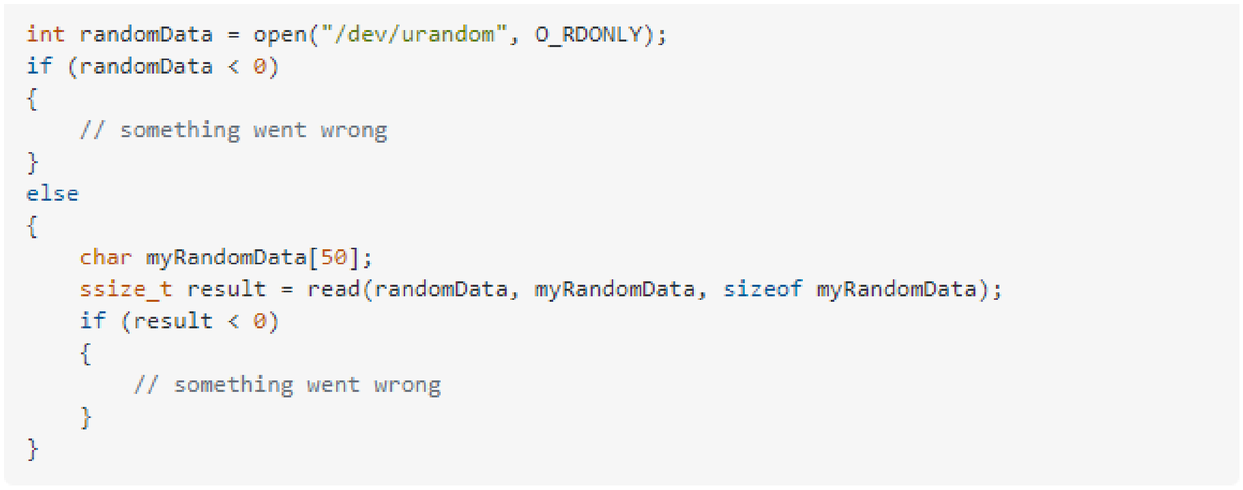
Writeup/Readme Assignment-4

To read the random data from the /dev/random use the following code



```c
int randomData = open("/dev/urandom", O_RDONLY);
if (randomData < 0)
{
    // something went wrong
}
else
{
    char myRandomData[50];
    ssize_t result = read(randomData, myRandomData, sizeof myRandomData);
    if (result < 0)
    {
        // something went wrong
    }
}
```

