SUMMER INTERNSHIP PROJECT REPORT

Submitted by

APURVA.S

Roll no.108117013

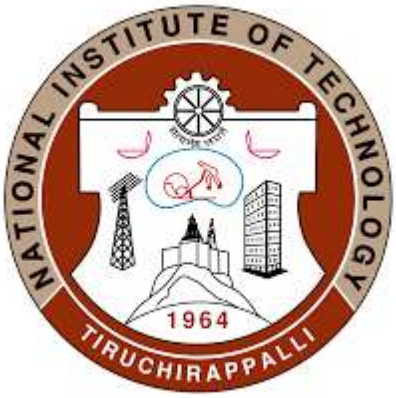
In fulfillment of Summer Internship for the award of the degree

Of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS AND COMMUNICATION ENGINEERING



NIT TIRUCHIRAPPALLI

National Institute of Technology (NIT-T)

Tiruchirappalli, Tamil Nadu

1. Acknowledgement

I wish to take this opportunity to express my deep gratitude to all those who helped, encouraged, motivated and have extended their cooperation in various ways during the project work. It is my pleasure to acknowledge the help of all those individuals who was responsible for foreseeing the successful completion of the project.

I express my profound gratitude to **Mr. MOHAMMED RAFIQ KAMAL,** Director, Foundry Software at Samsung Semiconductor India R&D center (SSIR), Bengaluru for providing me with this opportunity to carry out the project in Foundry multimedia SW team.

I would like to thank my mentor **Mr.Shaik Ameer Basha**, senior chief engineer, multimedia team, Foundry team SSIR-Bengaluru for guiding me in every step to carry out the project work.

I would like to thank the management of SSIR, Bengaluru for giving me this opportunity.

Thank You.

Apurva.S

2. Certificate from the Organization

SAMSUNG

Samsung Semiconductor India Research (SSIR)

30-Jun-2020

Experience Certificate

This is to certify that Ms. **Apurva S** (GEN ID: **20512103**) has interned with Samsung R&D Institute India - Bangalore Pvt. Ltd., from **May 07, 2020** to **June 30, 2020**.

India - Bangalore

For Samsung R&D Institute Pvt. Ltd.

Madhu Mohan Mulbagal Rat iah
Associate Director
Human Resources Dept.

INDEX

3. Introduction about the Organization:

Samsung

The Samsung Group is a South Korean multinational conglomerate headquartered in Samsung Town, Seoul. The company focuses on four areas: digital media, semiconductor, telecommunication network and LCD digital appliances. Samsung entered the electronics industry in the late 1960s.

Samsung Electric Industries is the world's largest manufacturer of consumer electronics by revenue. It is also the world's largest memory chip manufacturer and, from 2017 to 2018, had been the largest semiconductor company in the world, briefly dethroning Intel, the decades-long champion. As of 2019, Samsung Electronics is the world's second largest technology company. The semiconductor - business area includes semiconductor chips such as SDRAM, SRAM, NAND flash memory; Smart cards; mobile application processors; mobile TV receivers; RF transceivers; CMOS Image sensors, Smart Card IC, MP3 IC, DVD/Blu-ray Disc/HD DVD Player SOC and multi-chip package.

Samsung R&D Bengaluru

Samsung's first research and development (R&D) centre in the country was set up in Bengaluru in 1996. SRI - Bengaluru is also the largest R&D centre of the South Korean conglomerate outside its home country, South Korea. It is involved in manufacture of various products like:

- LCD and LED panels- OLED , AMOLED
- Mobile phones
- Semiconductors-NODES, MOSFET transistors, IC chips, SDRAM, SGRAM, TLC NAND Flash memory, GDDR6, LPDDR5
- Televisions

Teams and roles:

- There are mainly three divisions in the company :
  - Memory Solutions
  - Foundry
  - System LSI
- Each division has both Software and Hardware subdivisions and multiple sub teams.

4. Training Schedule:

I interned at Samsung Semiconductor Institute of Research, Bengaluru for a duration of two months beginning from 7th May 2020 to 30th June 2020. Five days per week work schedule was followed. In a week, minimum of 40 hours is required to complete the assigned tasks. The usual work timing followed was 10 AM to 6 PM.

The first two weeks was spent on learning the basics of kernel programming, device drivers and memory management. Next six weeks were dedicated to build on concepts related to project and finally implement the project. Also in the final week, the project was presented to the team and it was documented.

5. Work done/Observations:

I worked on building a memory allocator with buffer sharing mechanism at the Kernel level using kernel module programming and DMA-Buf sharing API. It was observed that this memory allocation method is an improvement over the currently prevailing memory allocator- “Android ion driver” for better performance in situations where the user requests smaller memory.

