**National University of Computer & Emerging Sciences**

**Karachi Campus**



**Project Report**

**Memory Management:**

**[]**

**Course Teacher:**

**Sir. Ghufran Ahmed**

**Ms. Safia**

**Group Members:**

**Saud Ahmed Abbasi 19K-0229**

**Zaid Bin Shahab 19K-1512**

**Umer Ahmed 19K-0181**

# **Memory Management with Drivers and Device File Access:**

## **Small Definitions:**

**Drivers:**  
Device drivers take on a special role in the Linux kernel. They are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works.

User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and “plugged in” at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

**Character Drivers:**  
A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*, *read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

## **Sysfs:**

Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices.The procfs is used to export the process specific information and the debugfs is used to used for exporting the debug information by the developer. At the heart of the sysfs model is the Kobject. Kobject is the glue that binds the sysfs and the kernel.

## **Example of Sysfs initialization:**

**static** **int** \_\_init **mymodule\_init** (**void**)

{

**int** error **=** 0;

pr\_debug("Module initialized successfully \n");

example\_kobject **=** kobject\_create\_and\_add("kobject\_example",

kernel\_kobj);

**if**(**!**example\_kobject)

**return** **-**ENOMEM;

error **=** sysfs\_create\_file(example\_kobject, **&**foo\_attribute.attr);

**if** (error)

{

pr\_debug("failed to create the foo file in /sys/kernel/kobject\_example \n");

}

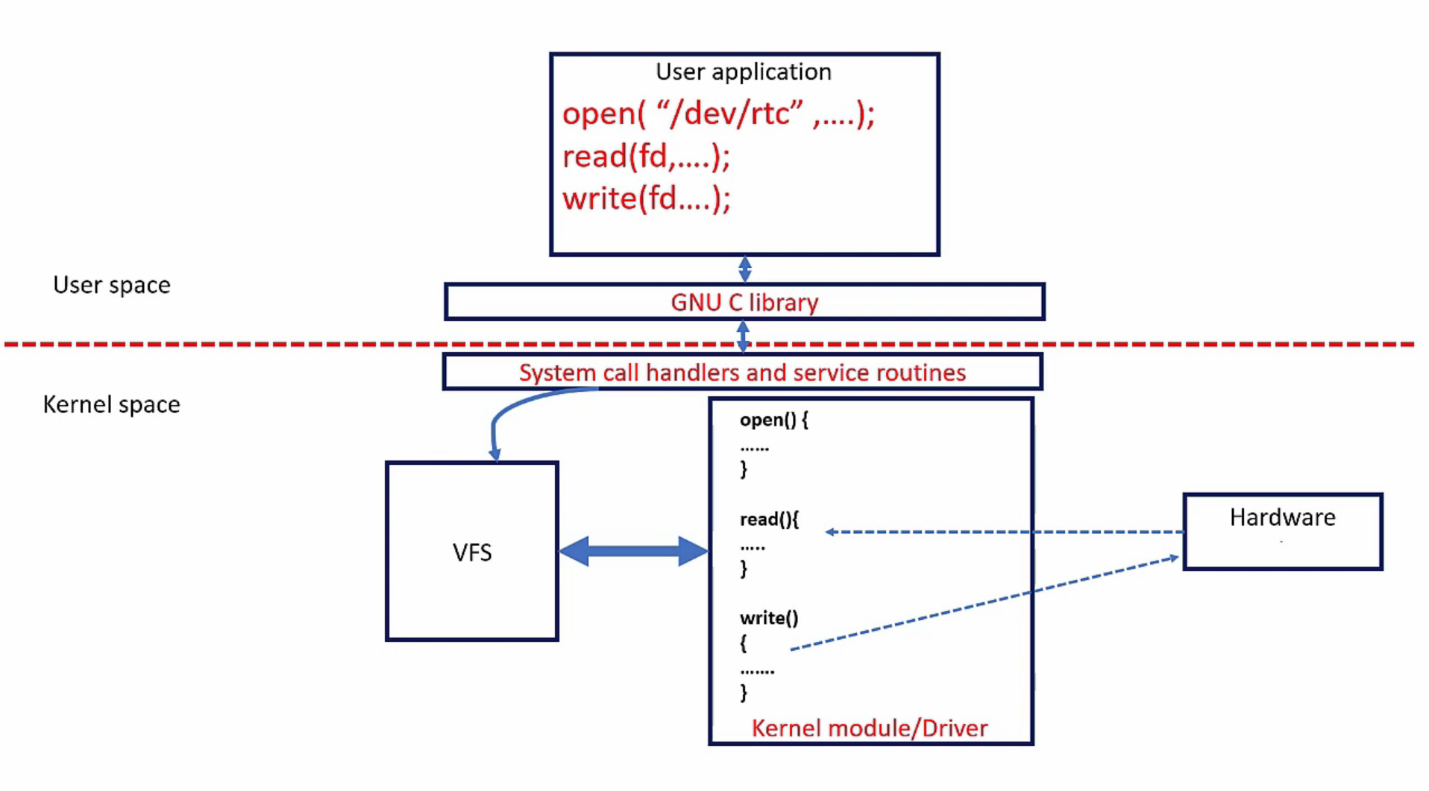
**return** error;