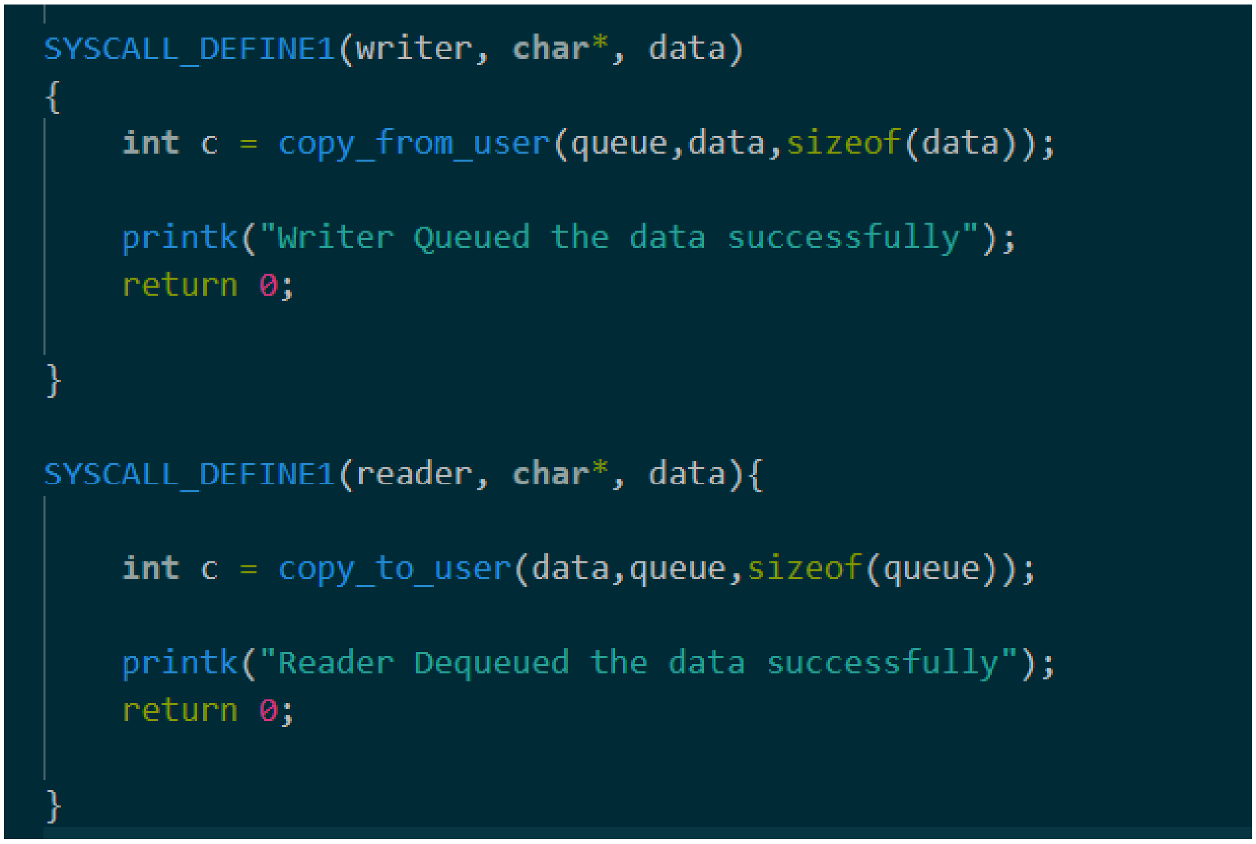
Steps followed for compiling the kernel and adding system call:
1. Download kernel source code using wget command wget

[https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.4.ta r.xz 2](https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.4.tar.xz).

Extract the tar.gz file using sudo tar -xvf linux-4.17.4.tar.xz -C/usr/src/

3. Goto to directory kernel/ in the extracted folder cd /usr/src/linux-4.17.4/

4. Edit file sys.c, ex : sudo nano sys.c

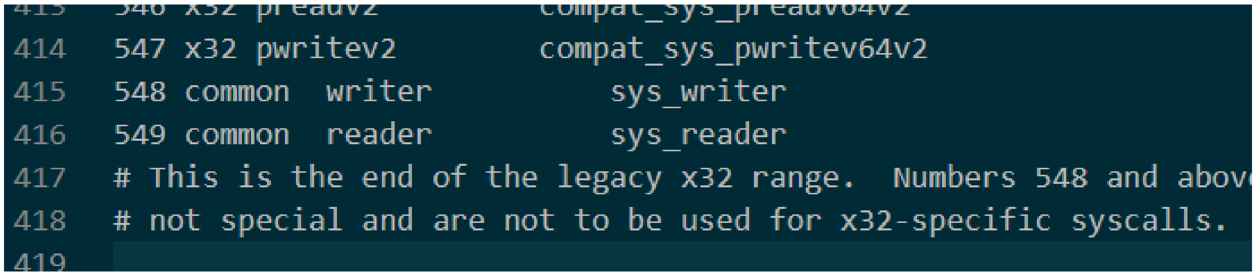


```c
SYSCALL_DEFINE1(writer, char*, data)
{
    int c = copy_from_user(queue,data,sizeof(data));

    printk("Writer Queued the data successfully");
    return 0;

}

SYSCALL_DEFINE1(reader, char*, data){

    int c = copy_to_user(data,queue,sizeof(queue));

    printk("Reader Dequeued the data successfully");
    return 0;

}
```

