```
 1 /*
 2  * Design:
 3  * The way I attacked this lab was by creating a master
   flagger thread which manages signaling and queuing of cars
   in the traffic circle.
 4  * The flagger would start by making sure the site was
   cleared which involved checking each car's global boolean
   varriable which
 5  * monitored if it was in the circle or not. Once it
   determined there was no cars in the circle, it started a
   loop timer that was the
 6  * duration before changing direction. In the loop it would
    check if the site was full of cars and if not, it would
   pop a car off the
 7  * queue and signal to that car though a condition
   varriable that it should start going through the site. The
   flagger would keep
 8  * repeating this cycle until all the cars finished.
 9  * In the car thread it would track waiting duration and
   number of completions locally while waiting for the
   condition varriable
10  * to be signaled. Once signaled it would sleep for the
   required duration and signal back to the flagger through
   global memory.
11  *
12  * Build Instructions: $make ./test day.txt
13  *
14  * Analysis:
15  * I believe my implementation to be as fair as possible.
   Implementing the feedback from the previous burger lab, I
   decided to
16  * create the extra overhead for car queue's on each (east
   and west) sides. This ensures cars are being popped in the
   order that
17  * they should (and as fairly as possible). When evaluating
    the actual time they waited, I feel like this is where I
   fall since
18  * there is unnessicary waiting time from the car signaling
    back to the flagger because the global memory flags were
   not as fast
19  * as I would expected. But comparing the wait times of
   each car, it's clear that they are all waiting similar
   lengths, so I
20  * guess that part it "fair".
21  * If you wanted to implement a priority queue which let
   faster cars though first, this could be done by modifying
```

```
21 the list such
22  * that as cars were pushed into the queue, the list it's
   self could inset the car into the position based on it's
   travel time.
23  * This would guarntee when popping that the fastest cars
   go first. Unfortunatly, this may cause starvation for the
   slower cars
24  * if they don't get to go though on that rotation...
25  *
26  * Conclusion:
27  * I liked the challenge of this lab. I feel that it was
   sufficiently different then the burger lab even though on
   the surface
28  * it initially seemed similar. It was for sure a fun
   challenge to get one thread to manage others.
29  * The real problem in this lab lies in it's timing. Or at
   least, I tried many strategies to get the signaling of the
   cars and
30  * signaling back to be quick so that it wouldn't be that
   noticable on the front of the direction changing or the car
    wait time.
31  * Unfortunatly, with the test case of 200us for the
   direction flipping, it was being impacted heavily by these
   outside factors.
32  * Even the time to loop around and check to see if another
    car could go though the site was ~30us for my
   implementation which
33  * is extremely significant and surly impacted the results.
34  *
35  *
36  */
37
38
39 #include <stdlib.h>
40 #include <stdio.h>
41 #include <time.h>
42 #include "llist.h"
43 #include <unistd.h>
44 #include <semaphore.h>
45 #include <pthread.h>
46 #include "ConstructionTraffic.h"
47 #include <signal.h>
48
49 // Mutex which manages top level interactions witch the car
    queue access.
50 pthread_mutex_t list_mutex;
```

```
51
52 // semaphore to manage the max number of cars in the
   traffic circle
53 int traffic_sem;
54
55 // flag to tell flagger that all the cars are done and to
   end
56 int flagger_flag;
57
58 // int array to for flagger to signal to a car that it can
   go though the traffic circle
59 // a '1' in their car's position will signal to that car
   that it can go though the traffic circle.
60 int* car_signals;
61 Signal* mutex_signals;
62
63 // public array of all cars
64 CarArray* car_array;
65
66 // queues for each cars waiting on the west and east.