}

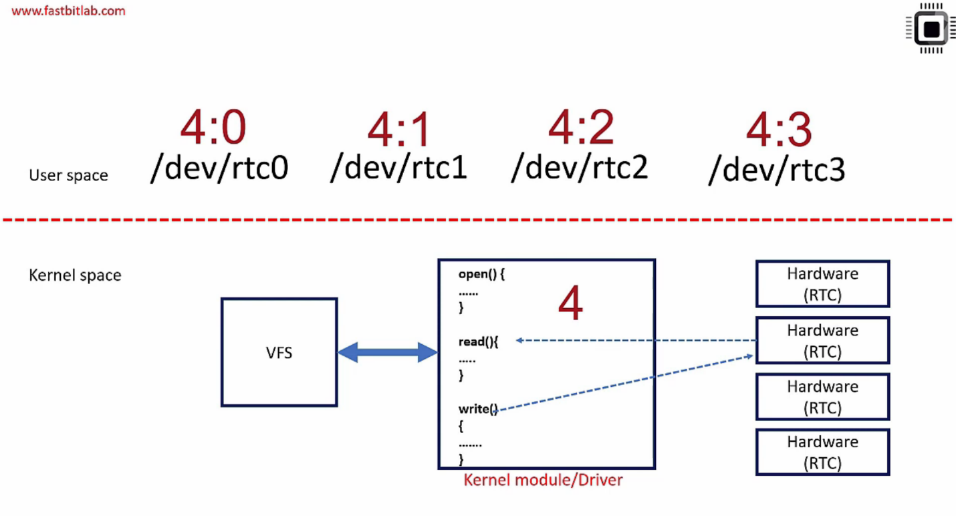
To create a single file attribute we are going to use ‘sysfs\_create\_file’. One can use another function ‘ sysfs\_create\_group ‘ to create a group of attributes.

static struct kobj\_attribute foo\_attribute =\_\_ATTR(foo, 0660, sysfs\_show, sysfs\_store);

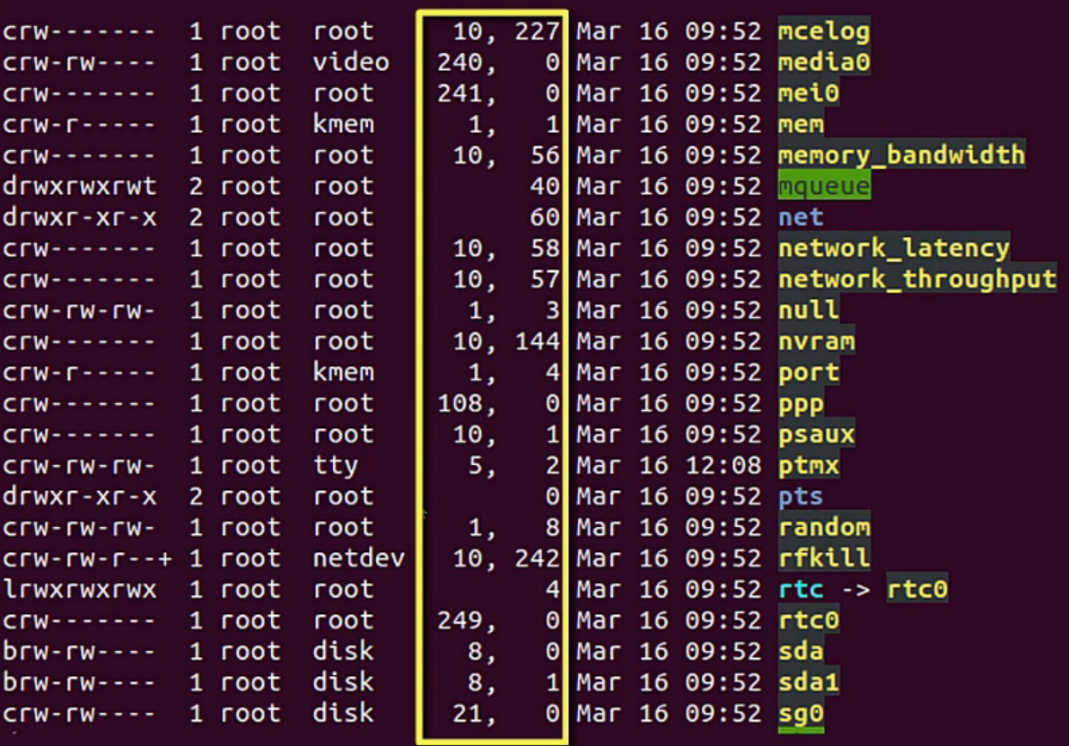
## **User Application invoking driver’s open methods:**



