**Design Description**

The RMS is launched by the main thread. Once launched, the RMS in turn launches 5 more threads, these are: the 4 BusyWork threads which contain deadlines that the RMS will strive to help them meet, and the dispatch thread. The dispatch thread contains an instance of the Dispatcher class, and is responsible for taking the threads scheduled by the RMS but have been paused as a result of waiting on a higher priority thread to finish running, and resuming it once that higher priority thread signals that it has finished (by setting a bool finishedRunning to true and releasing the semaphore this thread was waiting on). Without this dispatch thread, those threads that paused themselves would never resume. The RMS stops and renews the dispatch thread every scheduling time frame in order to prevent the dispatch thread from interfering with the RMS. Results are printed out only after the RMS has completed 10 frame periods. The main thread launches the RMS twice: one for the first case where there are no overruns, and once for the second case where t2 has an overrun. In order to synchronize the threads, the following process was taken: BusyWork threads 1, 2, 3, and 4 each received a semaphore of their own, thread 1’s semaphore is always set to 1 everytime the RMS does scheduling, and the remaining threads have their semaphores set to 0. Doing so makes it so that when the RMS starts the threads, the threads with closed semaphores will automatically put themselves to sleep until the dispatch thread receives notice that the thread being waited on has finished. In addition, threads 1, 2, and 3 have references to the semaphores of the thread waiting on them to complete, so thread 1 stores a reference to thread 2’s semaphore, thread 3 stores a reference to thread 4’s semaphore. When a thread finishes, it not only sets its finishedRunning bool to true and increments its counter for how many times it ran, but also releases the semaphore of the thread waiting on it, so that when the dispatch resumes that thread, it does not pause itself again; this, the combination of semaphores and the dispatch thread allowed for synchronization. The RMS was also synchronized with a semaphore, called RMSSemaphore in the RMS class. This semaphore is set to 1 in the constructor, and each time the RMS attempts to schedule, and must first acquire this semaphore. Once acquired, the RMSSemaphore is released by a timer, allowing the RMS to schedule things only after a set delay has passed, for this project this delay was idleFramePeriod \* unitSize, or 1 \* 10ms for a 10 ms delay. The idleFramePeriod was set to 1 because this is the smallest period amongst the BusyWork threads; if this were to be any value higher than the RMS would not schedule things fast enough and many deadlines will be missed. Lastly, for case 2, the overrun in BusyWork thread 2 caused threads 3 and 4 to also have several overruns.