CSCE 611: MP6: DESIGN DOCUMENT

AUTHOR: SRIVIDHYA BALAJI
UIN: 827007169

PROBLEM STATEMENT:

1) To implement Blocking Disk, where disks aren't waiting until the I/O is complete.
2) [BONUS 2 : DESIGNED] : Interrupt based Disk Handling
3) [BONUS 3: DESIGNED] : Thread safe system
4) [BONUS 4: IMPLMENTED]: Implemented a Thread Safe system.

1) [BASIC AND BONUS 3] DESIGN OF A BLOKCING DISK

In this while designing the basic functionality of the Blocking disk system, measures were taken to make it a Thread Safe system, thus simultaneously satisfying problem statement 1, 3, 4.

The following is the design for the same,

When a thread requests for an I/O operation, it is added to a separate queue called as the Blocking queue. When it is added to the Blocking queue, we yield the CPU to the next thread in the Ready Queue. A flag called IssueDiskOperationFlag is maintained, which checks whether a read or write operation is issued to the DISK. Initially this flag is set to false. In the Scheduler's yield, we check if this flag is still false and if the Blocking queue has an element. If so, we pop the element from the BLOCKING QUEUE and add it to he front of the Ready Queue. Now since it is in the front, FIFO order is maintained and this thread regains control at the point it lost the CPU. Now, we issue the operation to the disk and then add the thread to blocking queue and set the ISSUEOPERATION Flag to true and yield it to the next thread. The yield function now checks if the disk is ready, If it is ready, it pops the element from the Blocking queue and resumes it back to the Ready Queue. This is THREAD SAFE.

Let us consider two threads Thread 2, Thread 5 which perform disk Operations read and write. And remaining threads are Thread 1, Thread 3, Thread 4. The following occurs,

1) Thread 1 to Thread 5 are added to the READY QUEUE.
2) Thread 1 executes and passes CPU to THREAD 2.
3) Thread 2 requests for a read operation. The Blocking disk READ operation is called.
4) In that we call Wait Until READY which Adds THREAD 2 to Blocking queue and yields the CPU, if the disk is not ready.
5) In yield function, we check if IssueDiskOperationFlag is set. It is not set. So, the THREAD 2 is popped from the BLOCKING queue and is added to the front of the ready QUEUE.
6) Now THREAD 2 again regains back control at where it left. Now, we ISSUE READ OPERATION to the disk, set the IssueDiskOperationFlag as True and call waituntilready() , which would add THREAD 2 to Blocking queue and yield the CPU.
7) Now, in yield, we check if the flag is set. Since it is set, we next check if the Disk is ready. Let us assume the disk is not ready. So the CPU executes thread 3.
8) Thread 3 passes the CPU to thread 4, which in turn passes the CPU to thread 5.

9) Now THREAD 5 requests for an I/O read operation. Similarly it is added to the Blocking Queue and yields it to the next thread. In the yield function since IssueDiskOperationFlag is set as true, which indicates another thread has issued the disk operation, this disk 5 does not issue I/O operation and waits in the waiting thread. This is how, based on the flag IssueDiskOperationFlag, we ensure that only one thread is issuing disk I/O operation at a time. **Thus ensuring a thread safe system.**
10) So now, when disk becomes ready, thread 2 is popped from the blocking disk and is added back to the ready queue and IssueDiskOperationFlag is set to false, thus making it easier for Thread 5 to now issue a disk operation.

In this way, a thread safe system and the basic functionality of how Blocking disk does not wait for I/O return and yields the next thread is performed.

2) [BASIC, BONUS 4] COMBINED IMPLEMENTATION OF BASIC SCENARIO ENSURING THREAD SAFE SYSTEM:

I. BLOCKING DISK IMPLEMENTATION :

List of functions in Blocking disk is as below ,

1) BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size, Scheduler *_scheduler)

In this, the scheduler object is obtained from Kernel.C ie SYSTEM Scheduler object is obtained and initialized to the Scheduler object local to the class. And we also pass the current disk object to the Scheduler class so that it can access the disk functions.