67 static list westCarQueue;
68 static list eastCarQueue;
69
70 // Thread which run's the flagger
71 static void* flaggerThread(void* flagger);
72
73 // Threads which run each car
74 static void* carThread(void* car);
75
76
77 // Main method
78 int main(int argc, char* argv[]){
79
80     printf("Initializing and seting up threads...\n");
81     // initialize car lists
82     llInit(&westCarQueue);
83     llInit(&eastCarQueue);
84     flagger_flag = 0;
85
86     // create and fill car and flagger structs from file
87     Flagger flagger;
88     car_array = initTrafficCircle(argv[1], &flagger);
89
90     // setup array for IPC communication between flagger
   and car threads
91     car_signals = malloc(sizeof(int) * car_array->num_cars
```

```
 91 );
 92     mutex_signals = malloc(sizeof(Signal) * car_array->
    num_cars);
 93     for(int i = 0; i < car_array->num_cars; ++i){
 94         // initialize car signals to zero's
 95         *(car_signals + i) = 0;
 96         //sem_init(&car_signals[i], 0, 0);
 97         pthread_mutex_init(&(mutex_signals[i].m), NULL);
 98         pthread_cond_init(&(mutex_signals[i].cv), NULL);
 99         // initialize cars in their respective initial
    queues
100         car_array->cars[i].initial_side ? llPushBack(&
    eastCarQueue, i) : llPushBack(&westCarQueue, i);
101     }
102
103     // setup traffic semiphore to allow the max number of
    cars in the traffic circle
104     //sem_init(&traffic_sem, 0, flagger.car_capacity);
105     traffic_sem = 0;
106     pthread_mutex_init(&list_mutex, NULL);
107
108     // initalize thread counts for the flagger and each
    car
109     pthread_t flagger_thread;
110     pthread_t car_threads[car_array->num_cars];
111
112     printf("Starting threads!\n");
113     // create flagger thread
114     pthread_create(&flagger_thread, NULL, &flaggerThread
    , (void *)(&flagger));
115     // create car threads
116     for(int i = 0; i < car_array->num_cars; ++i)
117         pthread_create(&car_threads[i], NULL, &carThread
    , (void *)(&(car_array->cars[i])));
118
119     printf("Waiting for threads to finish.\n");
120     // wait for all cars to be done
121     for(int i = 0; i < car_array->num_cars; ++i)
122         pthread_join(car_threads[i], NULL);
123
124     printf("Day is over, time for flagger to go home.\n");
125     // stop flagger
126     flagger_flag = 1;
127     pthread_join(flagger_thread, NULL);
128
129     // print results
```

```
130     for(int i = 0; i < car_array->num_cars; ++i)
131         printf("Car %d waited a total of %d ns\n",
    car_array->cars[i].car_number, car_array->cars[i].
    drive_time);
132 
133     // cleanup memory
134     printf("Cleaning up resources\n");
135     deleteCars(car_array);
136     llClear(&westCarQueue);
137     destructList(&westCarQueue);
138     llClear(&eastCarQueue);
139     destructList(&eastCarQueue);
140 
141     return 0;
142 }
143 
144 static void* flaggerThread(void* flagger){
145     Flagger man = *((Flagger *) flagger);
146     //printf("flagger cap: %d, dir %d, flow %d\n",man.
    car_capacity, man.current_direction, man.flow_time);
147     while(1){
148         // ensure all cars are out of the traffic circle
149         int flag = 0;
150         do{
151             flag = 0;
152             for(int i = 0; i < car_array->num_cars; ++i){
153                 //int val = 0;
154                 //sem_getvalue(&car_signals[i], &val);
155                 //if(val)
156                 //  flag = 1;
157                 if(car_signals[i]){
158                     flag = 1;
159                 }
160             }
161         }while(flag);
162         //printf("all cars are cleared from intersection\n
    ");
163         long rotation = clock() * 1000000 / CLOCKS_PER_SEC
    ; // start timer
164         long end_rotation = rotation + man.flow_time; //
    needed time (in microseconds)
165         // allow traffic to flow in direction for
    designated time
166         do {
167             // check for day over
168             if(flagger_flag){
```

```
169                 pthread_exit(EXIT_SUCCESS);
170             }
171             //if there is a spot in the traffic circle,
    let the next car though and signal to the car to start
    going through
172             //int sem_val = 0;
173             //sem_getvalue(&traffic_sem, &sem_val);
174             if(traffic_sem < man.car_capacity){
175                 pthread_mutex_lock(&list_mutex);
176                 int popped_car = man.current_direction ?