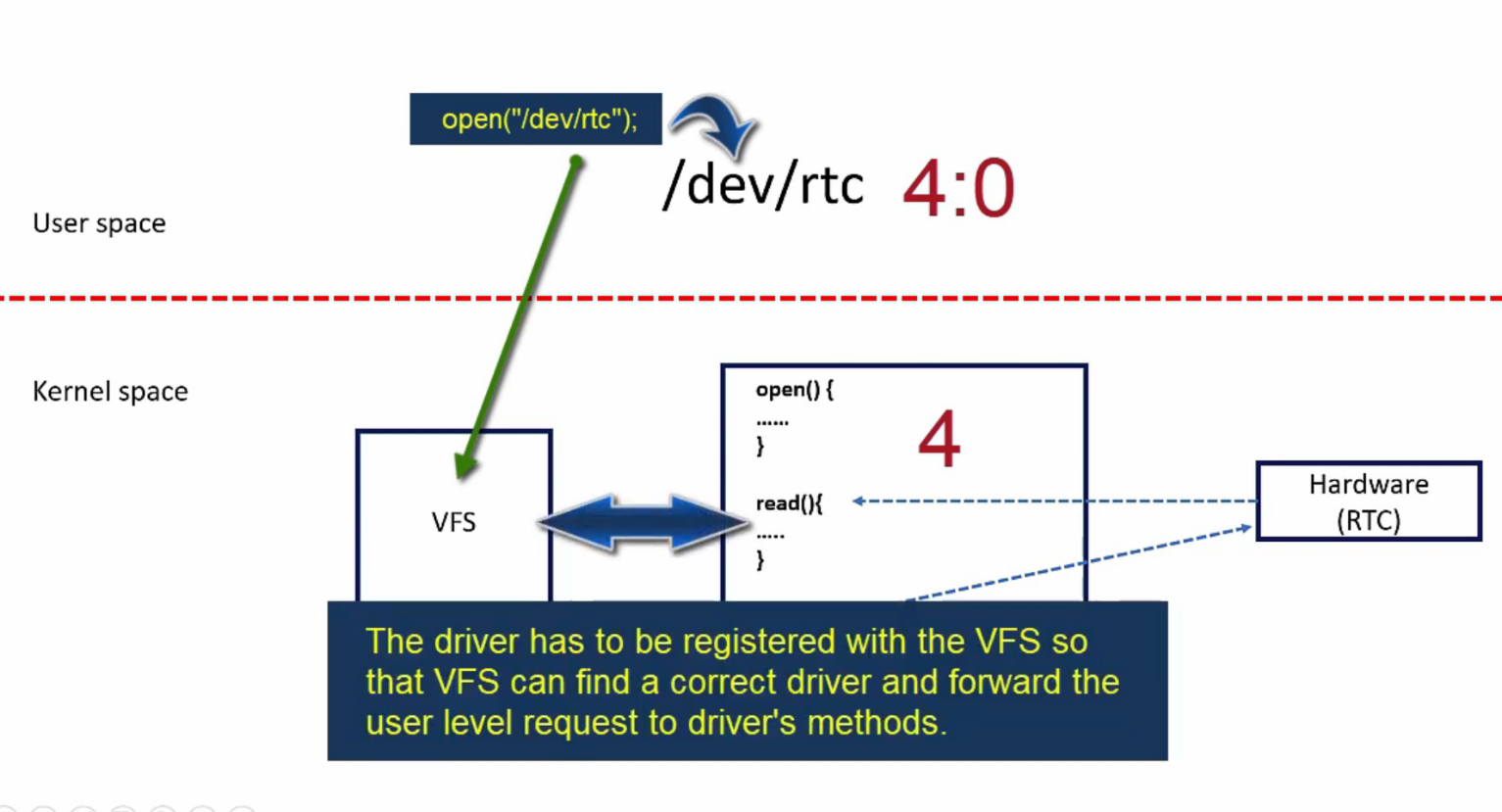
## **Major:Minor Numbers for Devices:**

****

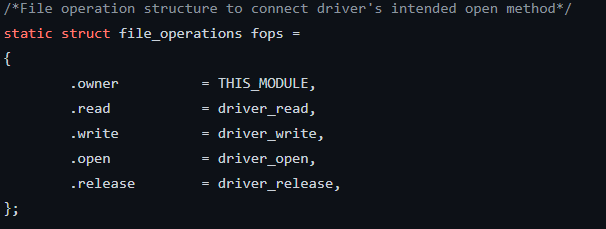
## **Major:Minor Number example (can be viewed using ls -l command):**