6. Specific Assignment/Project:

I was part of multimedia Sub-team from Foundry Software Division. I was assigned a project titled - ***Contiguous Memory allocator with DMA-Buf implementation***

6.1 Abstract:

DMA –Buf API is a framework used to allocate buffers and to share the allocated DMA buffer amongst multiple drivers in the kernel space. It provides uniform APIs that allows various operations related to buffer sharing. This framework also helps in sending back the buffer to user space on request by the user.

For example, shared buffers are used in decoding video stream into buffers suitable for graphics rendering and display. Camera captures into buffers suitable for encoding and rendering. In these cases DMA- Buf API framework is used.

The advantage of this framework is that the drivers need not know how and where the buffer is allocated but can still utilize its functionalities.

Therefore, the engineers working on drivers and other hardware tools can access these shared buffers without prior knowledge of how the buffer is allocated in the memory with the help of BMA-Buf API.

6.2 Problem statement:

The buffer sharing mechanism is not uniform. At present, frameworks and sub-systems have their own ways of sharing buffers. Thus, no uniform sharing APIs are available as it varies form one device to another. For Example, Video for Linux (V4L2) has a 'USERPTR' mechanism, which requires user space mmap if the underlying memory is coming from some other device. So a user space mmap is required which might keep oscillating between kernel space and user space, thus not allowing migrations.

6.3 Solution:

In order to overcome the above problem, a generic sharing mechanism is proposed. This can be implemented by DMA-Buf API. It is a generic kernel level framework to share the buffers. After declaring a DMA-Buf object, the exporter is used to share it among multiple devices. Thus, It provides a uniform API for buffer sharing related functions and solves the above mentioned problem.

6.4 Project Requirements:

Software:

- Coding language: C
- Operating System: Linux

7. Introduction about the project:

Linux is an open source operating system. Linux consists of various software components that manage computer hardware resources.

A kernel is the lowest level of software that interfaces with the hardware in your computer. The Linux kernel is a fundamental part of Linux and contains millions lines of code. The kernel is the core of the operating system because it manages communications between software and hardware components. The Linux kernel handles the lowest level of abstraction, or the most complex and specific data.

7.1 Kernel Module:

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, device driver is a type of module, which allows the kernel to access hardware connected to the system.

Every kernel module program has these two functions compulsorily - ***init_module()*** and ***cleanup_module().*** init_module() either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code. The cleanup_module() function is supposed to undo whatever init_module() did, so the module can be unloaded safely. init_module() is called when the module is inserted into the kernel using insmod, and the cleanup_module() which is called just before it is removed from the kernel. Also, Kernel modules need to be compiled a bit differently from regular user space apps.

User space and Kernel space:

Memory is divided into two distinct areas:

- The **user space**, which is a set of locations where normal user processes run (i.e. everything other than the kernel). The role of the kernel is to manage applications running in this space from messing with each other, and the machine.
- The **kernel space**, which is the location where the code of the kernel is stored, and executes under.

Processes running under the user space have access only to a limited part of memory, whereas the kernel has access to all of the memory. Processes running in user space also don't have access to the kernel space. User space processes can only access a small part of the kernel via an interface exposed by the kernel -the **system calls.** If a process performs a system call, a software interrupt is sent to the kernel, which then dispatches the appropriate interrupt handler and continues its work after the handler has finished. Kernel space code has the property to run in "kernel mode" Ring 0 (kernel space) is the most privileged ring, and has access to all of the machine's instructions.

7.2 Device Drivers

Device driver is a group of files that enable one or more hardware devices to communicate with the computer's operating system. The Linux kernel device drivers are essentially, a shared library of privileged, memory resident, and low level hardware handling routines.

Linux supports three types of hardware device: character, block and network. Character devices are read and written directly without

buffering. Block devices can only be written to and read from in multiples of the block size, typically 512 or 1024 bytes. Block devices are accessed via the buffer cache and may be randomly accessed, that is to say, any block can be read or written no matter where it is on the device.

The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. The minor number is used by the driver to distinguish between the various hardware it controls.

I have implemented a character device driver which takes input from user space and returns back the executed value to user space from kernel space .The user has to enter a string as input and the character driver returns the count of Upper case character letters.

Files required are: - chardev file, Test file, Makefile

This init module format is same as the one used in my main project. It is possible to check whether the registering of the device and class was successful or not by checking the log file.

Test file is the one that will be executed by the user. It is also the file which interacts with the user. Log file is also used for debugging the code.