2) void BlockingDisk::wait_until_ready()

Pseudo Code:

```
void BlockingDisk::wait_until_ready()
{
If(!isready()
    Add current thread to Blocking queue
    Scheduler->yield()
}
```

3) void BlockingDisk::pop_from_queue()

Pseudo Code:

```
void BlockingDisk::pop_from_queue()
{
    Thread* t = this->block_queue->dequeue();
        if(scheduler->IssueDiskOperationFlag == false) // if I/O not issued we pop and add to RQ.
                this->scheduler->enqueueFront(t);
        else
          this->scheduler->resume(t);  // or we resume it to the RQ. As explained above
```

```
            --queue_size;
}
```

4) **bool BlockingDisk::is_ready()**
Pseudo Code:

```
bool BlockingDisk::is_ready()
{
            /* This returns if the disk is ready or not.*/
return SimpleDisk::is_ready();
}
```

5) **void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf)**

Pseudo Code:

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf)
{
    Wait_until_ready() // Adding to the queue and yielding
    SimpleDisk::issue_operation(READ, _block_no);  // we issue the operation for THREAD SAFE
   scheduler->IssueDiskOperationFlag = true;  // Set flag once issued
   wait_until_ready();  // add to the blocking queue and wait till disk is ready

/*Perform READ* /
   int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++) {
    tmpw = Machine::inportw(0x1F0);
    _buf[i*2]   = (unsigned char)tmpw;
    _buf[i*2+1] = (unsigned char)(tmpw >> 8);
  }

/*reset the FLAG ONLY WHEN THE OPERATION IS COMPLETE. If reset, the next thread in the
blocking queue gets a chance to issue the I/O operation disk . THREAD SAFE SYSTEM*/
  scheduler->IssueDiskOperationFlag = false;
}
```

6) **void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf)**
same thread operations as read.

Psuedo Code:

```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf)
{
wait_until_ready();
 SimpleDisk::issue_operation(WRITE, _block_no);
 scheduler->IssueDiskOperationFlag = true;
wait_until_ready();

int i;
 unsigned short tmpw;
 for (i = 0; i < 256; i++) {
   tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
   Machine::outportw(0x1F0, tmpw);
 }

scheduler->IssueDiskOperationFlag = false;
}
```

II. <u>CHANGES IN SCHEDULER CLASS</u>

The following functions and variables were modified in the scheduler class.

1) **IssueDiskOperationFlag :** This is the variable which ensures only one thread is accessing the disk for issuing the operation and also to read and write. This is initially set to false.
2) **void Scheduler::yield() :**

   Few modifications were made to this function from the last MP.

   PSEUDO CODE:

```
void Scheduler::yield(){

if(!IssueDiskOperationFlag)

{

/*if  flag is not set and the BLOCKING QUEUE is not empty we pop and add the element to the front of the queue so that it can issue the operation                    .*/

disk->pop_from_queue();

}

Else if (disk !=NULL && disk->is_ready())

{

/* here we pop the thread from the blocking queue and add it at the back of the READY queue. This is done only when the disk is ready. */

        disk->pop_from_queue();

}

….

…otherwise the previous dispatch code.
```

}

3) **void Scheduler::addDiskobj(BlockingDisk *_disk) :** This receives the disk object and initializes it to the local object using which the disk functionalities are accessed.

   POSSIBLE IMPROVEMENT : One possible improvement here is to use a handler class which would register with the blocking disk class and inherit this handler class in scheduler in order to prevent the Blocking disk object being exposed directly.

4) **void Scheduler::enqueueFront(Thread *_thread)**
   This function adds the thread to the front of the readyQueue, so that it executes again and issues the operation. This is done in order to ensure FIFO operation, Without this, the threads do not execute in the order of 1,2,3,4 as currently.

III. <u>TESTCASES :</u>

The following test cases were considered.

1) Single thread issuing DISK operation. The FIFO order was maintained and disk operation was complete. This verified the problem statement 1
2) Two threads issuing DISK operations. FIFO order was maintained , and only one disk issued operations to the disk at a time, thus ensuring thread safety.

3. <u>[BONUS 2 : DESIGN] : USING INTERRUPTS FOR CONCURRENCY :</u>

Based on the disk configuration, it is observed that when the disk is ready, INTEERUPT 14 is being raised. This can be used in order to interrupt the threads rather than polling. This can be implemented in the following way,

1) Register Interrupt 14 with Interrupt Handler class
2) Handler is defined for this Interrupt 14, ie when this interrupt is raised, the list of operations that needs to be performed.
3) Since we know Interrupt 14 is raised once the disk is ready, in this handler function, we pop the thread which had issued the disk operation and add it to the ready queue.
4) In this way, we can prevent checking if the disk is ready every time and use interrupts.

4. <u>**LIST OF FILES INCLUDED :**</u>

1) **KERNEL.C :** This has additional code where the Blocking disk is created and initialized.
2) **BLOCKING DISK. C and BLOCKING DISK. H :** This contains the blocking disk implementation
3) **SIMPLE DISK . C AND SIMPLE DISK. H: Here, the issue function is made public as we are explicitly calling it in a specific order to maintain a THREAD SAFE SYSTEM. NO CHANGES were made to SIMPLE DISK.C**
4) **SCHEDULER.C AND SCHEDULER.H :** In this changes to the current yield function and few additional functions are added.