**Algorithm Design:**

I intend to implement a scheduling algorithm that adopts a methodology very similar to “Shortest Job First” so that jobs with the shortest estimated processing time by the SAC-SimOS will be processed first in order to minimize the average waiting time of each individual process that the synthetic CPU will be responsible for. However, I intend to build my algorithm in a preemptive manner so that my CPU scheduler can effectively multitask by switching and processing multiple jobs across two independent priority queues within a very short amount of time.

**Algorithm Pseudocode:**

1. Create a MinHeap Priority Queue Class
   1. Construct the Priority Queue in a way so that it automatically determines what the position of each new process in the queue should be based on whether its estimated completion time is shorter than other processes already in the queue
2. Instantiate 2 new Priority Queue objects in the new CPU\_Scheduler class
   1. The Primary Priority Queue will act as the initial queue that new processes are added to in order to be scheduled and later processed by the SAC-SimOS
   2. The Secondary Priority Queue is necessary so that processes can be switched (and automatically sorted in the new queue) once their allotted CPU Bursts have been spent. This same procedure will repeat itself once processes in the Secondary Priority Queue have used their allotted CPU Bursts and will then be sent back to the Primary Priority Queue until they have been fully processed and dropped entirely from the queue altogether
3. Create CPU\_Scheduler Class Constructor that implements the pre-made Process Scheduler Interface and its accompanying methods
4. Create and set burstRate to 10 so that each process is allocated up to 10 cycles before it is switched to the other queue in order to maintain preemptiveness
5. Create an “addProcess(SimProcessInfo process)” Method that is responsible for initially adding new processes to the Primary Priority Queue structure based on how quickly they processed by the SAC-SimOS
   1. Ensure that it will automatically sort through all elements and prioritizes those that have the shortest estimated completion time as their Keys so that they will be run by the SAC-SimOS first (thus minimizing the average waiting time for processes sitting in the queue)
   2. Add the new process to the Primary MinHeap Priority Queue accordingly
6. Create a “getNextProcess()” Method that is responsible for returning the p\_id (as an integer) for the next scheduled process that SAC-SimOS will run
   1. If the Primary queue has nothing in it, set boolean pqPrimaryEmpty to true and boolean pqSecondaryEmpty to false
   2. If the Secondary queue has nothing in it, set boolean pqPrimaryEmpty to false and boolean pqSecondaryEmpty to true
   3. Check to make sure that there is at least one more process in either the Primary or Secondary queues that needs to be run
      1. If there isn’t another process remaining, return “-1”
   4. Check to see whether pqSecondaryEmpty is true (if so do the following)
      1. While there are still processes left in the Primary queue and the state of the process at Index 0 of the queue is equal to “Terminated”, remove the process at Index 0 from the queue and push it to the Secondary queue
         1. Reset CPU Burst Counter to burstRate
      2. If CPU\_BurstCounter drops to 0, pop that process from the Primary queue and push it to the Secondary queue in order to maintain preemptiveness (and therefore increase efficiency)
      3. Decrement the “Key” of the Process at Index 0 in the Primary Queue by 1 to indicate that the process at Index 0 requires one fewer clock cycle (and therefore has less estimated computational time remaining) so that new processes being added to the queue can be added appropriately based on their total estimated computation time compared to the remaining computation time of the processes already in the queue
      4. If there are still items in the Primary queue:
         1. Check to see if the current item being processed has the same key as the item that was being processed in the previous clock cycle (with remainingBurstCount)
            1. If not, reset the CPU\_BurstCounter to burstRate
         2. Decrement the current item’s key by 1
         3. Decrement the CPU\_BurstCounter by 1
         4. Set remainingBurstCount equal to the current item’s key
         5. Return the p\_id of the next process at Index 0 in the Primary Queue
      5. Otherwise, set pqPrimaryEmpty to true
   5. Check to see whether pqPrimaryEmpty is true (if so do the following)
      1. While there are still processes left in the Secondary queue and the state of the process at Index 0 of the queue is equal to “Terminated”, remove the process at Index 0 from the queue and push it to the Secondary queue
         1. Reset CPU Burst Counter to burstRate
      2. If CPU\_BurstCounter drops to 0, pop that process from the Secondary queue and push it to the Primary queue in order to maintain preemptiveness (and therefore increase efficiency)
      3. Decrement the “Key” of the Process at Index 0 in the Secondary Queue by 1 to indicate that the process at Index 0 requires one fewer clock cycle (and therefore has less estimated computational time remaining) so that new processes being added to the queue can be added appropriately based on their total estimated computation time compared to the remaining computation time of the processes already in the queue
      4. If there are still items in the Secondary queue:
         1. Check to see if the current item being processed has the same key as the item that was being processed in the previous clock cycle (with remainingBurstCount)
            1. If not, reset the CPU\_BurstCounter to burstRate
         2. Decrement the current item’s key by 1
         3. Decrement the CPU\_BurstCounter by 1
         4. Set remainingBurstCount equal to the current item’s key
         5. Return the p\_id of the next process at Index 0 in the Secondary Queue
      5. Otherwise, set pqSecondaryEmpty to true

**Algorithm Analysis:**

* *What do you take into account when considering priority?*

My algorithm prioritizes jobs with shorter estimated completion times in order to minimize the total average wait time of other processes just sitting idly in my priority queues. My algorithm also makes a concerted effort to update its count for the estimated computation time of each process in the queue waiting to be processed in order to allow it to be even more efficient.

* *How does your algorithm treat high priority processes (i.e. what does it do differently)?*

My algorithm prioritizes processes that have a shorter estimated completion time and elevates them to the beginning of the given priority queue so that they will be processed sooner by the SAC-SimOS CPU. This will in turn decrease the average waiting time of all processes sitting in the queue.

* *Prove whether or not starvation is possible with your scheduling algorithm.*

Starvation is possible with my algorithm because processes that are estimated to have longer remaining completion times will always get pushed down towards the end of the given priority queue. However, the tradeoff is that more processes with shorter estimated completion times will be run by the SAC-SimOS operating system sooner, which in turn minimizes the overall average waiting time for processes being scheduled by my algorithm. In other words, unlike a Round Robin implementation for a CPU scheduler, my algorithm fully embraces the idea that it’s not going to treat every process equally so that it can pop smaller jobs from the given queue faster.