Snippets from the implementation of the driver

```
static struct file_operations fops =
{
 .open = dev_open,
 .read = dev_read,
 .write = dev_write,
 .release = dev_release,
};

 int init_module(void){
 printk(KERN_INFO "Initializing the LKM\n");
 majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
 if (majorNumber<0){
 printk(KERN_ALERT "Characdrv failed to register a major number\n");
 return majorNumber;
 }
 printk(KERN_INFO "Registered correctly with major number %d\n", majorNumber);

// Register the device class
 characdrvClass = class_create(THIS_MODULE, CLASS_NAME);
 if (IS_ERR(characdrvClass)){
 unregister_chrdev(majorNumber, DEVICE_NAME);
 printk(KERN_ALERT "Failed to register device class\n");
 return PTR_ERR(characdrvClass);
 }
 printk(KERN_INFO " Device class registered correctly\n");

 characdrvDevice = device_create(characdrvClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
 if (IS_ERR(characdrvDevice)){
 class_destroy(characdrvClass);
 unregister_chrdev(majorNumber, DEVICE_NAME);
 printk(KERN_ALERT "Failed to create the device\n");
 return PTR_ERR(characdrvDevice);
 }
 printk(KERN_INFO "Device class created correctly\n");
```

Snippets from the implementation of the driver

```
static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset){
int error_count = 0;
  int upcasecount=0;
  while(len  && *msg_ptr){
  printk(KERN_INFO "VALUE OF MESSAGE PTR IS: %c ",*(msg_ptr));

  if( (*(msg_ptr)>='A') && (*(msg_ptr)<='Z') ){
     upcasecount=upcasecount+1;
          }
     printk(KERN_INFO "NO. OF UPPERCASE CHARCS TILL NOW :%d",upcasecount);
     put_user(*(msg_ptr++),buffer++);
     len--;
     }

sprintf(message, "%d", upcasecount);
  size_of_message = strlen(message);
   printk(KERN_INFO "VALUE OF MESSAGE (count) IS: %s ",message);
  error_count = copy_to_user(buffer, message, size_of_message); // Sending the result to user

  if (error_count==0){            // if true then have success
     printk(KERN_INFO "Sent result to the user\n");
     return (size_of_message=0);  // clear the position to the start and return 0
  }
  else {
     printk(KERN_INFO "Failed to send to the user\n");
     return -EFAULT;
  }

}
```

Snippet from test file

```
#define BUFFER_LENGTH 256
static char receive[BUFFER_LENGTH];     // The receive buffer from the LKM

int main(){
   int ret, fd, i = 0;
   char stringToSend[BUFFER_LENGTH];
   printf("Starting device test code example...\n");
   fd = open("/dev/characdrv", O_RDWR);
   if (fd < 0){
      perror("Failed to open the device...");
      return errno;
   }
   printf("Type in a short string to send to the kernel module:\n");
 scanf("%[^\n]%*c", stringToSend);
 printf("Writing message to the device [%s].\n", stringToSend);
 ret = write(fd, stringToSend, strlen(stringToSend));
 if (ret < 0){
 perror("Failed to write the message to the device.");
 return errno;
 }

 printf("Press ENTER to read back from the device...\n");
 getchar();

printf("Reading from the device...\n");
 ret = read(fd, receive, BUFFER_LENGTH);
 if (ret < 0){
 perror("Failed to read the message from the device.");
 return errno;
 }
 printf("The number of UPPER CASE letters are: \t %s \n", receive);
 printf("End of the program\n");
 return 0;
}
```

Execution of the driver

```
apurva@apurva-VirtualBox:~/Downloads$ make
make -C /lib/modules/5.4.0-29-generic/build/ M=/home/apurva/Downloads modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-29-generic'
  CC [M]  /home/apurva/Downloads/mychardev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/apurva/Downloads/mychardev.mod.o
  LD [M]  /home/apurva/Downloads/mychardev.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-29-generic'
cc testmychardev.c -o test
```

```
apurva@apurva-VirtualBox:~/Downloads$ sudo insmod mychardev.ko
```

```
apurva@apurva-VirtualBox:~/Downloads$ sudo ./test
Starting device test code example...
Type in a short string to send to the kernel module:
Hello Everyone.
Writing message to the device [Hello Everyone.].
Press ENTER to read back from the device...

Reading from the device...
The number of UPPER CASE letters are:    Hello Everyone.2
End of the program
```

7.3 Memory allocation methods

Kmalloc:

void * kmalloc(size_t size, int flags)

The kmalloc() function is a simple interface for obtaining kernel memory in byte-sized chunks. The kmalloc() function's operation is very similar to that of user-space's familiar malloc() routine, with the exception of the addition of a flags parameter. It takes 2 arguments, size required and flags. The function returns a pointer to a region of memory that is at least ***size*** bytes in length. The region of memory allocated is physically contiguous. On error, it returns NULL.