    llPopFront(&eastCarQueue) : llPopFront(&westCarQueue);
177                 pthread_mutex_unlock(&list_mutex);
178                 if(popped_car != -1){
179                     car_signals[popped_car] = 1;
180                     //sem_post(&car_signals[popped_car]);
181                     traffic_sem = traffic_sem + 1;
182                     //sem_wait(&traffic_sem);
183                     pthread_cond_signal(&mutex_signals[
    popped_car].cv);
184                     //printf("signaling to car %d w/ %d
    cars in circle\n", popped_car, traffic_sem);
185                     //printf("flagger clock: %d\n", clock
    () * 1000000 / CLOCKS_PER_SEC);
186                 }
187             }
188             rotation = clock() * 1000000 / CLOCKS_PER_SEC;
189         } while (rotation <= end_rotation);
190         man.current_direction = !man.current_direction;
    // flip traffic direction
191         printf("Flagger indicating safe to drive %s \n", (
    man.current_direction == 1 ? "EAST->WEST" : "WEST->EAST"
    ));
192     }
193 }
194
195 static void* carThread(void* car){
196     Car driver = *((Car *) car);
197     //printf("car %d created\n", driver.car_number);
198     int total_crossings = 0;
199     int side = driver.initial_side;
200     while(total_crossings < driver.num_crossings){
201         //int signal = 0;
202         // wait for flagger to tell the car it can go
203         int clock_start = clock() * 1000000 /
    CLOCKS_PER_SEC;
204         //while(!signal)
```

```
205         //  signal = car_signals[driver.car_number];
206         //int val = 0;
207         //sem_getvalue(&car_signals[driver.car_number], &
    val);
208         //while(!val)
209         //  sem_getvalue(&car_signals[driver.car_number
    ], &val);
210         pthread_mutex_lock(&mutex_signals[driver.
    car_number].m);
211         //int ret = 0;
212         //do{
213         pthread_cond_wait(&mutex_signals[driver.car_number
    ].cv, &mutex_signals[driver.car_number].m);
214         //}while(!ret);
215         ((Car *) car)->drive_time = ((Car *) car)->
    drive_time + ((clock() * 1000000 / CLOCKS_PER_SEC) -
    clock_start);
216         // pass though intersection
217         printf("Car %d entering construction zone
    traveling: %s Crossings remaining: %d \n",
218                 driver.car_number, (side == 0 ? "EAST->
    WEST" : "WEST->EAST"),driver.num_crossings -
    total_crossings);
219         //printf("car %d signaled with %d. sleepint %d\n
    ", driver.car_number, car_signals[driver.car_number],
    driver.sleep_time);
220         //printf("car clock: %d\n", clock() * 1000000 /
    CLOCKS_PER_SEC);
221         usleep(driver.sleep_time);
222         // don't add car to queue if it's on it's last run
223         if(total_crossings < driver.num_crossings - 1){
224             // add car to other queue
225             side = !side;
226             pthread_mutex_lock(&list_mutex);
227             side ? llPushBack(&eastCarQueue, driver.
    car_number) : llPushBack(&westCarQueue, driver.car_number
    );
228             pthread_mutex_unlock(&list_mutex);
229         }
230         // tell flagger car is no longer in intersection
231         //printf("car %d leaving intersection\n", driver.
    car_number);
232         car_signals[driver.car_number] = 0;
233         //sem_wait(&car_signals[driver.car_number]);
234         total_crossings = total_crossings + 1;
235         traffic_sem = traffic_sem - 1;
```

```
236         //sem_post(&traffic_sem);
237         pthread_mutex_unlock(&mutex_signals[driver.
