Lab 2 Report

ECE350 Group 57

Data Structures and Algorithms

To implement the scheduler, the optimal data structure to use was a priority queue. In our priority queue, we have an array of TCB pointers which we treat like a binary tree. A min heap is an optimally efficient implementation of a priority queue, where the min value is always at the root of the heap. In our case, the minimum value that we're interested in is the priority of a task, where a smaller value translates to a higher priority for the task. We needed to implement the following methods for the min heap: enqueue, dequeue, remove, change priority, and peek.

1. To enqueue a task (add it to the heap), we start off by adding it to the binary tree in the first open space in the bottom level of the tree, to try to maintain a complete binary tree. However, we also want to maintain a rule in the tree: at any given node, its parent node is lesser in priority value (except the root node, which will be the minimum). Thus we need to take the node that we just added to the tree, and bubble it up through the tree (swap with parent node) until its parent is lesser in priority. Thus the time complexity of enqueueing a task is *O(log(n))* where n is the number of tasks in the heap, because the largest height that we could bubble up a node in a complete binary tree is log(n).
2. To dequeue the highest priority task (remove min node from the heap), we replace the root node with the last node in the tree, then heapify the tree to 'bubble down' the new root node into its desired position. This entails swapping it with lesser-valued children until both children are larger in value. Thus, with the same reasoning as enqueue, the time complexity of dequeue is *O(log(n))*.
3. To remove an arbitrary node from the heap, we set its priority to be the minimum possible value, then bubble it up through the tree so it ends up at the root. Then we call the dequeue method, which removes the root node. The time complexity of these operations is still *O(log(n))*.
4. When we've changed the priority of a node, if the new priority is less than the old one, we need to bubble up the node through the tree, but otherwise we'd need to bubble it downwards. This will restore the condition that all nodes' parent node is less in value. The time complexity of this method is also *O(log(n))*;
5. To peek at the max priority task in the heap (which has the minimum priority value), we just need to look at the root node, because we know that our other methods maintain the root node as the minimum value. Thus the time complexity of peek is *O(1)*.

We also used a linked list to manage the TCB data structures in memory.

Testing Scenarios

Test Case 1: k_task_get and k_tsk_set_prio

To test out k_tsk_get and k_tsk_set_prio, there are 2 tasks that call k_tsk_set_prio. First, after creating the task, k_task_get is called to confirm the tcb info. Then, the kernel task changes the task priorities to LOW and LOWEST and k_tsk_get is called to confirm the change. Afterwards, the user task calls k_tsk_set_prio to set the tasks as HIGH and MEDIUM. The k_tsk_get function is called to confirm that the user task priority has been changed, but the kernel task has not been changed.

Test Case 2: mem_alloc and mem_dealloc (owned memory)

1. In a first user task, create another user task with a high priority and yield.
2. Now the high priority task should be running, where it will allocate memory that it will own. It also sets a global scope pointer to point to that memory block.
3. Change priority from HIGH to LOWEST, and yield back to the first user task.
4. Try to deallocate the memory block owned by the second user task, which should fail.
5. Eventually the scheduler will run the second user task.
6. The second user task will deallocate the memory block, which should deallocate successfully, since this task is its owner.

Test Case 3: Scheduling tasks with same priorities and different priorities

Let's create 3 tasks on boot, the NULL task of priority NULL, and two user tasks of priority LOW. The method k_tsk_init will add all of these tasks into the g_tcbs array. From there, k_tsk_run_new will call the scheduler to get the highest priority task in the system and set it's TCB state to RUNNING and the gp_current_task will point to it. Scheduler will initialize the priority queue and set the first user mode task to be the highest priority because it is of the same priority of the second task, but was created first.

1. Let's say that another user task, *task 3,* is created with a priority HIGH, and the gp_current_tsk yields. Create would set flags in the system, allowing the scheduler to update it's priority queue. The priority queue would have *task3* run next, as it now has the highest priority
2. At this point, another task, *task 4,* is created with a priority HIGH. When the scheduler is called to update its state, it'll insert the *task4* to run **after task3**, and therefore when k_tsk_run_new is called the current running task, *task3,* will be returned