****

## **VFS Character Device Open Method Registration:**



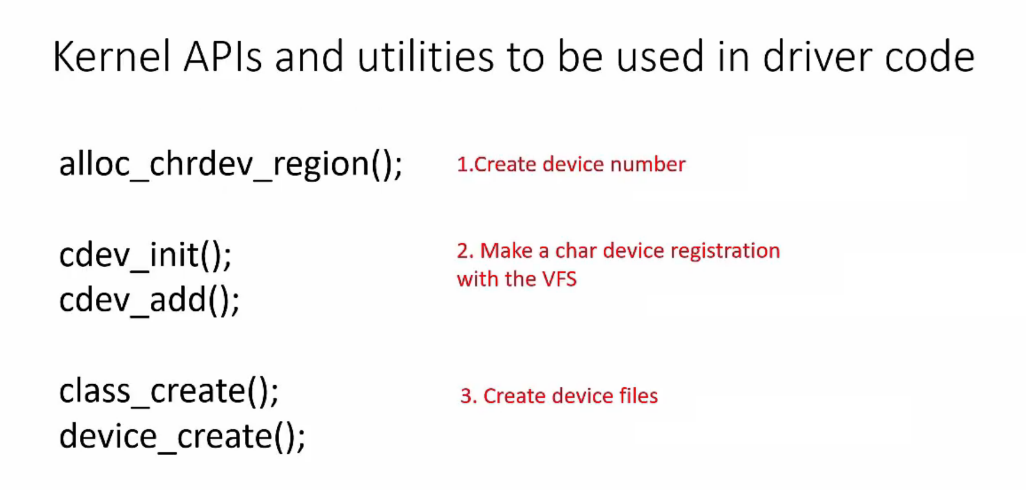
Structure that shows how internal variables defined inside the file\_operations structure are pointed to the driver’s open methods, the static struct here acting as a constructor of sorts:

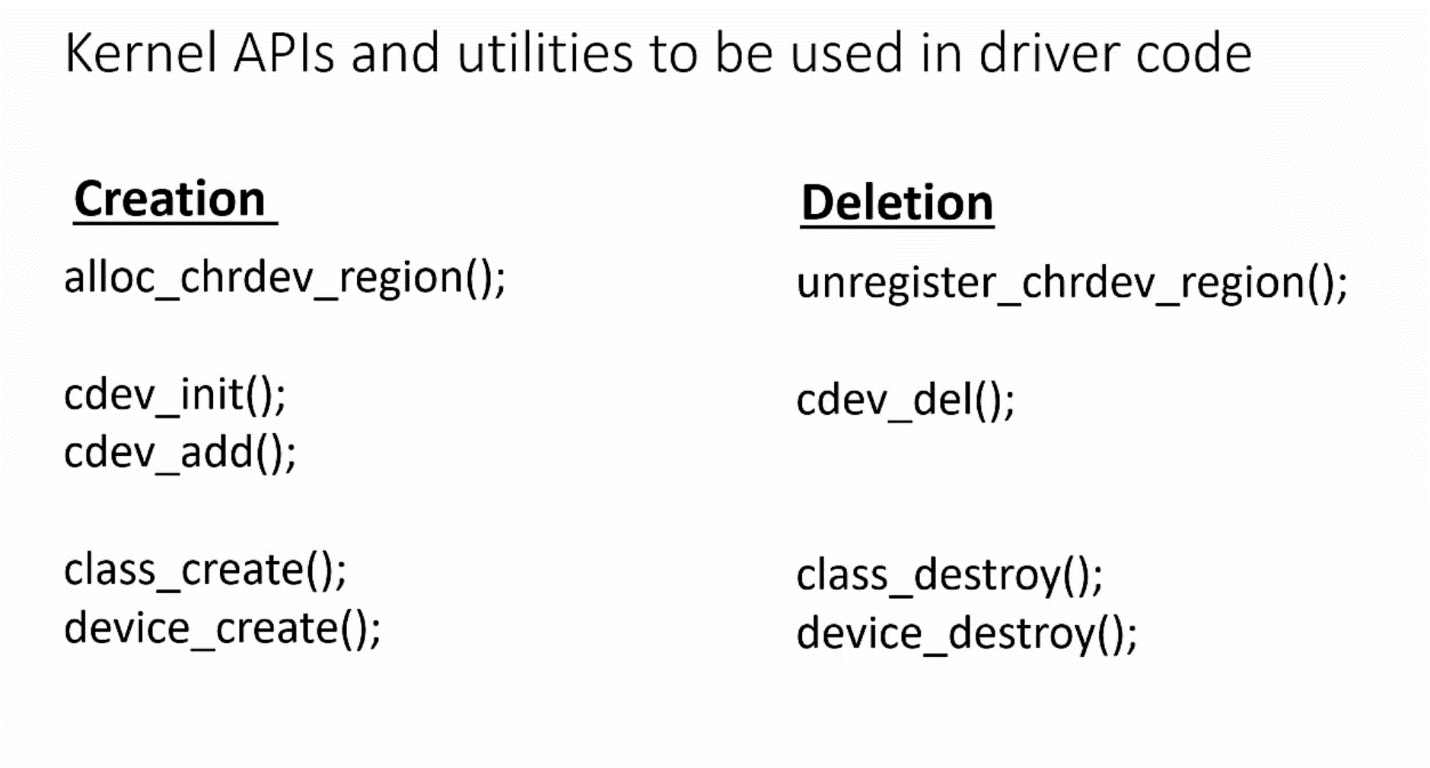


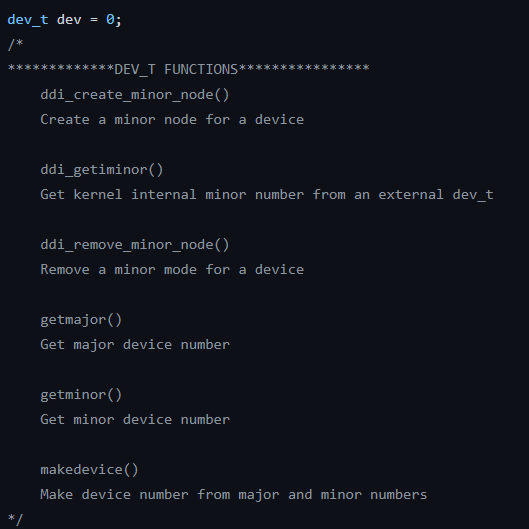
**Cdev, what is it ?**  
The struct cdev is the kernel's internal structure that represents char devices. So cdev\* i\_cdev field of struct inode is a pointer to cdev structure while the inode refers to the char device file. Therefore if the kernel has to invoke the character device, it has to register a structure of cdev type.