The kernel manages the system's physical memory, which is available only in page-sized chunks. Linux handles memory allocation by creating a set of pools of memory objects of fixed sizes. Thus, the kernel can allocate only certain predefined, fixed-size byte arrays. So if the asked

amount is not a multiple of page size, then the memory allocated will be more than the requested size. It also has restriction on maximum size that can be allocated.

The most commonly used flag, ***GFP_KERNEL***, means that the allocation is performed on behalf of a process running in kernel space. Using GFP_KERNEL means that kmalloc can put the current process to sleep waiting for a page when called in low-memory situations.

Other flags include ***GFP_ATOMIC,GFP_HIGHUSER,GFP_USER*** .

Zone modifiers: In addition to these if the following flags are used the allocation changes (they tell the allocator where and how to allocate memory*):*

_ _*GFP_DMA* : This flag requests allocation to happen in the DMA-capable memory zone. The exact meaning is platform-dependent and is explained in the following section.

_ _*GFP_HIGHMEM*: This flag indicates that the allocated memory may be located in high memory.

Vmalloc:

void *vmalloc(unsigned long size);

It allocates a contiguous memory region in the virtual address space. Although the pages are not consecutive in physical memory, the kernel sees them as a contiguous range of addresses. Vmalloc returns NULL address if an error occurs, otherwise, it returns a pointer to a linear memory area of size at least **Size**. Vmalloc is generally not used as it is less efficient than kmalloc.

Memory Zones

Memory in kernel can be divided into three zones: DMA-capable memory, normal memory, and high memory. While allocation normally happens in the normal zone, setting either of the bits just mentioned requires memory to be allocated from a different zone.

DMA-capable memory is memory that lives in a preferential address range, where peripherals can perform DMA access. On most platforms, all memory lives in this zone.

High memory is a mechanism used to allow access to large amounts of memory on 32-bit platforms.

Whenever a new page is allocated to fulfil a memory allocation request, the kernel builds a list of zones that can be used in the search. If _ _GFP_DMA is specified, only the DMA zone is searched: if no memory is available at low addresses, allocation fails. If no special flag is present, both normal and DMA memory are searched; if _ _GFP_HIGHMEM is set, all three zones are used to search a free page.

7.4 Direct memory Access:

Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main system memory (random-access memory) independent of the central processing unit (CPU). With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller (DMAC) when the operation is done. This feature is useful at any time that the CPU cannot keep up with the

rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer.

DMA can offload expensive memory operations, such as large copies or scatter-gather operations, from the CPU to a dedicated DMA engine.

A DMA controller can generate memory addresses and initiate memory read or write cycles. It contains several hardware registers that can be written and read by the CPU.

7.5 DMA buffer sharing:

DMA Buffer sharing has lot of applications in multimedia section. As previously mentioned, decoding video stream into buffers suitable for graphics rendering and display. Camera capture into buffers suitable for encoding and rendering.

But, a uniform mechanism to share DMA buffers across different devices and sub-systems does not exist. However, there are lot of problems associated with the present methods used such as in V4L2, it requires a user space mmap which might oscillate between kernel space and user space.

Thus, DMA-BUF API is used for buffer sharing and Synchronization. It is a generic kernel level framework to share buffers for DMA access across multiple device drivers and subsystems.

The three main components:

- dma-buf, representing a sg_table, is exposed to user space as a file descriptor to allow passing between devices

- fence, which provides a mechanism to signal when one device as finished access
- reservation, which manages the shared or exclusive fence associated with the buffer.

Any device driver which wishes to be a part of DMA buffer sharing, can be either the 'exporter' of buffers, or the 'user' or 'importer' of buffers.

The exporter implements and manages all operations on the buffer, allows other users to share buffer using the dma_buf sharing APIs, manages the actual allocation of memory backing storage and takes care of any migration of scatterlist - for all shared users of this buffer.

