SYSCALL IMPLEMENTATION

Two syscalls are implemented namely kernel_device_reader and kernel_device_writer. The reader syscall dequeues data block from the kernel queue, and the writer syscall enqueues data to the kernel queue which is a thread-safe character device.

```
SYSCALL_DEFINE2(kernel_device_reader, char __user *, read_buffer, int,
len_buffer)
```

Reader syscall takes in a user-space char * and the size of the string which is fixed to 8bytes, the data block dequeued from the kernel queue, is stored in the char *, so consumer program 'c' could print the data block in userspace.
Reader syscall is called by the 'c' file.

```
SYSCALL_DEFINE2(kernel_device_writer, char __user *, write_buffer, int,
len_buffer)
```

Writer syscall takes in a user-space char * and the size of the string which is fixed to 8bytes, the data from this char* is enqueued to the kernel queue(character device). so producer program 'p' reads random 8-bytes of the device /dev/urandom and then passes this to the writer syscall, which enqueues this on the kernel queue.
Writer syscall is called by the 'p' executable file.

To open the kernel queue(as it is a character device). Filp open is used

```
struct file *filp_open(const char *, int, umode_t);
```

To close the character device filp_close is used.

```
int filp_close(struct file *, fl_owner_t id);
```

To read(dequeue) from the kernel queue, vfs_read is used.

```
ssize_t vfs_read(struct file *, char __user *, size_t, loff_t *);
```

To write(enqueue) to the kernel queue, vfs_write is used.

```
ssize_t vfs_write(struct file *, const char __user *, size_t, loff_t *)
```

All the errors generated during file opening, closing, reading and writing are checked and syscall returns -EFAULT and errno is set accordingly and the error is printed to stdout using perror.
If the syscall is successful 0 is returned.

Kernel log:
Also when the system call is used, data relevant to the syscall is also printed in the kernel logs. In case of any error, it is printed in the kernel log and the log can be accessed by dmesg | tail.

Kernel Queue

About the kernel queue:

A kernel level queue is used as a character device, to exchange strings between user-level processes. The device maintains a FIFO queue that can contain a configurable number of strings. Several concurrent user-level processes can read and write to the character device. So the kernel queue is made thread-safe so various user-level processes can read and write to the character device without any race conditions. This is achieved by the use of kernel synchronization primitives like kernel reader-writer semaphores. Some of them are:

```
struct semaphore {
    raw_spinlock_t        lock;
    unsigned int     count;
    struct list_head wait_list;
};

void sema_init(struct semaphore *sem, int val)
void up(struct semaphore *sem);
int down_interruptible(struct semaphore *sem)
```

Sema_init is used to initialize the semaphore,
Up is used to increment the value of the semaphore (count variable) is similar to sem_post()
Down_interruptible is used to decrement the value of the semaphore and is similar to sem_wait().

To install the character device(kernel queue):

1) **Do "cd new_device"**
2) **Do "make"**
3) Do "**sudo insmod new_device.ko buffer_size=<size>**" to install the module.

Now our kernel queue is installed by the name of “new_device” and can be looked up by the command “**lsmod”** which lists all the loaded modules and registration about the device is also printed in the kernel log.
The kernel queue (character device) is installed in /dev/new_device.
If the kernel fails to install and load, details are printed in the kernel log and can be checked by Dmesg | tail.

```
[kern@admin9 new_device]$ lsmod
Module                  Size  Used by
new_device             16384  0
vmwgfx                380928  1
intel_rapl_msr         20480  0
intel_rapl_common      28672  1 intel_rapl_msr
vmw_balloon            24576  0
crct10dif_pclmul       16384  1
crc32_pclmul           16384  0
```

The kernel implements the following functions,

```
    .open = &open_device,
    .read = &dequeue_data,
    .write = &enqueue_data,
    .release = &close_device
```

To remove the loaded module, do **“rmmod new_device”.**

Modified files:

1)arch/x86/entry/syscalls/syscall_64.tbl
2)kernel/sys.c
3) /dev/new_device

Diff files/ Patch file:

Diff.txt and Patch_file.patch is attached along with the code.
Commands used:
Git diff > diff.txt
Git diff > patch_file.patch

Producer.c and consumer.c

Makefile to compile both producer.c and consumer.c is provided. Do “make” which makes two executable files namely p and c.

To run these files do :

Sudo ./c first
then sudo ./p

The implemented syscalls and the kernel queue loaded can be tested using these files, which implements the producer-consumer problem.
All the errors are checked and errors are printed out by perror().

Producer.c reads random 8-bytes of the device /dev/urandom and passes them to the kernel via a system call, kernel_device_write. This system call enqueues the data sent by producer.c to the kernel queue.

And consumer.c calls the kernel_device_read system call which deques the data from the kernel and this block of data(string) is sent back to the consumer where this string is printed.