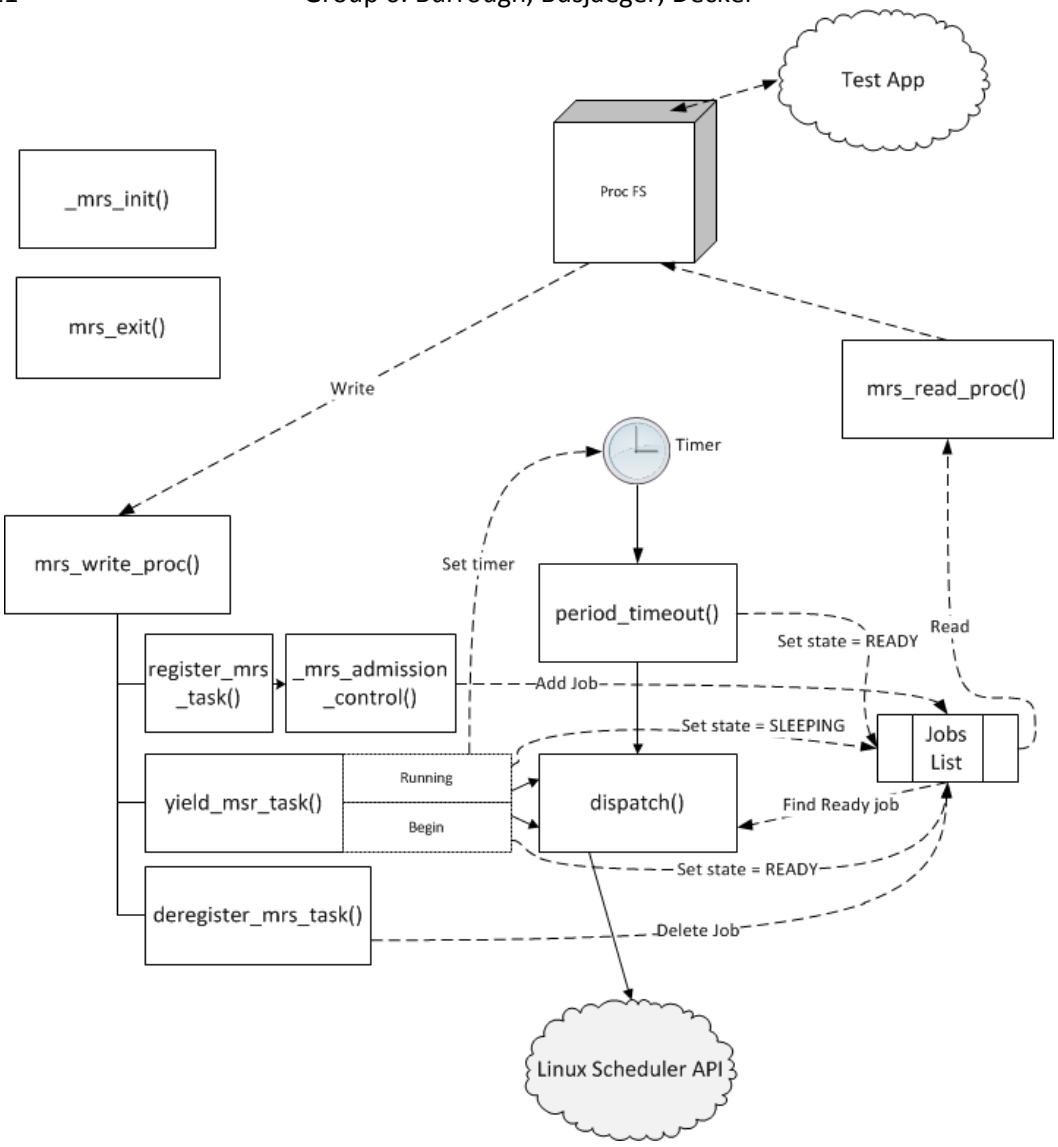

1. Synchronization

The MRS task data structure is concurrently accessed by multiple threads of execution and therefore needs to be properly guarded. We decided to use a spin lock for this purpose for several reasons:

- the durations during which the lock is held are very short
- the lock needs to be used in interrupt context, so a sleep-lock cannot be used
- consistency: the Linux scheduler also uses spin locks

We chose a variation of the spin lock that disables interrupts while the lock is held. This is to avoid deadlock that could occur in two separate circumstances. First, a timer could go off while the lock is being held by the task being interrupted. Second, a higher priority task may preempt a lower priority task that is holding the lock and try to acquire the lock (priority inversion).

In addition, the application and timer threads trigger the dispatcher to process state transitions they initiated. This is a type of producer/consumer problem with unbounded buffer. We chose to use a semaphore to coordinate this interaction: the dispatcher (consumer) calls 'down', while the application and timer threads (producers) call 'up'. The semaphore is initialized to 0, so the dispatcher is initially blocked. We reason for choosing a semaphore over wake_up_process calls is to avoid losing notifications of state transitions. Consider the following sequence of events:

task 1: period=15, runtime=5
task 2: period=5, runtime=1

- timer 1 goes off, puts task 1 in ready queue, wakes up dispatcher
- dispatcher wakes up and finds task 1 in ready queue
- timer 2 goes off, puts task 2 in ready queue, wakes up dispatcher (ignored)
- dispatcher schedules task 1

In this scenario task 2 misses its deadline, because the dispatcher already checked the ready queue before timer 2 was triggered. The use of a semaphore fixes this problem:
- timer 1 goes off, puts task 1 in ready queue, ups semaphore
- dispatcher wakes up, decrements semaphore, and finds task 1 in ready queue
- timer 2 goes off, puts task 2 in ready queue, ups semaphore
- dispatcher schedules task 1
- dispatcher decrements semaphore, finds task 2 in ready queue, stops task 1, and schedules task 2

Using these synchronization primitives had certain implications for the implementation. We had to take special care not to call functions that could put a task to sleep while holding the spin lock. Specifically, the dispatcher cannot call 'down' on the semaphore while holding the lock. Conversely, it is safe to invoke 'up' as this merely increments the semaphore and wakes up the first task in the semaphore's wait queue. In addition, we had to introduce a 'should_stop' flag to stop the dispatcher thread during module exit. The exit function sets this flag and then 'up's the semaphore to let the dispatcher know that the module is shutting down. Only after that can exit call kthread_stop to deallocate the thread.

2. Added state: NEW
Besides the READY, RUNNING, SLEEPING states, we decided to add another state called NEW to distinguish between the Initial Yield an application uses to tell the scheduler it is ready to run its Real-Time Loop, and the regular Yield to inform the scheduler it is done with its work and will block until the next period. When a new task is created during Registration its state is set to NEW and then on the Initial Yield its state is set to READY and its timer is started.