The importer is one of the many users which are using the shared buffer. It does not know how and where the buffer is allocated. It only needs a functionality to get access to the scatterlist that makes up this buffer in memory, mapped into its own address space, so it can access the same area of memory.

steps involved :

- The exporter defines his exporter instance and calls dma_buf_export() to wrap a private buffer object into a dma_buf. It then exports that dma_buf to user space as a file descriptor by calling dma_buf_fd()
- Userspace passes this file-descriptors to all drivers it wants this buffer to share with: First the file descriptor is converted to a dma_buf using ***dma_buf_get().*** Then the buffer is attached to the device using ***dma_buf_attach()***
- Once the buffer is attached to all devices user space can initiate DMA access to the shared buffer. In the kernel this is done by

calling ***dma_buf_map_attachment()*** and ***dma_buf_unmap_attachment()***

- Once a driver is done with a shared buffer it needs to call ***dma_buf_detach()*** and then release the reference acquired with dma_buf_get by calling ***dma_buf_put().***

Structures used:

struct dma_buf

It represents the shared buffer. It is created by calling dma_buf_export() and is represented in user space as a file descriptor.

struct dma_buf_attachment

It is used to represent information related to user devices- buffer attachment.

Struct dma_buf_ops

It has information operations possible on struct dma_buf

Functions used:

dma_buf_export()

It is used to creates a new dma_buf, and associates an anon file with this buffer, so it can be exported. Connects the exporter's private metadata for the buffer, an implementation of buffer operations for this buffer, and flags for the associated file.

dma_buf_fd()

Function returns a file descriptor for the given dma_buf. It takes pointer to the dma_buf and flags as input parameter. User space then passes the FD to other devices / sub-systems participating in sharing this dma_buf object.

dma_buf_get()

It is used by importing device to get the dma_buf object associated with the FD. It takes FD of the dma_buf as input parameter and returns the dma_buf structure related to an FD.

dma_buf_attach()

It is called by the importer to attach itself to the dma_buf object. It is called at the beginning. It is an optional function.

dma_buf_detach()

It is used to remove attachment created by calling attach().It is called at the end. The importer informs tells the exporter that it no longer wants to access the shared buffer. It is an optional function but if attach is used this function is also used.

The following are compulsory callbacks required for dma_buf operations.

map_dma_buf ()

It is used to map a shared dma_buf into device address space, and it is mandatory. It can only be called if attach has been called successfully.

It returns a sg_table scatter list of the DMA buffer, already mapped into the device address space of the device attached with the provided dma_buf_attachment.

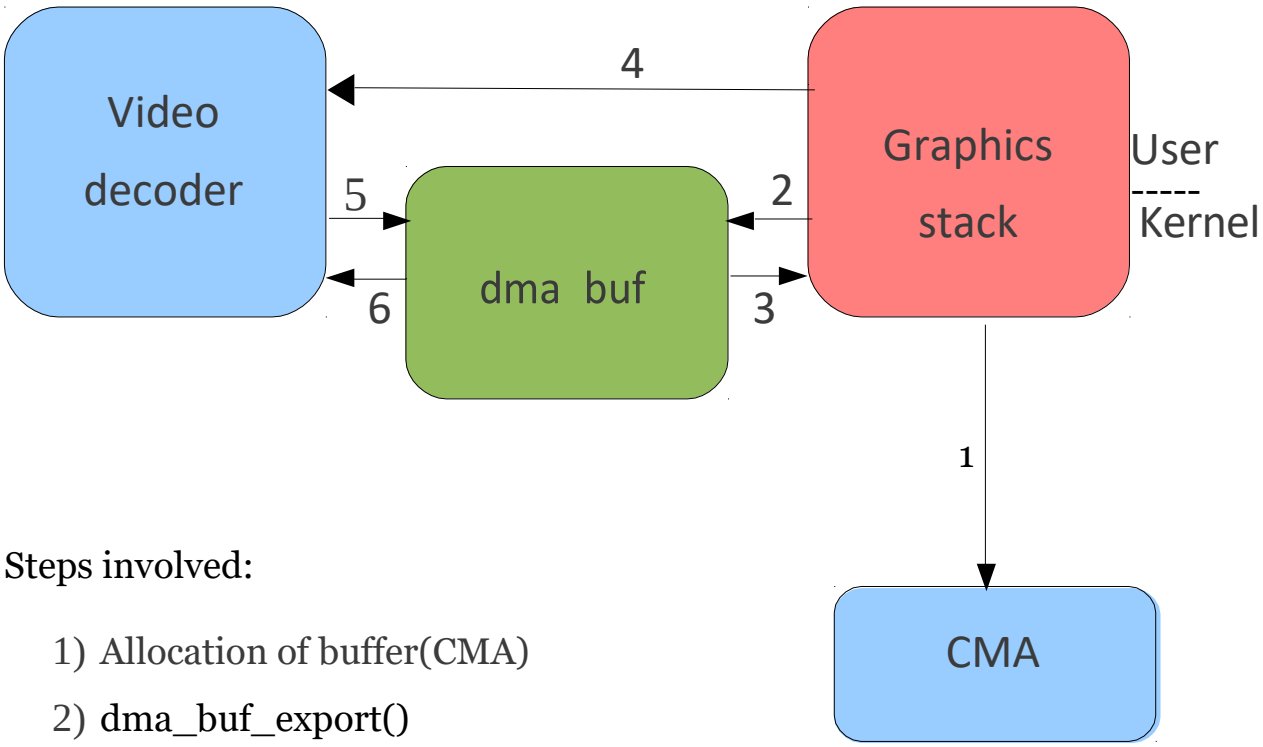
unmap_dma_buf()