**To establish connection device file access and the driver, we have to:**  
1- Create device number  
2- Create device file  
3- Make a character device registration with the VFS  
4- Implement the driver’s operation methods for open, write, read etc

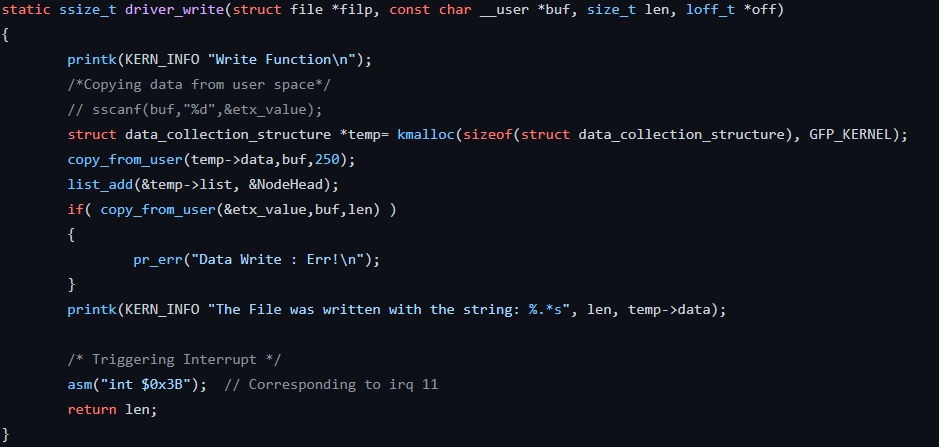
## **For all of this we use the functions:**



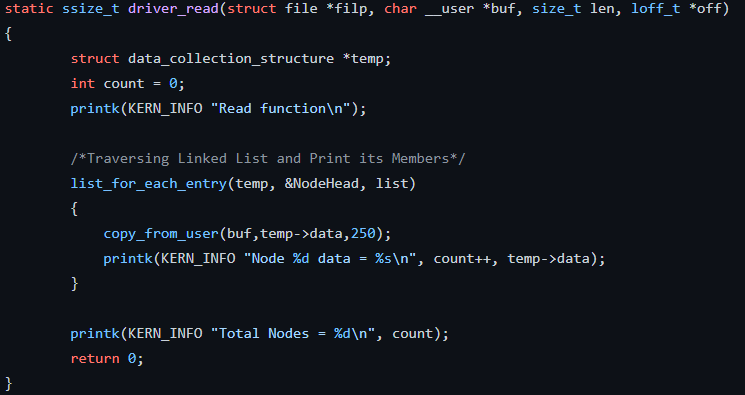




Dev\_t variables are used in the function alloc\_chrdev\_region() function and can be used to access Major and Minor Number values:

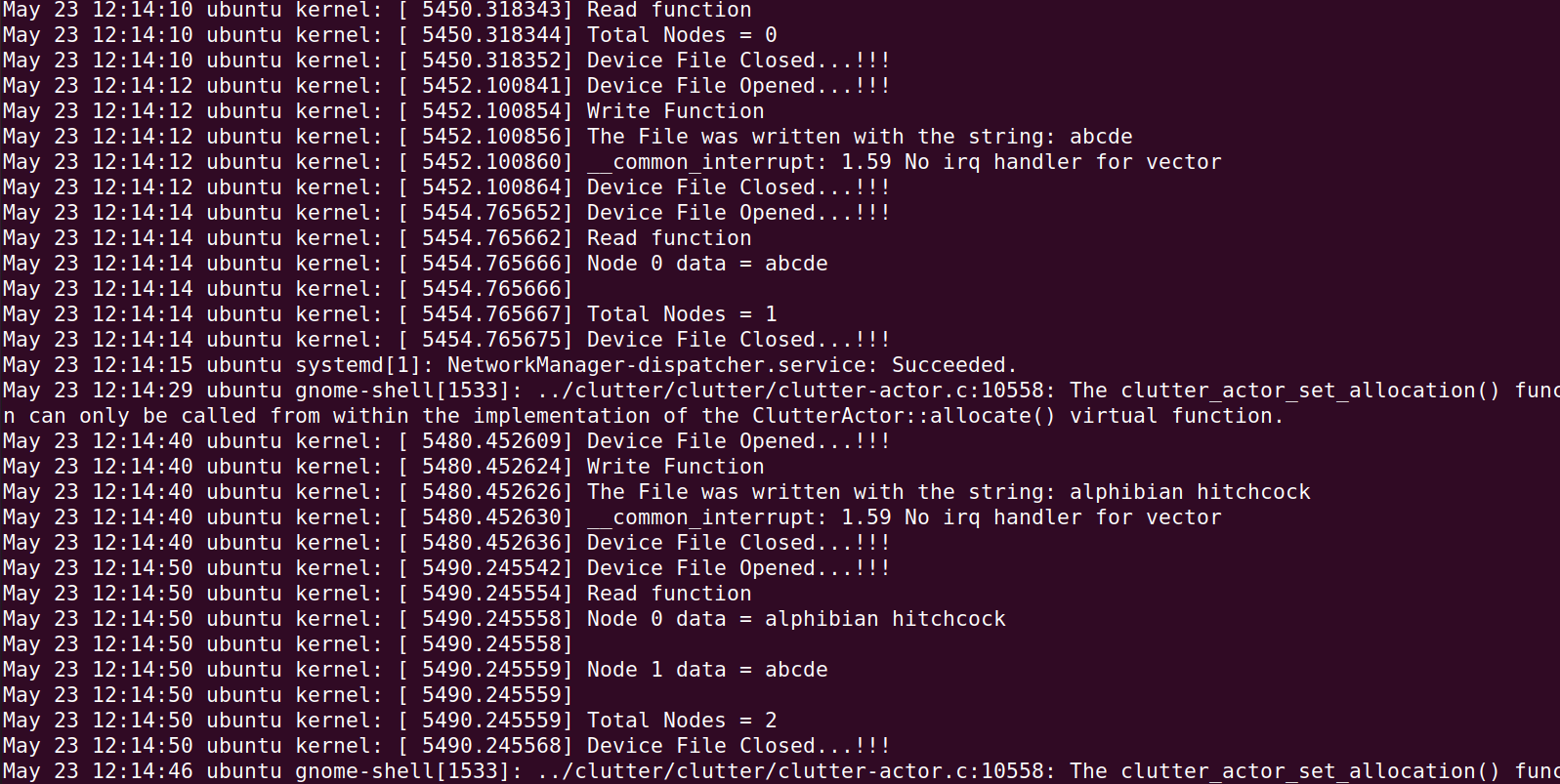
**Project minor code explanation in relation to memory:**  


When the driver\_write function is invoked (for this particular example the **echo [insert string here] > /dev/etx\_device** command will call it), the function produces a temp structure pointer of the structure type data\_collection\_structure, this copies the data from the buffer and using the **list\_add()** kernel defined function, prepends the data in the list.



When the driver read function is invoked (for this particular example **cat /dev/etx\_device** command will call it), the function produces a temp structure pointer of the structure type data\_collection\_structure, and then with the function for list **list\_for\_each\_entry()** it traverses the list and prints all the nodes.

Using the **sudo tail -f /var/log/syslog** terminal command we can see the actions being performed in kernel space in real-time:



# Heap Memory in Assembly:

Heap Memory allocation here is done to first of all, show windows related memory allocation (as the heap allocator is described in the windows library), secondly it explicitly showcases the extensivity of the Assembler and how memory can be allocated in low-level languages. The linked list implementation of the Heap Allocator in assembly showcases its strength in parsing memory.

Things to remember:

**GetProcessHeap** returns a 32-bit integer handle to the program’s heap area. Save this

handle and use it when calling other memory-related functions. Using this function, you can

request (allocate) memory without having to create your own heap.

**HeapAlloc** returns the address of block of memory from an existing heap, identified by a

heap handle. The allocated memory cannot be moved. If the memory cannot be allocated, the

function returns NULL (0).

**HeapFree** frees a block of memory previously allocated from a heap, identified by its

address and heap handle. If the block is freed successfully, the return value is nonzero. If the

block cannot be freed, the function returns zero and you can call the GetLastError API function

to get more information about the error.

Note: GetProcessHeap allows you to allocate memory from the process heap without invoking **HeapCreate** first (by invoking HeapAlloc).

Example: .data

hHeapProc DWORD ?

dwBytes DWORD 1000 ;bytes to allocate

hHeapBlock DWORD ?

