David Fantin
ID: 1525813
CS-3053
4/21/2020

OS Project 2 Report

Task 1: Boat from Oahu to Molokai

The first thing I had to do before implementing (or figuring out) the bulk of the logic was to create each child and adult thread. I did this by first copying the given code for the thread creation already in nachos and passed it once (one for adults and one for children) and for each all I did was iterate through each child/adult using a for loop, and in the run method put “ChildItinerary()” and “AdultItinerary()” respectively (I also set a name for each of the threads as well). To finish out the begin method I used a condition variable I created called “end” so after acquiring the lock, I slept end. This works as a finish condition, so when ever there are no more people left on Oahu, I wake end, which tells the program to end and stop (If I don’t have this the program never ends and loops infinity).

Next was the Adult Itinerary method. For this, the main idea for the logic is this: if there are adults on Oahu they get to row to Molokai if and only if the boat is on Oahu and there are either 0 or 1 children left on Oahu (this last part I checked by having a variable that keeps a running count of how many children are on Oahu called “childrenLeft”). The reason for this last condition is that for the test to work with any number of adults, there must be at least 2 children total, so if there are 0 or 1 children, there must be at least one child on Molokai that can row back if there are any adults left. If this test does not pass, sleep the condition variable “AdultsOnOahu,” since we can no longer move those threads until more children are sent over or the boat is on Oahu (I keep track of this by creating a boolean that is true when it is and false when it isn’t). if it passes though: I first decremented the variable “adultsLeft” (which I declared that keeps track of how many adults are on Oahu) then show that the boat is not on Oahu anymore, and then “bg.AdultRowToMolokai()” to actually make the thread go to Molokai. After this I have my FINISH CONDITION if statement to check if there is anyone else on Oahu, and if there isn’t anyone: wake end to stop the program (I will use this later in the code as well). Next (assuming the FINISH CONDITION does not pass), we need to get the children threads on Molokai ready to send a child thread back with the boat by waking all child threads on the condition variable “ChildrenOnMolokai” and sleep all the adults on Molokai (since they can’t do anything anymore) with AdultsOnMolokai.sleep(). All of this code in the adults itinerary is looped while there are still adults on Oahu.

Now the Child Itinerary, which was MUCH more complicated (NOTE: for this method I outlined most of my thought process in the Image at the end of this report). After the lock is acquired for the condition variables I made it so all of the code loops while there are still children or adults left on Oahu. So for this method the bulk of the logic can be divvied up into 3 main sub splits. The first split is to see if the boat is on Oahu or Molokai. If it is on Oahu: Check if there are no children or one child on the boat (there would be one if and only if there is a child rowing already). Here if there are no children on the boat, make a child row to Molokai, else if there is one child on the boat make one child ride to Molokai then test the FINISH CONDITION wake all children on Molokai then immediately sleep them (these two lines help with synchronization between the adults and children and are used a few more times in this method: I’ll refer to them as the Sync Lines). There is also a 3rd condition to check if the boat is on Oahu, and if it isn’t sloop the children on Oahu. The final split is to check if there are any children left on Oahu: if there are, then wake all the children on Oahu so that they can prepare to move, if not test the FINISH CONDITION then I used the Sync Lines again. Back tracking a bit, if the boat is on Molokai and there are still children or adults on Oahu, you have to have a child row back to Oahu, so you need to make all the adults on Molokai then use the Oahu version of the Sync Lines, else sleep the children on Molokai (since they are not being used anymore). NOTE: Because there were so many splits in my logic for the children itinerary I couldnt think of a solution that did not involve the use of two booleans for where the boat was (boatIsOnOahu, and TMPboatIsOnOahu).

Task 2: Priority Scheduler

For the priority scheduler part of this project I relied HEAVILY on Dr. Papa's collaborate video, and I will note where I used it in this report with an asterisk (*). The first method to implement was nextThread()*. What I had to do here was first make a variable of type thread state that picked the next thread, then have a test to see if it = null, and if it did, then return null → if I did not do this then I would get a null pointer exception error. Then, if it wasn't null, just return the thread associated with the state by using "nextThreadState.thread."

Next was pickNextThread()*. For this method I needed to create a thread state queue like in video (I used a Linked List because I am comfortable using them). I created TSQ with the line: "public LinkedList<ThreadState> TSQ = new LinkedList<ThreadState>()." Now for the method steps itself: 1) Check if TSQ is empty, if it is, return null, (you don't want the null pointer exception errors) 2) create a variable of type threadState that is equal to the first element on TSQ and should represent the element with the highest priority. 3) Loop over TSQ and check if the priority of the variable you just created is less than each elements priority in the list, and if it is then set it equal to that element. This is done because the goal is to return the next theadState that has the highest priority (since it should be done first anyway). After this, the for loop ends, so now we just need to remove the element with the highest priority from the list (since we already have it) and return it.

After skipping the print() method, let's move on to setPriority() in the ThreadState class, I honestly didn't chance anything from the original nachos file and the autograder still passed with "this.priority = priority," so I did not want to mess with it. For the waitForAccess()* method all I needed to do was add the current threadState to the "waitQueue" priority queue. The last of the short methods was the aquire method, which basically adds the waitQueue to the priority queue of priority queues owned by an owner of type KThread that the video mentioned. I created this queue in the exact same way as the TSQ except for type ThreadState, became type PriorityQueue. And finally it was time for the big one: getEffectivePriority()*. For this I got nearly all my code directly from the collaborate video, so I used the recursive method. The steps are: 1) create an integer that represents the effective priority (called "epriority") and set it = to the priority of the associated thread. 2) loop over the Priority Queues owned by the thread I mentioned previously (PQ_Owned). 3) Set the condition that if the queue transfers priority, then you now must iterate through each threadState in the TSQ associated with the priority queue. 4) Create a new integer that = the Effective Priority of the threadstate (this is the recursion part). Next I needed to check if that variable is greater than epriority, and if it is, set epriority to it. Then after all the loops, finally return epriority.

I did run into problems with the 4th autograder test however (this took SOOO long to figure out!), so I needed to modify the KThreads class. I first needed to change private static ThreadQueue JQ = ThreadedKernel.scheduler.newThreadQueue(false) to: public ThreadQueue JQ = ThreadedKernel.scheduler.newThreadQueue(true) and then I had to change the loop in my finish method to account for it no longer being static.

Children

First split
- (1) Boat B on Oahu
- (2) Boat B on Molokai

Secondary split (of (1))
- (a) No children on Boat
- (b) One child on Boat

Tertiary split (of (a))
- (I) There are children left on Oahu
- (II) There are NO children Left on Oahu

If II → Finish.

- Need 2 Booleans to check for ~~if the Boat B or~~ where the Boat B!!