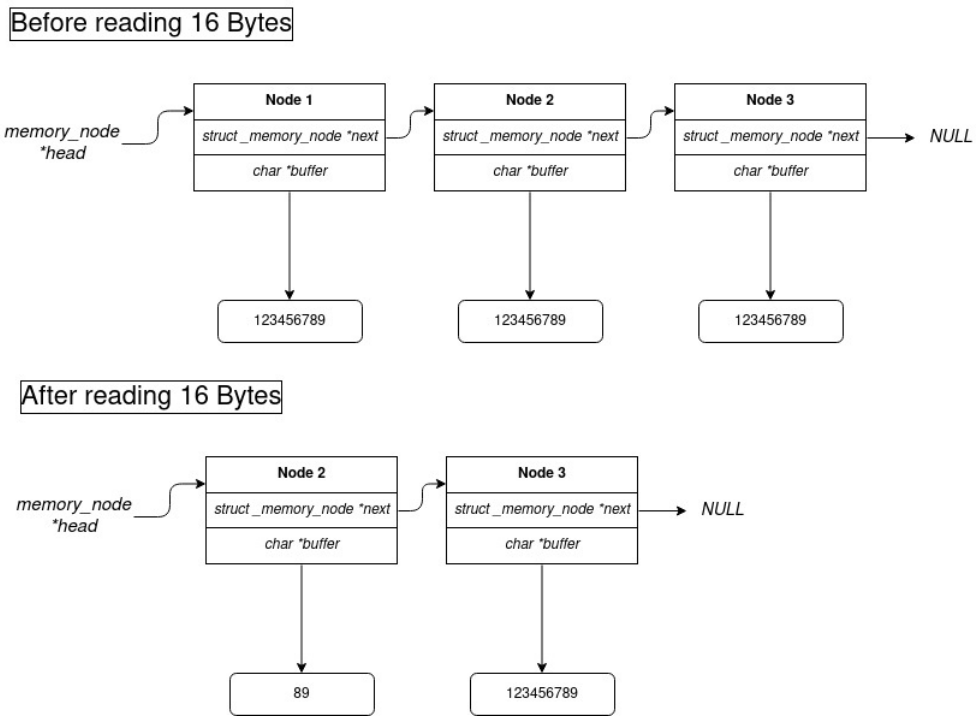

Figure 2.2: Example of read operation

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

4.1 Organization

The structure of the code is as follows:

1. in */code/*
   (a) *common.h*
   (b) *info.h*
   (c) *Makefile*
   (d) *multi_flow.h*
   (e) *read.h*
   (f) */user/*
   (g) *write.h*
2. in */code/user/*
   (a) *Makefile*
   (b) *script.py*
   (c) *user.c*

4.2 Commands

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
make

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

#Disable a list of devices by index
sudo python3 script.py 0 1 2 45 127
```

To test blocking operations a macro is defined:

```
//FILE: info.h
#define TEST
```

TEST does not release the lock after a write, and does not wake up the threads in the wait queue.

```
//FILE write.c
#ifndef TEST
   mutex_unlock(&(the_object->operation_synchronizer));
   wake_up(wq);
#endif
```