Raymond Chan, Brandon Lau
11/12/18
ECS 36C
Joël Porquet, PhD

Phase #1: Headers and Tester Coverage

The header file llrb_map.h is very similar to llrb_set.h, with slight modifications on the class definition (now takes in both key and value), as well as the Get(), Insert() and Remove() functions; all of which just make the implementation compatible with mult-typename definition. The llrb_multimap.h is similar to llrb_map.h but allows for duplicate key insertions, and for removals, only the first element in the key is removed if key holds multiple elements.

For map_tester.cc, we accounted for the new implementation for map's Removal, Insert, and Get function. For Removal, since one of the implementation is to replace target removing node to the min node of its right subtree, we need to check if the replacement is properly done, with both the key and value replaced. For Insert, we need to make sure both the key and value are inserted properly, as well as checking for duplicating key being invalid insertion key. The Get function should return the right value instead of the key, and that Get from empty tree and for non-existing key are invalid

For multimap_tester.cc, we just need to test if Get would always return the first value of the list of value contained by a key, and that inserting duplicate key only extents list of value, while removing duplicate key only remove the first element of the list, until the whole list is empty, which then the key should be removed from the tree.

Phase #2: CFS Implementation

In our CFS implementation, we utilized a Schedule class that contains a struct Task. Each line from the file is a Task object. Schedule is a wrapper for all of the operations performed on the Tasks. In terms of data structures, we utilized a vector containing pointers to Task objects, and we used a multimap (LLRB implementation) that has the key as the Task's individual virtual runtime which are mapped onto corresponding Task pointer objects. In addition, we kept track of each Task's individual virtual runtime in its struct, and we had a pointer in Schedule that kept track of the current task to process.

Following the scheduling loop steps provided in the homework assignment, we utilized a top-down approach and broke our code up into different functions, where each function would be a step. In the first step, we had the function SearchForUnlaunchTask() that finds if there are tasks that match the current tick value and need to be scheduled. In the 2nd step, we had CheckOwnershipForProcessRescource() that checks whether or not to process the next task. In the 3rd step, we had CheckCurrentTaskIsNull() that obtains the next task from the timeline when there is no current task to process. In the 4th step, we had ProcessCurTask() that performs the incrementations of the Task's duration and virtual runtime, and in the final step, we had PrintAndIncrementGT() that outputs the scheduling status. As shown, each function represents a step in the scheduling loop steps that the Professor provided in the assignment.