Anuska Pant
CSCE 611 - OPERATING SYSTEMS

Machine Problem 6: Primitive Disk Device Driver

Bonus: Option 2, Option 3, Option 4

Introduction

The objective of this machine problem is to investigate kernel-level device drivers on top of a simple programmed-I/O block device, i.e to add a layer on top of this device to support the same blocking read and write operations as the basic implementation, but without the busy waiting in the device driver code .

Bonus Points

OPTION 3: Design of a thread-safe disk system

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 mp6 and bonus 3, 4.

As several threads call the read/write function we queue the read write requests to a block queue and make the thread give up the cpu. This way only a single read/write request is handled at a time.For a multiprocessor system, we will race conditions between several cpu to access the I/O queue and will need some form of protection. For this we will need to implement a thread safe queue and a lock on the I/O device. The thread safe queue will make sure that different processors do not update the queue in the wrong manner and a lock on the I/O device will ensure that we do not give up the resource if it is busy. This implementation cannot be tested in our current setup as we only have one processor.

OPTION 4: Implementation of a thread-safe disk system

The scheduler class is updated with a blocking disk object which is added using the addDisk() function. The yield function in the scheduler is modified to check for the disk object and verify disk reads and write and then scheduling of the thread.

Next, the blocking disk class is implemented, where it will contain a queue of blocked threads on the disk. We add a queue size member as well which will track the total number of threads in the queue. The extra functions which we add to the blocking disk class are:

1. Is ready- this function calls the is_ready function from the simple disk class.

2. Wait until ready- This checks if the disk is ready or not using the is_ready function. If the disk is not ready we take the thread that issued the read/write and add it to the blocked queue. The thread yields the cpu and control is given to the next thread. When the disk is ready we dequeue the thread from the blocked queue and give the cpu. When the disk isn't ready we let the threads in the ready queue to execute in a loop. This way we avoid a busy waiting loop.

Option 2 (DESIGN) : Using Interrupts for Concurrency

Since the disk is configured to raise INTERRUPT 14 on ready, this can be used to avoid polling and interrupt the threads. First of all we register the Interrupt 14 with the interrupt handler class which we define to handle a list of operations to be performed. In the handler function we dequeue the thread that raised the Interrupt14 and add it to the ready queue which will enable us from not having to check the disk every time.

Files Modified :

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) SCHEDULER.C AND SCHEDULER.H : Changes to the current yield function and an additional function addDisk() added..