.code

INVOKE GetProcessHeap

mov hHeapProc,eax ;save handle to process heap

INVOKE HeapAlloc,

hHeapProc, ;handle to process heap

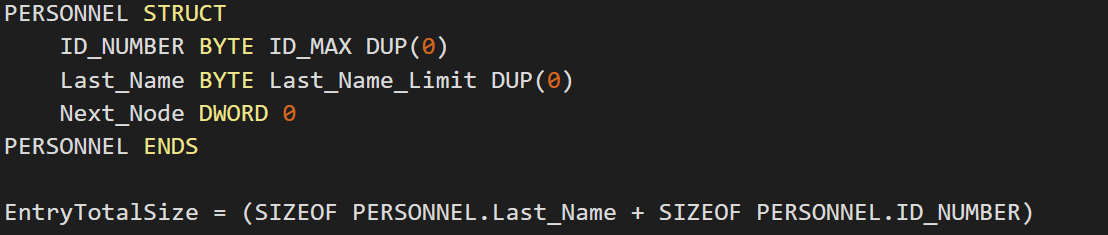
0,

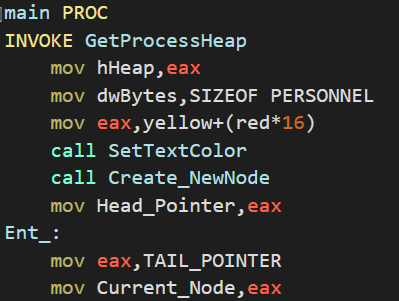
dwBytes

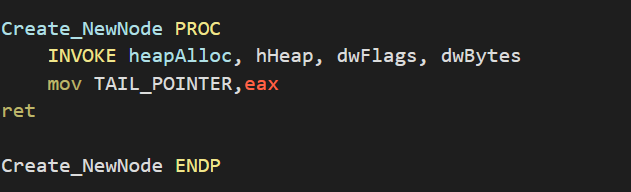
mov hHeapBlock,eax ;save handle to allocated block from heap

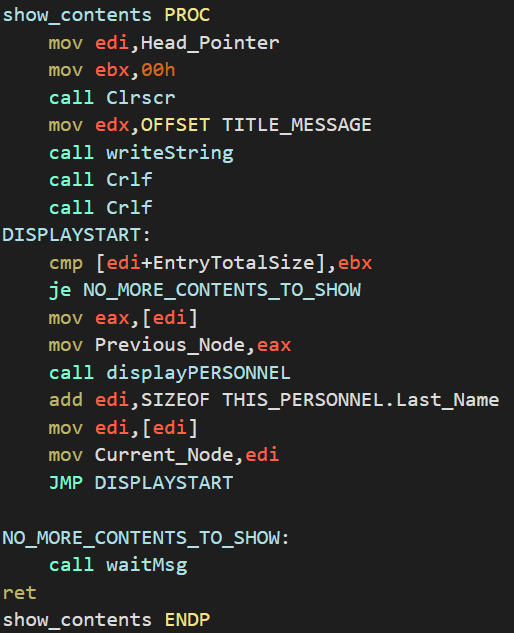
Call GetProcess heap to request memory to allocate memory without having to call your own heap.

Set dwBytes with size of the PERSONNEL structure, make a rootpointer with Create\_NewNode, set the tail pointer.

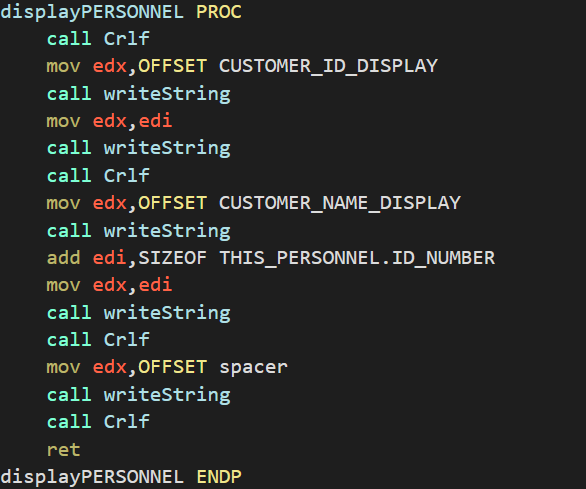
Structure Definition:  


****

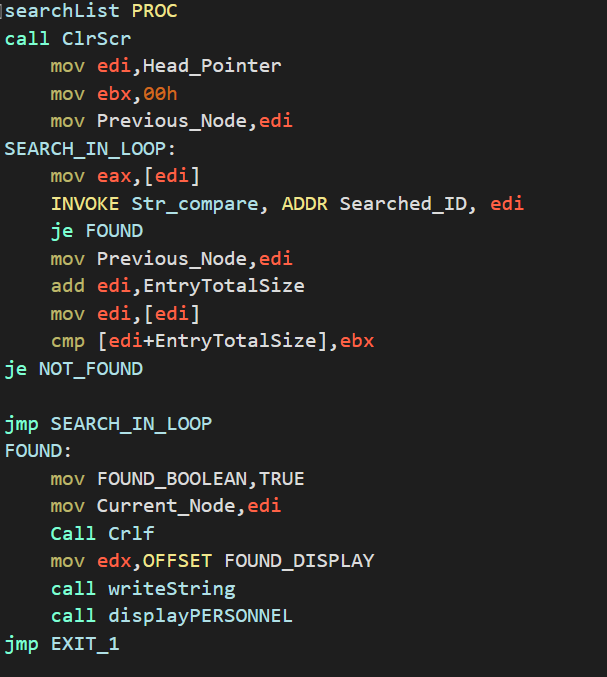
Creation of New Node:  


