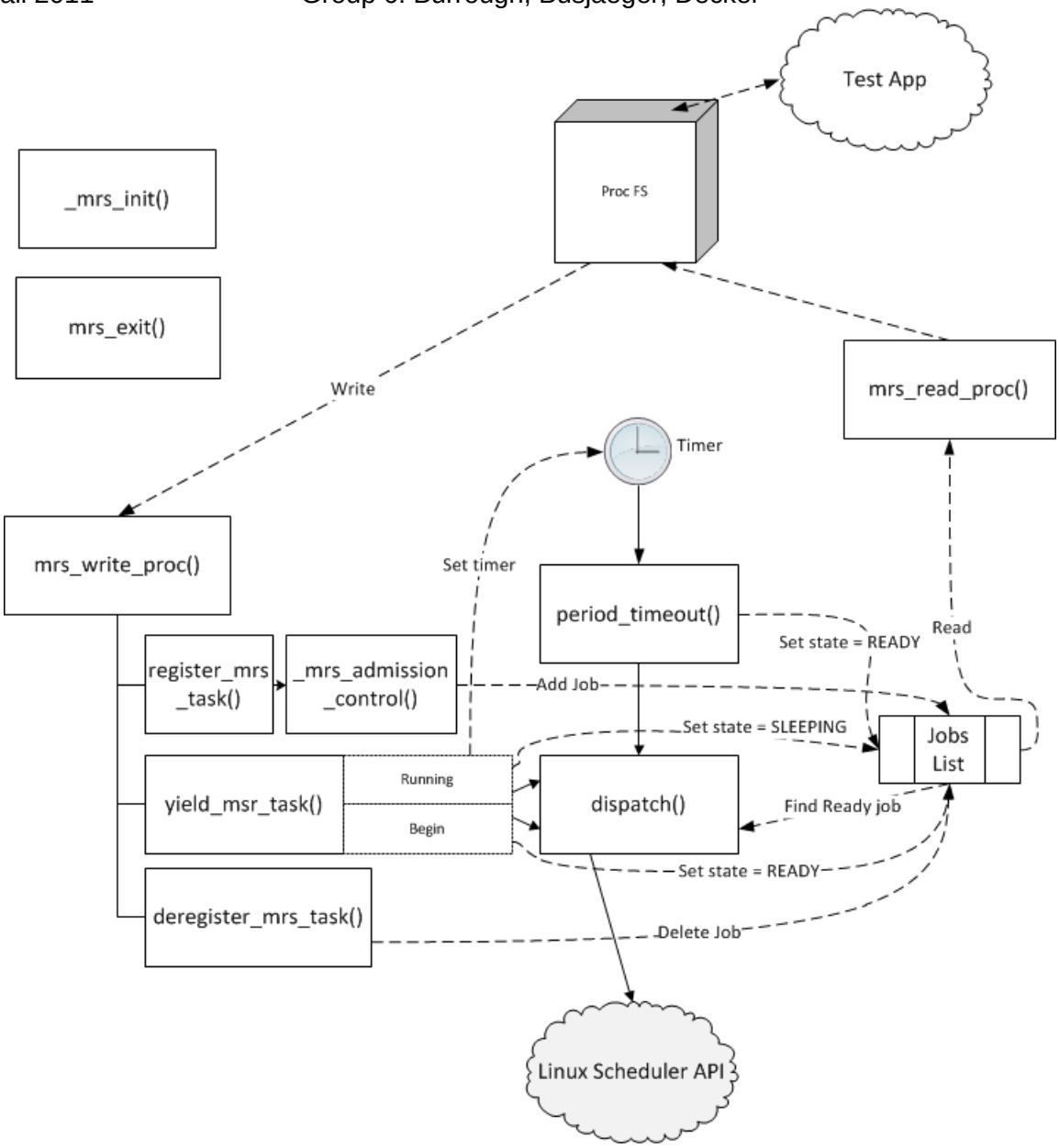

1. Synchronization

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. The 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. Task States
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.

3. Timer
The timer for a specific task is restarted whenever the timer handler for the task is triggered. We decided to this here rather than when the task yields, because:
1. it is not necessary to track how long a task ran
2. it allows dealing with clients that do not adhere to the contract
For example, the timer handler could be easily extended to preempt tasks which (repeatedly) do not yield within their designated period.

4. Scheduling Policy
The dispatcher always preempts the current task if it is sleeping. If there are ready tasks, the scheduler picks the one with shortest period and schedules it if (1) no task is currently running, or (2) if the ready task has a shorter period than the currently running task. Note that in the latter case the currently running task is preempted.

5. Error Handling
5.1 Timer Handler
If the timer's task no longer exists, then the timer exits without resetting the timer (this case should not occur). Otherwise the timer is always reset, even if the current task is still running or still in ready state. As previously mentioned, more sophisticated policies could be implemented here.
5.2 Write Function
Standard error codes are returned for possible error conditions. Examples for error conditions are invalid arguments, task already exists, or task not admitted. The system call returns the error to user space and the write proc also logs a kernel error.

6. Data Structures
We decided to use a linked list to hold tasks in all states as this is sufficient for the given requirement and keeps the code simple and short. In a more sophisticated design we would have indexed the task list by ID and created a separate ready-queue tree structure sorted by period. In addition, it would be beneficial to cache the admission threshold when tasks are added and removed.