David Merrick

CS 411

27 April, 2013

Individual Writeup for Project 2

1. *What do you think the main point of this assignment is?*

   The main points of this assignment were to learn about the block I/O subsystem of the kernel, gain a sense for using the built-in data structures that the kernel provides, and to write our own custom kernel code. Block I/O is an entire subsystem of the kernel, and this assignment gave us an in-depth understanding of how that system functions at a low level. The kernel data structures we made use of and the operations that can be performed on them are used universally in kernel development and will undoubtedly be very useful in future assignments. In the last assignment, it was a trivial matter to diff the 3.0.4 kernel to find out which changes needed to be made to the process scheduler. In this assignment, we did not have that luxury so we were completely on our own in writing our custom kernel code. There was a learning curve associated with that, which helped prepare us for future assignments in which we will always be expected to write custom code.

2. *How did you approach the problem? Design decisions, algorithm, etc.*

   Background: Block devices are characterized by random access of fixed-sized blocks of data. The smallest addressable unit on a block device is the sector. All device I/O must be done in units of sectors, which are then mapped to blocks in memory. The Linux kernel provides an entire subsystem for block devices, the source code for which can be found under the "block" directory. The most common type of block device is the hard disk. Hard disks are complex enough that optimizations can be created for performing I/O operations on them.

   The subsystem dedicated to performing these I/O operations is called the I/O scheduler. Block devices maintain request queues to store their pending I/O requests. The I/O scheduler essentially works by managing this request queue, deciding the order of requests in the queue and when to dispatch each request to the block device. Seek time is the amount of time required to position the hard disk's head at the location of a specific block. One of the primary optimizations that can be made to block I/O performance is minimizing seek time. I/O schedulers do this by merging and sorting the requests in the queue. The entire request queue is kept sorted sectorwise so that all seeking along the queue moves sequentially. Since this is similar to the algorithm employed by elevators, I/O schedulers are referred to as elevators.

   There are several flavors of I/O schedulers built in the kernel already and optimized for different purposes. The Deadline I/O Scheduler is designed to prevent starvation of certain requests (such as write operations starving reads). The Anticipatory I/O Scheduler is designed to maximize thoroughput and is ideal for servers. The Completely Fair Queueing (CFQ) I/O Scheduler performs well in nearly all workloads and is the default scheduler in Linux. The Noop I/O Scheduler only performs merging of requests, maintaining the request queue in near-FIFO order. It is intended for block devices that are truly random-access, such as flash memory cards.

   As for managing request queues, the kernel provides several data structures. Kernel developers are encouraged to use these data structures wherever possible as opposed to designing their own. While generic, they have been thoroughly tested and perform well in most circumstances. The most useful of these data structures are linked lists, queues, maps, and binary trees. The I/O schedulers use either linked lists or binary trees to manage their request queues. For our purposes in designing our shortest-seek-time-first I/O scheduler, the built-in linked list data structure was well-suited. The kernel provides methods for adding to, deleting from, and retrieving items from lists, as well as moving items from one list to another or to a different location in a list. Our particular implementation was a circular linked list, meaning that the last item in the list was linked to the first.

   Design:

   We designed our system to perform exactly the way an elevator does. The scheduler will dispatch requests in order of increasing sectors until it reaches the highest one, then it will dispatch requests in order of decreasing sectors until it reaches the lowest. It will then repeat this cycle.

   Algorithm:

   When new requests are added to the queue, our scheduler iterates through the other requests and adds the new request in order of sector. If there are no requests, it is added first. One optimization we made was a comparison between the new request's sector and the sector of the request at the head of the queue. If the new request is lower,

we iterate in the lower direction. If it is higher, we iterate in increasing request sector order. Overall, this algorithm ensures that the request queue will always be sorted. The SSTF data structure we designed holds a variable to indicate the direction the disk head is moving through the queue. Initially, it is set to 1 (forward), which will cause the scheduler to iterate through the requests in order of increasing sectors and dispatch them. When it reaches the highest sector in the queue, this variable is changed to 0 (backward), causing the scheduler to move through the requests in order of decreasing sectors. When it reaches the lowest sector, the direction variable is changed back to 1 and this process repeats itself. This algorithm provides two benefits: it minimizes seek time without starving the requests on the edges of the queue.

Code:

We used the Noop scheduler as our template for designing our SSTF scheduler. We changed most instances of "noop" to "sstf" to distinguish our functions and variables, but to a large degree we kept the same functionality of the Noop scheduler.

One change we made was to the noop_data struct. We changed this to sstf_data and added head_pos and direction variables to track the current sector and the direction the scheduler is moving through the queue, respectively.

We modified sstf_add_request() to place all new requests in a queue that is sorted by sector. We added local variables to this function to hold the current request's sector as well as the sectors of the next and previous requests in the queue. This was necessary to be able to compare the sectors in order to place the new request where it belonged in the queue.

We modified sstf_dispatch() to use our sstf_data struct to keep track of the direction the scheduler is moving through the queue. This function is the meat of the I/O scheduler; it does the actual dispatching of requests. We implemented our elevator algorithm inside this function to have the scheduler switch directions when it reaches the request with the highest and lowest sectors. After dispatching each request, this function deletes it from the queue.

3. *How did you ensure your solution was correct? Testing details, for instance.*

We used printk statements inside our scheduler for debugging and testing. We initially used the no-op function for dispatching and used printk statements inside our add_request() function to make sure that requests were being added in the appropriate order of sectors. We grepped dmesg for the output of these statements. Once we knew that add_request() was working, we then wrote our dispatch function. When a request was dispatched, we printed the its sector. When the elevator switched directions, we printed the current direction and the new direction. This provided us the context to be able to determine whether requests should be printing in ascending or descending order of sectors, and then we were able to check the dispatched requests' sectors to confirm this.

4. *What did you learn?*

Before working on this assignment, I just assumed that block I/O was a basic aspect of the kernel, happened by magic, and there was only one way to perform it. I now know that block I/O has an entire subsystem in the kernel dedicated to it, that block devices can vary greatly in how they store and access data (flash memory cards vs hard disks, for instance), and that there are several different optimizations that can be made for different purposes. I had heard the term "sector" previously but now have an understanding that it is the smallest addressable unit on a block device. I learned how to use the built-in kernel data structures and re-learned some aspects of my sophomore data structures class that I had forgotten.