It is used to unmap and release the sg_table allocated in map_dma_buf. It is used by importing device to indicate to exporter the end of DMA access. It is a mandatory function.

release ()

It is called after the last dma_buf_put to release the dma_buf. It is a mandatory function.

Example for DMA Buf sharing Usecase



Steps involved:

1) Allocation of buffer(CMA)
2) dma_buf_export()
3) dma_buf_fd()
4) fd passed to video decoder
5) dma_buf_get(fd)
6) dma_buf_attach()

8. Project implementation

I implemented the memory allocator by using two different modules for importer and exporter. One module had importer related functions and the other module had exporter related functions. Not all data members from the existing model were not used , the structures were customized to best fit the requirement. Also after every function executed the scatter gather list was updated. The project had the feature to take in input from the user to decide which of the following functions to be performed-

1. Allocate a Buffer
2. Buffer attach for an exporter
3. Write by exporter
4. Detach the buffer attached to exporter
5. Attach a buffer to importer
6. Read the contents by importer
7. Detach the buffer attached to importer
8. Read contents of the buffer from the user space

Depending on the option given by the user, corresponding module is called through **ioctl** calls.

For the first four functions exporter module is invoked and for the next three importer module is called. For the last function **mmap()** function is used to get address of the buffer location in the memory and later return the contents present at that address to user space.

Possible functions related to buffer in the project are (Buf ops) :

- Buffer allocation
- Buffer attach
- Write into the buffer
- Buffer detach
- Read from buffer

Depending on whether it is importer/exporter and which of the above options, corresponding ioctl calls is invoked.

Snippets from the importer module implementation of the project

```
static long my_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    struct dma_buf *dmabuf;
    struct fd_data fddata;
    void *kvaddr = NULL;

    switch (cmd) {

    case SDMABUF_ALLOC:
        pr_info("SDMABUF_ALLOC ----------------- \n");
        break;

    case SDMABUF_ATTACH:
        pr_info("SDMABUF_ATTACH ----------------- \n");
        copy_from_user(&fddata, (struct fd_data *)arg,
                       sizeof(struct fd_data));
        dmabuf = dma_buf_get(fddata.fd);

        cur_attach = dma_buf_attach(dmabuf, export_device);
        dma_buf_put(dmabuf);
        break;

    case SDMABUF_WRITE_MSG:
        pr_info("SDMABUF_WRITE_MSG ----------------- \n");
        break;

    case SDMABUF_READ_MSG:
        pr_info("SDMABUF_READ_MSG ----------------- \n");
        copy_from_user(&fddata, (struct fd_data *)arg,
                       sizeof(struct fd_data));
        dmabuf = dma_buf_get(fddata.fd);

        /* begin cpu access */
        dma_buf_begin_cpu_access(dmabuf, DMA_NONE);
```

Snippets from the importer module implementation of the project

```
        /* begin cpu access */
        dma_buf_begin_cpu_access(dmabuf, DMA_NONE);

        /* get kernel virtual address to write */
        kvaddr = dma_buf_kmap(dmabuf, 0);
        pr_err("Message Received in the Importer\n");
        pr_err("---------------------------------\n");
        pr_err(" Message: kvaddr: 0x%p, %s\n", kvaddr, (char *) kvaddr);
        pr_err("---------------------------------\n");

        dma_buf_kunmap(dmabuf, 0, kvaddr);

        dma_buf_end_cpu_access(dmabuf, DMA_NONE);
        dma_buf_put(dmabuf);
        break;

    case SDMABUF_DETACH:
        pr_info("SDMABUF_EXIT ---------------- \n");
        copy_from_user(&fddata, (struct fd_data *)arg,
                       sizeof(struct fd_data));
        dmabuf = dma_buf_get(fddata.fd);
        pr_info("dmabuf: 0x%p, fd: %d\n", dmabuf, fddata.fd);
        dma_buf_detach(dmabuf, cur_attach);
        dma_buf_put(dmabuf);
        break;

    default:
        return -EINVAL;
    }
    return 0;
}
```

