Lab 4: Concurrency
Kevin Truong and Rob Brunstad

The required functions were implemented so as to interact with the global variable `current_process`. Because our `struct process_state` is defined such that each process points to a successor, our ready queue is defined as the queue starting at `current_process->next`. The relevant `struct` is reproduced below.

```
typedef struct process_state {
    unsigned int sp; /* stack pointer */
    struct process_state *next; /* link to next process */
    int block;
} process_t;
```

This implementation is convenient because it requires less pointer manipulation operations, thereby granting more efficient dequeuing.

Each function's implementation is explored thoroughly in comments included with the code, but a high-level walkthrough of the implementation is also provided here.

First, `process_create()` allocates stack space and receives a stack pointer by calling `process_init()`. More space is allocated for a new process, and its fields are initialized. If there does not exist a current process, the new process is made the current process. Otherwise, it is added to the ready queue.

Our implementation of a ready queue requires no initialization on the part of `process_start()`, as `process_create()` must necessarily initialize both `current_process` and the ready queue. Therefore we simply call `process_begin()`.

The function `process_begin()` calls `process_select()`, which returns the stack pointer for the next ready process or zero in the case that no such process exists. First and foremost, if there does not exist a current process, there cannot, as per our implementation, exist a process in the ready queue. Therefore `current_process` is checked against zero, and if the test returns true, 0 is returned. The next two tests involve the current stack pointer. If it has been set to a null value, one of two events must have occurred. Either the system has just been initialized, and `process_select()` should return the stack pointer corresponding to the current process so that it might actually run, or a process has just terminated, and `process_select()` should dequeue the next process waiting in the ready queue and free memory associated with the old process. In our implementation, `process_select()` distinguishes between these two cases by checking `run_flag`, a global variable that is initially set to 0. If `run_flag` was never changed, `run_flag` is updated to 1 and code corresponding to the first case is executed. If the flag is found to be 1, then code corresponding to the second case is executed. If none of these prior conditions have been met, then it must be true that there exists a current process and it should be updated to the process at the head of the queue. Of course, if there does not exist a `current_process->next`, then we simply return the current process's stack pointer so that work on it might be resumed. Importantly, it must also be true that the current process was interrupted, and so we save its progress by writing the current stack pointer to `current_process->sp`. Also, it must be placed back onto the ready queue because its work has not been completed.

In implementing locks, we define our `lock_state` structure as presented below.

```
typedef struct lock_state {
    struct process_state * curr;
    struct process_state * locked;
} lock_t;
```

Here a more traditional implementation of a queue has been established, where `locked` gives the waiting queue and `curr` gives the process currently in possession of the lock.

The function `lock_init()` performs a rudimentary initialization of the `struct lock_state` fields.

The function `lock_acquire()` necessarily requires that interrupts be disabled during its execution to ensure mutual exclusion. Essentially, the function must determine whether or not to grant `current_process` the lock, and it does so by checking `current_process` against `l->curr`. If another process already holds the lock, then `current_process` is added to the waiting queue and the process's blocked field is updated to 1.

The function `lock_release()` similarly requires that all interrupts be disabled during its execution. At the highest-level, the function passes off the lock to the next process in the waiting queue and sends said process to the ready queue. The waiting queue and the blocked status of the now-ready process are accordingly updated.

A simple schematic detailing an example setup of the Arduino is presented below.

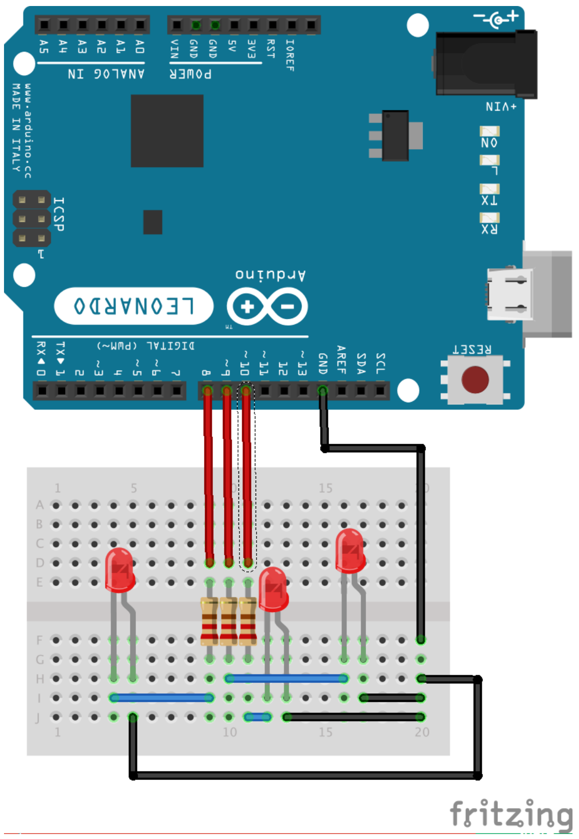

Figure 1. Schematic of Arduino Setup.