Spencer Rawls
CSCE 410-500
5/2/17

MP6 Design Documentation

I added in my implementation of the scheduler class from the previous machine problem, and modified the Thread class by declaring Scheduler as a friend class of it to make that solution work again. The Scheduler class implements a FIFO ready queue with the variables readyQueue, readyStart, readyEnd, readyCapacity, and readyQueueEmpty. ReadyQueue is an array of thread pointers, readyStart is the index in that array of the head of the queue, readyEnd is the index after the tail of the queue, readyCapacity is the total room in the array, and readyQueueEmpty is a bool representing whether the queue is empty or not. Elements wrap around the array mod readyCapacity. Adding a thread to the queue resizes the array if necessary, and increases readyEnd by 1 mod readyCapacity. Removing a thread from the front of the queue increases readyStart by 1 mod readyCapacity. The resume function adds the thread to the queue, and the yield function pops the next thread from the front of the queue and dispatches it.

I modified the SimpleDisk class only by making the issue_operation function protected so that my BlockingDisk read and write functions could call it. I also modified my Scheduler class by adding a function registerDisk(BlockingDisk* disk). This function is called by each blocking disk in its constructor to register itself with the scheduler. The scheduler keeps a vector of BlockingDisks, so that each time the yield function is called on the scheduler and a new thread is about to be dispatched, it can tell each blocking disk to check if the disk is ready, and if so, place the blocked thread back on the ready queue.

I implemented blocking I/O operations in BlockingDisk by giving each disk a queue of blocked threads. I added three private functions and one public function to the BlockingDisk class. The private functions are:

```
Thread* getNextBlocked();
Thread* popNextBlocked();
void addBlocked(Thread* t);
```

getNextBlocked returns a pointer to the thread currently at the head of the blocked queue. PopNextBlocked removes the thread at the head of the blocked queue and returns a pointer to it. AddBlocked adds a thread to the end of the blocked queue. The public function I added is void checkAndResume(). This is the function that the scheduler calls on each disk to check if the disk is ready and if so, pop the head of the blocked queue, tell the scheduler it can resume, and if there was another blocked thread waiting on that one, issue the operation for it.

I implemented the blocked queue in the same way I implemented the ready queue in my scheduler: with variables blockedThread (the array), blockedStart, blockedEnd, blockedCapacity, and blockedQueueEmpty. My read and write functions first add the current thread to the disk's blocked queue, then call yield on the scheduler in loop while there are other blocked threads in front of the current thread. Once this thread is at the head of the queue, it issues the operation to the disk and then yields again. Once the disk is ready, the next time yield is called on the scheduler by another thread, the scheduler will call checkAndResume on the disk, which will move that thread back to the scheduler's ready queue so it can complete the read or write operation.

Lastly, obviously, I modified kernel.C to use a BlockingDisk instead of a SimpleDisk.