Snippets from exporter module implementation

```
static int sdma_ops_mmap(struct dma_buf *dmabuf, struct vm_area_struct *vma)
{
    struct sdma_buffer *buffer = dmabuf->priv;
    struct sg_table *table = buffer->sgt;
    unsigned long addr = vma->vm_start;
    unsigned long offset = vma->vm_pgoff * PAGE_SIZE;
    struct scatterlist *sg;
    int i;
    int ret;

    mutex_lock(&buffer->lock);

    /* now map it to userspace */
    for_each_sg(table->sgl, sg, table->nents, i) {
        struct page *page = sg_page(sg);
        unsigned long remainder = vma->vm_end - addr;
        unsigned long len = sg->length;

        if (offset >= sg->length) {
            offset -= sg->length;
            continue;
        } else if (offset) {
            page += offset / PAGE_SIZE;
            len = sg->length - offset;
            offset = 0;
        }
        len = min(len, remainder);
        ret = remap_pfn_range(vma, addr, page_to_pfn(page), len,
                              vma->vm_page_prot);
        if (ret)
            break;

        addr += len;
```

Snippets from exporter module implementation

```
static const struct dma_buf_ops dmabuf_ops = {
    .attach     = sdma_ops_attach,
    .detach     = sdma_ops_detatch,

    /* Below are the mandatory callbacks. Checked inside dma_buf_export() */
    .map_dma_buf    = sdma_ops_map_dma_buf,
    .unmap_dma_buf  = sdma_ops_unmap_dma_buf,
    .release    = sdma_ops_release,

    .mmap       = sdma_ops_mmap,
    .map        = sdma_ops_kmap,
    .unmap      = sdma_ops_kunmap,
};

static void sma_free_pages(struct sdma_buffer *buffer)
{
    struct sg_table *sgt = buffer->sgt;
    struct scatterlist *sg;
    int i;

    for_each_sg(sgt->sgl, sg, sgt->nents, i)
        __free_pages(sg_page(sg), 0);

    sg_free_table(sgt);
    kfree(sgt);
}

static void sdma_free_dmabuf(struct sdma_buffer *buffer)
{
    pr_info("dmabuf in sdma_free_dmabuf: 0x%p\n", buffer);

    sma_free_pages(buffer);
    kfree(buffer);
```

Below are snapshots of building of the modules and inserting them into the kernel. Taking inputs from the user to perform necessary actions. Accessing log file for checking flow of control and how the buffer is allocated in the memory.

