Paul Fretz
Philron Hozier
Ethan Miller
CMPSCI 377 Operating Systems
Lab 02: Part 02, Task 01
March 16, 2016

**Overview**
Our implementation of a job scheduler in C is mostly contained in one file, `part2.c`. This file relies on the neighboring files `queue.h` and `queue.c`. To run, `cd` into the `Part2-Task2/` directory and run `make`, then `./part2`. Details of our queue implementation to solve the consumer/producer problem can be found below.

**Queue Implementation Details**
The implementation for our request scheduler's circular queue was written through three tiers of abstraction. The foundation of the circular queue's implementation is a linked list. Atop the linked list layer sits a queue implementation. Specialising the queue implementation to behave as a bounded buffer produces our eventual circular queue. The implementation detail will peel away the layers and describe in kind the construction of the data structure from the ground up.

Three central typedefs which recur throughout the implementation of queue.c are `node`, `data` and `linked`. `node` is a struct which defines a node of a linked list. `linked` is a struct which defines the data structure representing a linked list. Among other important properties of `linked` is the field `head` which denotes the head of the linked list structure. `data` is a struct which defines a collection of information that a node can hold within a node's data property.

The API of the queue structure as seen in queue.h lists just five functions which provide the ability to create and manipulate a queue object. Three of the five are simple allocation functions for `node`, `data` and `linked` and as such will not be explored.

The core of the implementation resides within the `add` and `pop` functions.

Unlike a normal queue implementation of `add`, which searches for the NULL pointer through constant steps through a pointer's next field and then proceeds and link a new node -- a circular queue places a bound of some size `N` onto the queue. In words, the queue can grow to be only so big. At the queue's size limit, if all places are full, it wraps around to the head. Hence, enqueuing a node beyond size `N` of the bounded queue will result in a circular effect.

The circular behavior of our queue was achieved through calculating offsets with modulus arithmetic. The computation occurs only if the condition `if(l->counter >= l->N){...` is true. In words, if we have reached the limit of our bounded queue, then wrap around by the computed offset -- where offset is defined as `((l->counter) % l->N) +`

`l->offset_count`. `l->offset_count` is defined as a persistent tally of how many elements in the queue (beginning at the head) to cycle past before insertion occurs.

If the circular queue has not reached limit `N`, its behavior is that of a normal queue. The add function executes `while (ptr->next != NULL/* condition */) {`... instead and searches for the last non-empty node to link a new node to.

The `pop` function behavior is a standard FIFO implementation. `pop` executes a search for the head of the queue, shifts list over by one (if length of queue is greater than one) and performs unlinking and relinking. `pop` returns the current head of the queue.

The aggregation of the circular queue within part two/task one's implementation provides a robust solution to our C request scheduler.