5. Add your syscall code in the file sys.c

6. Go to directory arch/x86/entry/syscalls/ in the extracted folder

7. Add your system call to file syscall_64.tbl Ex: 440 common sample sys_sample



```
546 x32 preadv2         compat_sys_preadv64v2
547 x32 pwritev2        compat_sys_pwritev64v2
548 common  writer          sys_writer
549 common  reader          sys_reader
# This is the end of the legacy x32 range.  Numbers 548 and abov
# not special and are not to be used for x32-specific syscalls.
```

kmalloc — allocate memory

Synopsis

```
void *           size_t
kmalloc (        size,

                 gfp_t
                 flags);
```

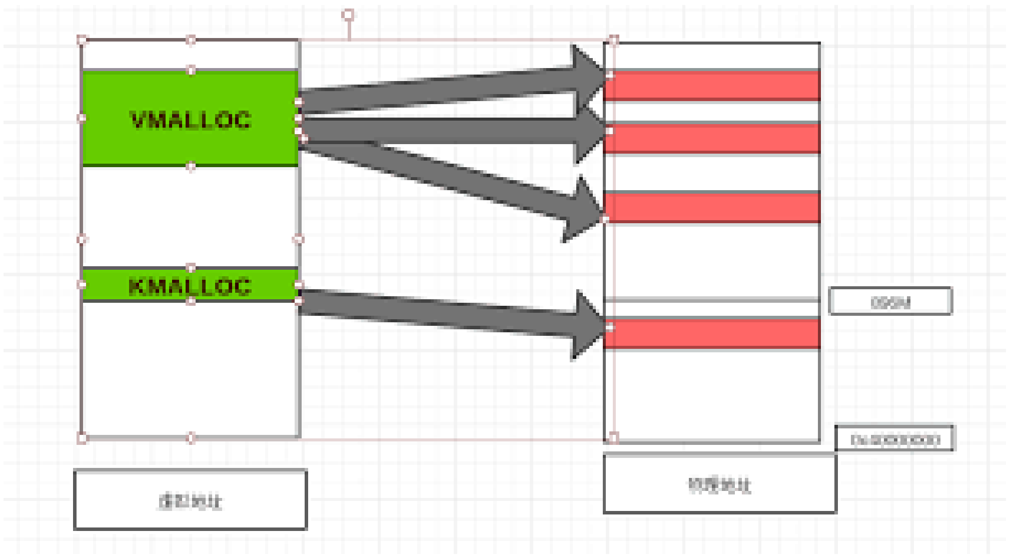
Arguments

size_t size

how many bytes of memory are required.

gfp_t flags

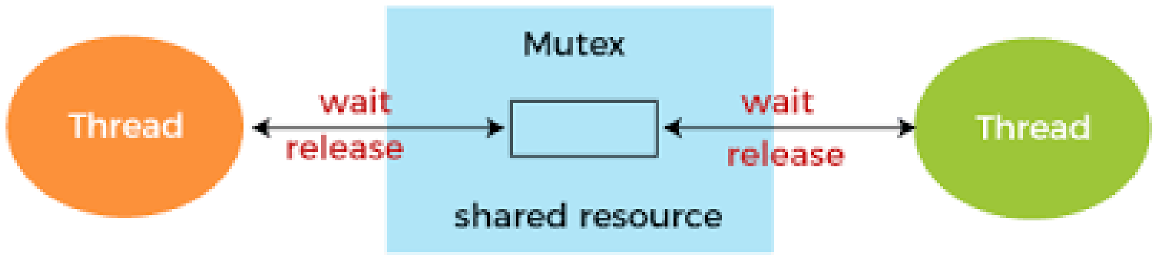
the type of memory to allocate.



Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.


Thread
wait
release
Mutex
shared resource
wait
release
Thread