Final execution of the project snapshots.

```
apurva@apurva-VirtualBox:~/Desktop/final$ sudo -s
root@apurva-VirtualBox:/home/apurva/Desktop/final# make
make -C /lib/modules/5.4.0-37-generic/build/ M=/home/apurva/Desktop/final modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-37-generic'
  CC [M]  /home/apurva/Desktop/final/kernel_exporter.o
  CC [M]  /home/apurva/Desktop/final/kernel_importer.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC [M]  /home/apurva/Desktop/final/kernel_exporter.mod.o
  LD [M]  /home/apurva/Desktop/final/kernel_exporter.ko
  CC [M]  /home/apurva/Desktop/final/kernel_importer.mod.o
  LD [M]  /home/apurva/Desktop/final/kernel_importer.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-37-generic'
gcc user_sdma_test.c -o sdma_test
root@apurva-VirtualBox:/home/apurva/Desktop/final# insmod kernel_importer.ko
root@apurva-VirtualBox:/home/apurva/Desktop/final# insmod kernel_exporter.ko
root@apurva-VirtualBox:/home/apurva/Desktop/final# dmesg |tail -10
[ 1664.633837] ------------------------------------------------
[ 1664.633839] sdma_importer: Initializing ...
[ 1664.633844] sdma_importer: device registered with major number 240
[ 1664.633923] sdma_importer: Device created
[ 1664.633924] ------------------------------------------------
[ 1685.399116] ------------------------------------------------
[ 1685.399118] sdma_exporter: Initializing ...
[ 1685.399123] sdma_exporter: device registered with major number 239
[ 1685.403614] sdma_exporter: Device created
[ 1685.403615] ------------------------------------------------
```

```
root@apurva-VirtualBox:/home/apurva/Desktop/final# ./sdma_test
---------------MENU--------------
1. Allocate
2. Exporter Buf Attach
3. Exporter Write
4. Exporter Buf Detach
5. Importer Buf Attach
6. Importer Read Message
7. Importer Buf Detach
8. Userspace Read of the Buf content
0 and >= 9, Exit

Enter your choice: 1
fd = 5

Enter your choice: 2

Enter your choice: 5

Enter your choice: 3

Enter your choice: 8
String read from buffer: 1: Exporter is AWESOME!!!

Enter your choice: 6

Enter your choice: 7

Enter your choice: 4
```

```
root@apurva-VirtualBox:/home/apurva/Desktop/final# dmesg| tail -48
[ 1868.975934]  func: my_open, line: 449
[ 1868.975943]  func: my_open, line: 33
[ 1882.541309] SDMABUF_ALLOC -----------------
[ 1882.541314] sdma_buffer: 0x00000000953b9303
[ 1882.541325] sgt: 0x00000000d781bc15, entries: 5
[ 1882.541339] dmabuf in sdma_alloc: 0x000000002854402e
[ 1882.541340] allocated dmabuf: 0x000000002854402e
[ 1882.541342] fd in kernel == 5
[ 1887.953452] SDMABUF_ATTACH -----------------
[ 1887.953457] dmabuf: 0x000000002854402e, fd: 5
[ 1887.953459]  func: my_ioctl, line: 510
[ 1887.953462] sdma_buffer: 0x00000000953b9303
[ 1887.953466] buffer->sgt: 0x00000000d781bc15
[ 1887.953468] sgt: 0x00000000d781bc15, sgt->nents: 5
[ 1887.953470]  func: dup_sg_table, line: 89
[ 1887.953472]  func: dup_sg_table, line: 98
[ 1887.953473] a->sgt: 0x00000000d781bc15, a->sgt->nents:0x5
[ 1887.953474]  func: sdma_ops_attach, line: 142
[ 1887.953475]  func: sdma_ops_attach, line: 145
[ 1887.953477]  func: my_ioctl, line: 512
[ 1915.148154] SDMABUF_ATTACH -----------------
[ 1915.148161] sdma_buffer: 0x00000000953b9303
[ 1915.148166] buffer->sgt: 0x00000000d781bc15
[ 1915.148168] sgt: 0x00000000d781bc15, sgt->nents: 5
[ 1915.148171]  func: dup_sg_table, line: 89
[ 1915.148172]  func: dup_sg_table, line: 98
[ 1915.148174] a->sgt: 0x00000000d781bc15, a->sgt->nents:0x5
[ 1915.148175]  func: sdma_ops_attach, line: 142
[ 1915.148176]  func: sdma_ops_attach, line: 145
[ 1917.078684] SDMABUF_WRITE_MSG -----------------
[ 1917.078690] dmabuf: 0x000000002854402e, fd: 5
```

```
[ 1917.078690] dmabuf: 0x000000002854402e, fd: 5
[ 1917.078719] Message Written from the Exporter
[ 1917.078722] --------------------------------
[ 1917.078725]  Message: kvaddr: 0x0000000090d013cf, 1: Exporter is AWESOME!!!
[ 1917.078727] --------------------------------
[ 1954.700547] SDMABUF_READ_MSG ----------------
[ 1954.700551] Message Received in the Importer
[ 1954.700555] --------------------------------
[ 1954.700560]  Message: kvaddr: 0x0000000090d013cf, 1: Exporter is AWESOME!!!
[ 1954.700561] --------------------------------
[ 1960.923485] SDMABUF_EXIT ---------------
[ 1960.923490] dmabuf: 0x000000002854402e, fd: 5
[ 1967.838343] SDMABUF_EXIT ---------------
[ 1967.838347] dmabuf: 0x000000002854402e, fd: 5
[ 2010.261744]  func: my_close, line: 455
[ 2010.261753]  func: my_close, line: 39
[ 2010.261970] freed dmabuf: 0x000000002854402e
[ 2010.261972] dmabuf in sdma_free_dmabuf: 0x00000000953b9303
```

9.Outcome of the Project:

- Learnt Linux driver development and working of memory allocation
- Implemented two kernel modules to simulate dma-buf exporter and importer functionality.
- Implemented dma-buf operations for allocating and sharing memory buffers to user-space when requested in the exporter module.
- Importer will receive the file descriptor shared by the exporter for accessing the exporter allocated buffer.
- Implemented a user space program and ioctl to interface with both exporter and importer devices.
- Learnt the kernel programming style of coding.

10. Conclusion:

DMA_Buf sharing API was implemented in Linux using separate kernel modules for importer and exporter functionalities along with a test application module to interact with the user in the user space. It is observed that using this API provides a uniform method to share the buffers amongst multiple devices and to have comparatively more efficient memory allocation in the kernel space.