# Design Document MP6  Operating Systems CSCE 611


## Chinmay Ajit Sawkar


## Attempted Bonus [1,2,3,4]

### Implementation

In this MP I have implemented a thread-safe blocking system with interrupts and only designed a master disk and dependent disk framework so that we can read from whichever disk is ready (master or dependent) and we write to both the master and the dependent disk.

### Design and Implementation of Blocking Disk [basic +bonus3+bonus4] :

The Blocking Disk class is implemented by inheriting from the Simple Disk. The blocking disk calls the read and write functions in a way that the disk operations are thread safe. This is done by calling the disk_lock() function before disk operation. This function sets the flag to locked(L) and keeps it locked until we call the disk_unlock() function. This is implemented by using the test and set technique which is implemented by the check_disk_locking function. Basically, in the lock_disk function the check_disk_locking() is called until it returns false and thus if another thread is using the disk another thread will get that the disk is in locked(L) state from check_disk_locking(). Only when the disk unlock is called any other thread can proceed its disk operation.

The scheduler files scheduler.C and scheduler.H from MP5 are used for scheduling the readyQ for threads.

The blocking disk has the following functions for ensuring thread safe disk operations:

1. check_disk_locking: This is used to implement the test and set technique for locking it checks if disk is unlocked (ie unlock_disk() is called or not) and returns the status locked or unlocked. It also sets the status to locked finally for the next check from lock_disk().
2. unlock_disk() : Sets the flag to unlocked (U)
3. lock_disk() - Used to lock the disk to make the disk operation thread safe.
4. flag_init() - to initialize the flag to an unlocked state.(U)

5. wait_until_ready(): this function checks if the disk is ready to read or write the data. In this I call the SimpleDisk class is_ready() function. If the disk is not ready I resume the job putting it in the end of the ready queue and yield the CPU. If the interrupts are enabled instead of sending it to the ready queue I add it to the disk blocking queue DQueue. Thus when we get an interrupt from the interrupt handler when the disk is ready the thread is popped and given access to the disk for the disk operation. This is further explained in interrupt concurrency.

6. read(): Used to call the simple disk read functions (in a thread safe manner between lock_disk and unlock_disk as it is the parent class.
7. write(): Used to call the simple disk write functions (in a thread safe manner between lock_disk and unlock_disk as it is the parent class.

**[Bonus2] Implementation of Interrupts for Concurrency.**

For concurrency the interrupts are enabled and then I have added a DQueue Class in blockingdisk.H file for implementing a disk queue in which I add thread to the disk blocking queue using the insert_thread() function of the Dequeue class object. When we get an interrupt from the interrupt handler when the disk is ready the thread is popped using the pop_thread function and given access to the disk for the disk operation. This is done in the handle_interrupt function. The thread is then resumed and added to the ready queue. This way if there are multiple jobs in the queue they get the CPU and the polling is reduced.

**[Bonus1] Designed the Support for Disk Mirroring**

** The implementation for this part is not done due to time constraints, just the design of the frame is given below.

For implementing the support for the Disk Mirroring we can implement a MirroredDisk Class (which is child class of Blocking Disk) where we write on both the disks and read from either of the disk which is ready. I define two objects MASTER_DISK and DEPENDENT_DISK of the Blocking Disk Class and for this we can define functions the functions below.

The Mirror_wait_until_ready check if both disk are not ready then sends the thread to ready queue or put in blocking queue(if interrupt enabled). If one of them is ready the function returns which one.

```
int Mirror_wait_until_ready()
{
        while (!MASTER_DISK->check_is_ready() &&
!DEPENDENT_DISK->check_is_ready())
        {
         Resume() thread or put in blocking queue(if interrupt enabled)
         yield()
        }
        if (MASTER_DISK->blocking_is_ready()){
                return 1
        }
        else{
                return 0
        }
}

//Used to check which of the disk is ready and run read function of parent class.
void Mirror_read(block_no,  buffer){
        if(wait_until_ready_mirr()){
                MASTER_DISK->read (block_no,  buffer);
        }
        else{
                DEPENDENT_DISK->read (_block_no, _buf);
        }
}

void Mirror_write(block_no, buffer){
        //writing to both the disks master and dependent
        MASTER_DISK->write (block_no, buffer );
        DEPENDENT_DISK->write (_block_no, _buf);
}
```

**Testing**

For testing the implementation I have written the buffer  to disk block 1 initially and initialized read_block as 1 and write block as 2. Thus the function 2 reads the buffer in block 1 and writes in block two. In the next iteration we read from block2 and write in block 3 and so on. Thus the buffer is carried forward as we read and write in the next block.

```
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
Writing in block 1 initially
***writing on disk on block 1 ***
FUN 2 IN ITERATION[0]
Calling Read operation from disk for read block1
disk is not yet ready:
THREAD: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
```

Thus when the disk is not ready the thread 2 is added to the disk blocking queue and cpu is yielded to thread  3, 4,1  as shown below. Meanwhile the disk is ready and interrupt adds thread 2 back to the ready queue. Thus it is read actually after thread 3,4,1 are executed.  Now the buffer is printed and the write disk operation is called and the disk is written.

```
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN ITERATION[1]
Calling Read operation from disk for read block2
disk is not yet ready:
FUN 3 IN BURST[2]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4 IN BURST[2]
```

```
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
***reading from disk on block1 ***
*buffer*
Calling write operation on disk for block2
***writing on disk on block 2 ***
FUN 3 IN BURST[1]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
```