Multi-flow device file

Di Battista Mattia (0304938)

23 03 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 Read Operation

Read operation 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:

```
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). 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.3.1). Reading begins from the first written node (**First-in-First-out policy**).

The procedures for doing this are:

```
memory_node *shift_buffer(int, int, memory_node *);
int read(object_state *, const char *, loff_t *, size_t, session
    *, int);
```

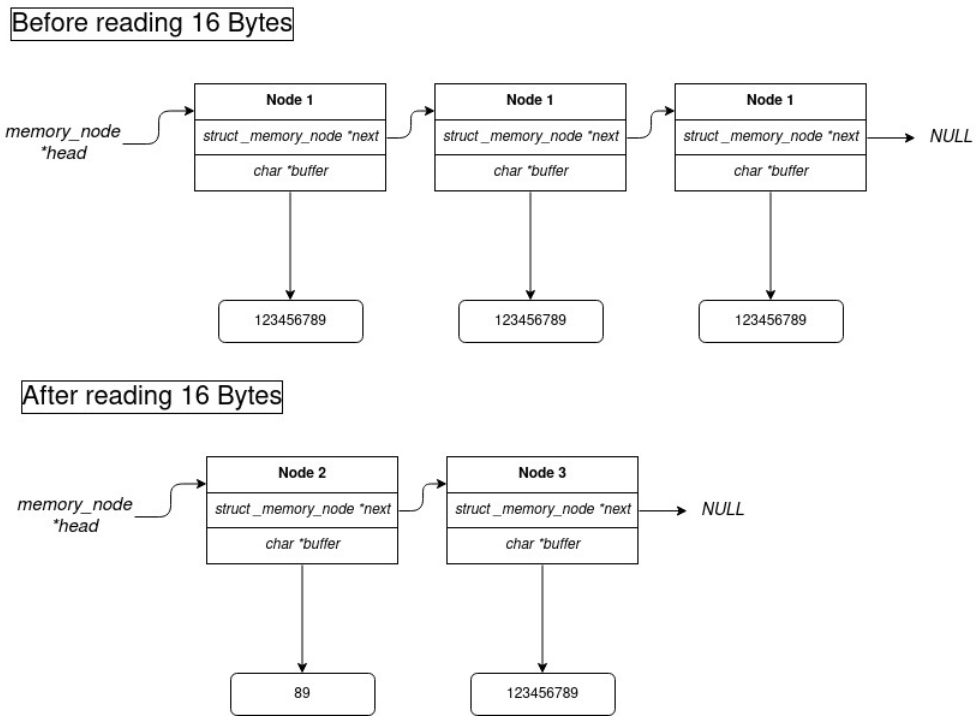

Figure 2.1: Example of read operation

Chapter 3

Write Operation

For each write operation a new node and a buffer (both with *kmalloc()*[1] of size equal to the Bytes the user wants to write are allocated. Finally the new node is added to the list.

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

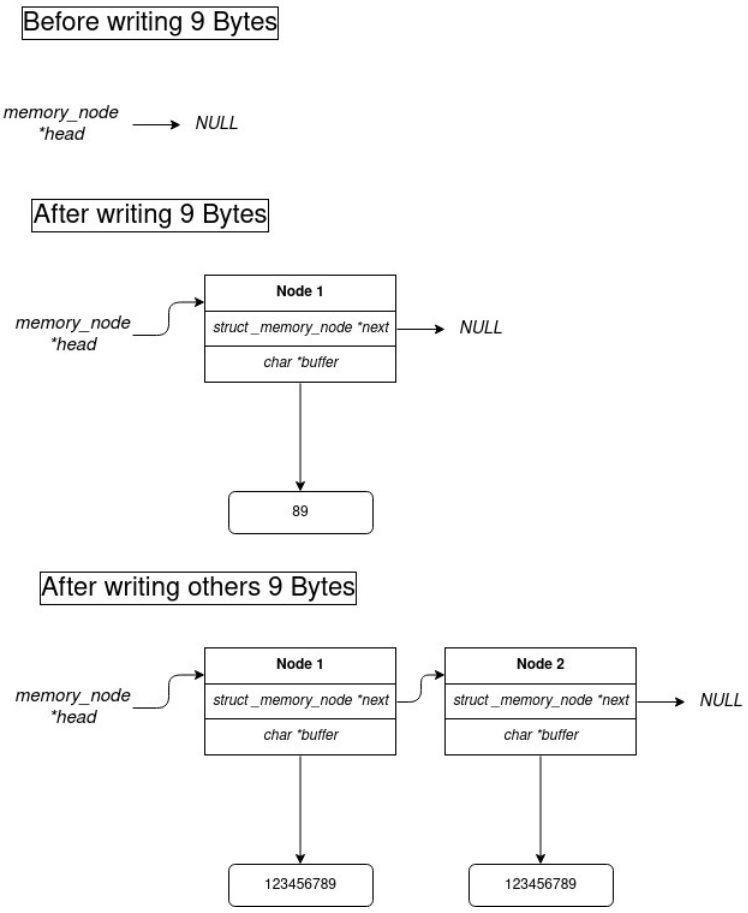

Figure 3.1: Example of write operation

Chapter 4

Reference