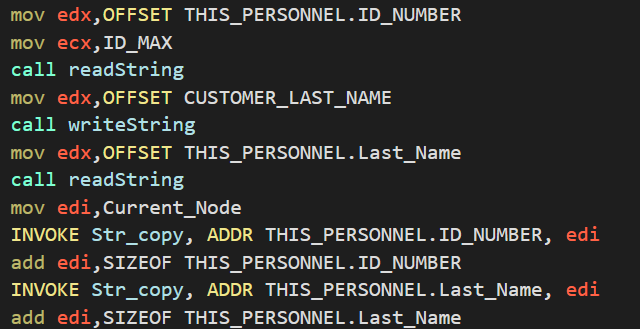
**How to show all records?**  
Move the head pointer into edi, now to get to each record sequentially, add the EntryTotalSize DWORD variable to edi to get to every successive entry, call displayPERSONNEL after every entry traversed. ****

In displayPERSONNEL, the first entry is the ID number, simply move the edi into edx and print the contents, when you want to print the name, simple add the offset inside edi with **SIZEOF THIS\_PERSONNEL.ID\_NUMBER,** to get the name entry inside the record.



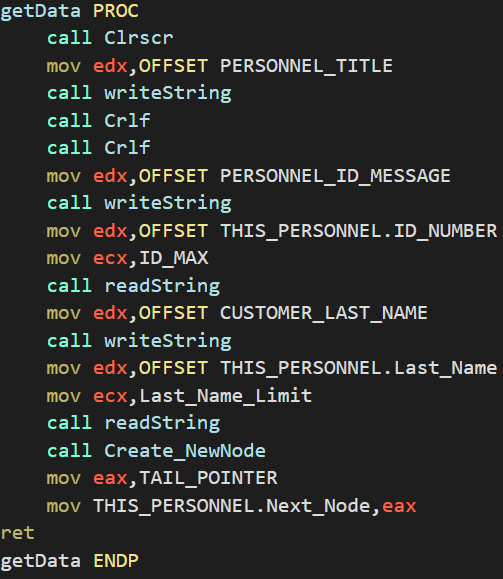
Similar to displaying the records, we can mold this function to search the records, all that’s different now is that during traversal we compare our searching ID with the contents in edi (remember, whenever we add EntryTotalSize with edi, the first variable inside the record is ID, hence it is the first thing compared)

****

For the updating of a particular record, simply input the values inside the record THIS\_PERSONNEL, then copy the contents of this structure into our offset of the record that we want to update (contained in edi), here’s a snippet of code to emphasize this:  
****

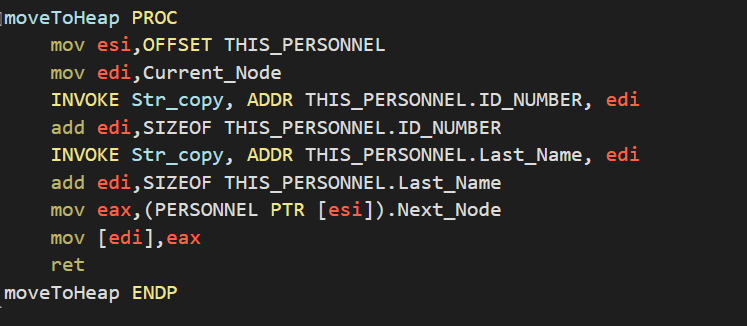
**How is a fresh new node created?**

Simply read the information in the previously aforementioned THIS\_PERSONNEL structure, this time call the Create\_NewNode which will put the value of a pointer pointing to a new allocated space of memory in the heap, move this value into the tail pointer, have the THIS\_PERSONNEL structure point to this new allocated space of memory inside the heap.



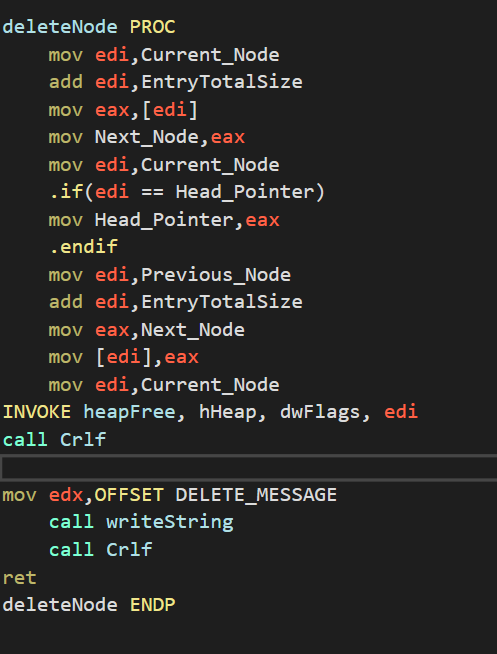
**How is Current\_Node being updated when we create a new node?**

Once you’re done with the creation of a new node, copy the values of this newly made node inside the current node (previously updated), traverse all record entries one by one (edi + sizeof THIS\_PERSONNEL.ID\_NUMBER + sizeof THIS\_PERSONNEL.Last\_Name to get to the Next Node DWORD variable inside the record, update the current node to point to this)

****

**How is node deleted?**If the current node == pointer, set the head pointer to equal to the next pointer,

Move the value after Previous\_Node into edi, and then have this value point to the Next\_Node, and then point the value inside edi to the current node, and then free the memory allocated in this register.

****