    car_number].m);
238     }
239     //printf("exiting car thread %d\n", driver.car_number
    );
240     pthread_exit(EXIT_SUCCESS);
241 }
```

```
1 4 3 20 100 3
2 10 200
3 10 50
4 8 300
5 2 100
6 20 40
7 10 50
8 30 10
```

```c
#include <stdlib.h>
#include "llist.h"
#include "string.h"
#include <stdio.h>
#include <pthread.h>

static void updateMaxSize(list* myList);

void llInit(list *myList){
    // Free these malloc's
    myList->mutex = malloc (sizeof(pthread_mutex_t));
    pthread_mutex_init((myList->mutex),NULL);
    myList->head = NULL;
    myList->tail = NULL;
    myList->size = 0;
    myList->maxSize = 0;
}

int llSize(list *myList){
    pthread_mutex_lock(myList->mutex);
    return myList->size;
    pthread_mutex_unlock(myList->mutex);
}

int llPushFront(list *myList, int value){
    //if(value != NULL){
        node* newNode = malloc (sizeof (node));
        newNode->prev = NULL;
        newNode->val = value;
        pthread_mutex_lock(myList->mutex);
        newNode->next = myList->head;
        if(myList->head != NULL){
            myList->head->prev = newNode;
        }
        myList->head = newNode;
        if(myList->tail == NULL){
            myList->tail = newNode;
        }
        (myList->size)++;
        updateMaxSize(myList);
        pthread_mutex_unlock(myList->mutex);
        return 1;
    //} else {
    //  return 0;
    //}
}
```

```
47
48 int llPopFront(list *myList){
49     if(myList->head != NULL){
50         pthread_mutex_lock(myList->mutex);
51         node* oldHead = myList->head;
52         if(oldHead->next != NULL){
53             myList->head = myList->head->next;
54             myList->head->prev = NULL;
55         } else {
56             myList->head = NULL;
57             myList->tail = NULL;
58         }
59         int ret = oldHead->val;
60         free(oldHead);
61         (myList->size)--;
62         pthread_mutex_unlock(myList->mutex);
63         return ret;
64     } else {
65         return -1;
66     }
67 }
68
69 int llPushBack(list *myList, int value){
70     //if(value != NULL){
71         node* newNode = malloc (sizeof (node));
72         newNode->val = value;
73         newNode->next = NULL;
74         pthread_mutex_lock(myList->mutex);
75         newNode->prev = myList->tail;
76         if(myList->tail != NULL){
77             myList->tail->next = newNode;
78         }
79         myList->tail = newNode;
80         if(myList->head == NULL){
81             myList->head = newNode;
82         }
83         (myList->size)++;
84         updateMaxSize(myList);
85         pthread_mutex_unlock(myList->mutex);
86         return 1;
87     //} else {
88     //  return 0;
89     //}
90 }
91
92 int llPopBack(list *myList){
```

```
 93      if(myList->tail != NULL){
 94          pthread_mutex_lock(myList->mutex);
 95          node* oldTail = myList->tail;
 96          if(oldTail->prev != NULL){
 97              myList->tail = myList->tail->prev;
 98              myList->tail->next = NULL;
 99          } else {
100              myList->tail = NULL;
101              myList->head = NULL;
102          }
103          int ret = oldTail->val;
104          free(oldTail);
105          (myList->size)--;
106          pthread_mutex_unlock(myList->mutex);
107          return ret;
108      } else {
109          return -1;
110      }
111  }
112
113  void llClear(list *myList){
114      if(myList->size != 0){
115          int status = 1;
116          while(status != 0){
117              status = llPopBack(myList);
118          }
119          myList->size = 0;
120      }
121  }
122
123  int llInsertAfter(list* myList, node *insNode, int value){
124      if((insNode == NULL) || (value == NULL)){
125          return 0;
126      } else {
127          node* newNode = malloc (sizeof (node));
128          newNode->prev = insNode;
129          newNode->val = value;
130          pthread_mutex_lock(myList->mutex);
131          if(insNode == myList->tail){
132              myList->tail->next = newNode;
133              newNode->next = NULL;
134              myList->tail = newNode;
135          } else {
136              newNode->next = insNode->next;
137              insNode->next->prev = newNode;
138              insNode->next = newNode;
```

```
139         }
140         (myList->size)++;
141         updateMaxSize(myList);
142         pthread_mutex_unlock(myList->mutex);
143         return 1;
144     }
145 }
146
147 int llInsertBefore(list* myList, node *insNode, int value
    ){
148     if((insNode == NULL) || (value == NULL)){
149         return 0;
150     } else {
151         node* newNode = malloc (sizeof (node));
152         newNode->next = insNode;
153         newNode->val = value;
154         pthread_mutex_lock(myList->mutex);
155         if(insNode == myList->head){
156             myList->head->prev = newNode;
157             newNode->prev = NULL;
