Machine Problem 6

Jicheng Lu, 525004048

1. Introduction

In Machine Problem 6, we investigate kernel-level device drivers on top of a simple programmed-I/O block device (i.e., programmed-I/O LBA disk controller). The given block device uses busy waiting to wait for I/O operations to complete. We will add a layer on top of this device to support the same blocking read and write operations as the basic implementation, but without busy waiting in the device driver code. The user should be able to call read and write operations without worrying that the call may either return prematurely (i.e. before the disk is ready to transfer the data) or tie up the entire system waiting for the device to return.

2. Scheduler

We assigned a blocked-thread queue associated with each disk. This is done in the "scheduler.C". As is shown in Fig 1, we initialize a block queue when constructing a scheduler. We also add a "blocking" function, where we insert the thread into the block queue (Fig 2).

```
50  Scheduler::Scheduler() {
51
52    Node ready_queue;
53    unsigned int ready_queue_size = 0;
54
55    Node block_queue;
56    unsigned int block_queue_size = 0;
57
58
59    Console::puts("Constructed Scheduler.\n");
60  }
```

Fig 1. Scheduler constructor.

```
void Scheduler::blocking(Thread * _thread) {
    block_queue.enqueue(_thread);
    block_queue_size++;
}
```

Fig 2. "Blocking" in scheduler.C.

Then in the "resume" function of the scheduler, we check the status of the disk queue and status of the disk. If it is ready, we extract the thread from the disk block queue and put it in the ready queue. On the other hand, if it isn't, we return. The modification in this part is given in Fig 3.

```
77  //
78  void Scheduler::resume(Thread * _thread) {
79
80      ready_queue.enqueue(_thread);
81      ready_queue_size++;
82
83      if (block_queue_size != 0) {
84          // check if the disk is ready
85          if ((Machine::inportb(0x1F7) & 0x08) == 0) {
86              return;
87          }
88
89          Console::puts("blocking queue size:");
90          Console::puti(block_queue_size);Console::puts("\n");
91
92          Thread* target = block_queue.dequeue();
93          block_queue_size--;
94
95          ready_queue.enqueue(target);
96          ready_queue_size++;
97      }
98
99
100     Console::puts("Resuming ... \n");
101
102 }
```

Fig 3. “Resume” in scheduler.C.

3. Blocking Disk

Inside the Blocking Disk, we define a different “wait_until_ready” function. When a thread issues a read operation, it queues up on the disk queue and then yield the CPU. The change in code is given in Fig 4.

```
40  void BlockingDisk::wait_until_ready() {
41
42      SYSTEM_SCHEDULER->blocking(Thread::CurrentThread());
43
44      SYSTEM_SCHEDULER->yield();
45  }
46
47
48  void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
49
50    SimpleDisk::read(_block_no, _buf);
51
52  }
53
54
55  void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
56
57
58    SimpleDisk::read(_block_no, _buf);
59  }
```

Fig 4. Blocking Disk.

4. Other changes

There are also a minor change in the kernel file. Here we use the Blocking Disk instead of the original Simple Disk. The change is illustrated in Fig 5.

```
288
289     /* -- DISK DEVICE -- */
290
291     // SYSTEM_DISK = new SimpleDisk(MASTER, SYSTEM_DISK_SIZE);
292     SYSTEM_DISK = new BlockingDisk(MASTER, SYSTEM_DISK_SIZE);
293
```

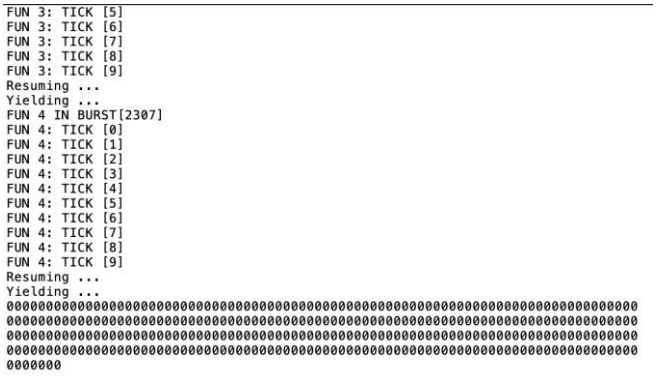
Fig 5. Change in kernel.C.

5. Results

Here we give some snapshots from the screen, as is shown in Fig 6. Note that these snapshots are not continuous. It indicates that the user is able to call read and write operations without worrying that the call may either return prematurely or tie up the entire system waiting for the device to return.

(a)

(b)



(c)

Fig 6. Shapshots from the screen.