158             myList->head = newNode;
159         } else {
160             newNode->prev = insNode->prev;
161             insNode->prev->next = newNode;
162             insNode->prev = newNode;
163         }
164         (myList->size)++;
165         pthread_mutex_unlock(myList->mutex);
166         return 1;
167     }
168 }
169
170 int llRemove(list* myList, node *rmvNode){
171     if(!(rmvNode == NULL)){
172         pthread_mutex_lock(myList->mutex);
173         node* workingNode = myList->head;
174         while(workingNode != NULL){
175             if(workingNode == rmvNode){
176                 if((workingNode == myList->head) && (
    workingNode == myList->tail)){
177                     myList->head = NULL;
178                     myList->tail = NULL;
179                 } else if(workingNode == myList->head){
180                     workingNode->next->prev = NULL;
181                     myList->head = workingNode->next;
182                 } else if(workingNode == myList->tail){
```

```
183                     workingNode->prev->next = NULL;
184                     myList->tail = workingNode->prev;
185                 } else {
186                     workingNode->prev->next = workingNode
    ->next;
187                     workingNode->next->prev = workingNode
    ->prev;
188                 }
189                 free(workingNode);
190                 break;
191             } else {
192                 workingNode = workingNode->next;
193             }
194         }
195         (myList->size)--;
196         pthread_mutex_unlock(myList->mutex);
197         return 1;
198     } else {
199         return 0;
200     }
201 }
202 
203 void printList(list* myList){
204     int size = 0;
205     printf("Printing List...\n");
206     pthread_mutex_lock(myList->mutex);
207     node* workingNode = myList->head;
208     while(workingNode != NULL){
209         printf("node: %d val: %d\n",size,workingNode->val
    );
210         size++;
211         workingNode = workingNode->next;
212     }
213     pthread_mutex_unlock(myList->mutex);
214 }
215 
216 void printHeadAndTail(list* myList){
217     pthread_mutex_lock(myList->mutex);
218     if(!((myList->head == NULL) || (myList->tail == NULL
    )))
219         printf("head: %d, tail: %d\n",myList->head->val,
    myList->tail->val);
220     pthread_mutex_unlock(myList->mutex);
221 }
222 
223 node* getNode(list* myList, int position){
```

```
224     pthread_mutex_lock(myList->mutex);
225     node* nodeWalker = myList->head;
226     for(int i =0;i<position;i++){
227         nodeWalker = nodeWalker->next;
228     }
229     pthread_mutex_unlock(myList->mutex);
230     return nodeWalker;
231 }
232
233 void destructList(list* myList){
234     free(myList->mutex);
235 }
236
237 /* Private helper method for keeping track of the list max
     size
238  *
239  * myList : list which is being updated
240  *
241  * return : void
242  */
243 static void updateMaxSize(list* myList){
244     if(myList->size > myList->maxSize){
245         myList->maxSize = myList->size;
246         }
247 }
248
```

```c
#ifndef LLIST_H
#define LLIST_H

#include <pthread.h>

/* llist.h
 *
 * External (public) declarations for simple doubly-linked list in C.
 *
 * This list will know head and tail, and will be capable of forward
 * and reverse iteration.  The tail node should always assign its next
 * pointer to NULL to indicate the end of the list, and likewise, the
 * head node should assign its previous pointer to NULL for the same
 * reason.
 *
 * Note that the pop operations do not return a reference to the
 * popped node.  This would require storage for the node to be
 * released by the user, which could lead to memory mishandling.
 * The user will need to use head or tail pointers to interact with the
 * node prior to popping it.
 *
 * This list will only hold strings (char arrays) for simplicity, and
 * will have ownership of the strings.  This means the string will
 * need to be copied into memory under control of the list.
 *
 */

/* Structures */

/* a node - cannot be anonymous because we need a node * inside, but
 * we can make the struct name and the typedef the same */
typedef struct node {
    char val;
    struct node* next;  /* need struct here because inside
```

```
34 typedef */
35     struct node* prev;  /* need struct here because inside
   typedef */
36 } node;
37
38 /* The list itself - struct can be anonymous */
39 typedef struct {
40     node *head;  /* could have been struct node* as well */
41     node *tail;
42     pthread_mutex_t *mutex;
43     int size;
44     int maxSize;
45 } list;
46
47 /* List methods
48  *
49  * These methods are used to create and operate on a list
   as a whole.
50  */
51
52 /* llInit()
53  *     Initialize a list structure.  An empty list will
54  *     be characterized by head and tail pointer both being
    NULL.
55  *         Parameters:  myList - a pointer to the structure
    to be init
56  *         Returns: void
57  */
58 void llInit(list *myList);
59
60 /* llSize()
61  *     Reports the current size of the list.  Will need to
   iterate
62  *     the list to get this data size there is no size
   property, nor
63  *     can there really be one given that users can access
   nodes.
64  *         Parameters:  myList - the list
65  *         Returns: int, size of list
66  */
67 int llSize(list *myList);
68
69 /* llPushFront()
70  *     Add a new node with provided data and place node at
71  *     front of list.  The new node will replace the head
   node.
```

```
 72  *     This method should check to make sure the provided
    char * is
 73  *     not NULL.  If it is NULL, this method should do
    nothing and
 74  *     make no changes to the list.  If it is not NULL, it
     can be
 75  *     assumed that it is a valid null-terminated string.
 76  *         Parameters:  myList - the list
 77  *                      toStore - the char array to store
 78  *         Returns: int - 0 if no push (toStore was NULL)
    or non-zero
 79  *                        if push successful
 80  */
 81 int llPushFront(list *myList,int value);
 82
 83 /* llPopFront()
 84  *     Removes first item in list.  Note, this does not
    return
 85  *     any data from the list.  If the data in the node is
     needed
 86  *     it should be accessed prior to the pop (list->head
    ->string).
 87  *         Parameters:  myList - the list
 88  *         Returns: int - 0 if no pop (list was empty) or
    non-zero
 89  *                        if pop successful
 90  */
 91 int llPopFront(list *myList);
 92
 93 /* llPushBack()
 94  *     Add a new node with provided data and place node at
 95  *     end of list.  This new node will be the new tail
    node.
 96  *     This method should check to make sure the provided
    char * is
 97  *     not NULL.  If it is NULL, this method should do
    nothing and
 98  *     make no changes to the list.  If it is not NULL, it
     can be
 99  *     assumed that it is a valid null-terminated string.
100  *         Parameters:  myList - the list
101  *                      toStore - the char array to store
102  *         Returns: int - 0 if no push (toStore was NULL)
    or non-zero
103  *                        if push successful
104  */
```

```
105 int llPushBack(list *myList, int value);
106
107 /* llPopBack()
108  *     Removes last item in list.  Note, this does not
    return
109  *     any data from the list.  If the data in the node is
     needed
110  *     it should be accessed prior to the pop (list->tail
    ->string).
111  *         Parameters:  myList - the list
112  *         Returns: int - 0 if no pop (list was empty) or
    non-zero
113  *                        if pop successful
114  */
115 int llPopBack(list *myList);
116
117 /* llClear()
118  *     Clears all nodes and releases all dynamic memory.
    List
119  *     structure should be NULLed and can be reused.
120  *         Parameters:  myList - the list
121  *         Returns: nothing
122  */
123 void llClear(list *myList);
124
125
126 /* Node methods
127  *
128  * These methods allow iteration of nodes within the list
    .  A list
129  * reference is still needed if head or tail needs to be
    modified.
130  */
131
132 /* llInsertAfter()
133  *     Add a new node with provided data and place node
    after
134  *     provided node reference.
135  *     This method should check to make sure the provided
    char * is
136  *     not NULL.  If it is NULL, this method should do
    nothing and
137  *     make no changes to the list.  If it is not NULL, it
     can be
138  *     assumed that it is a valid null-terminated string.
139  *     If this method is called on the tail node, a change
```

```
139  to the
140  *     list structure will need to be made.
141  *         Parameters:  myList - theList
142  *                      insNode - the node after which
    item is added
143  *                      toStore - the char array to store
144  *         Returns: int - 0 if no insert (toStore was NULL
     or insNode
145  *                                        is NULL)
146  *                        non-zero if insert successful
147  */
148 int llInsertAfter(list* myList, node *insNode, int value);
149
150 /* llInsertBefore()
151  *     Add a new node with provided data and place node
    before
152  *     provided node reference.
153  *     This method should check to make sure the provided
    char * is
154  *     not NULL.  If it is NULL, this method should do
    nothing and
155  *     make no changes to the list.  If it is not NULL, it
     can be
156  *     assumed that it is a valid null-terminated string.
157  *     If this method is called on the head node, a change
     to the
158  *     list structure will need to be made.
159  *         Parameters:  myList - theList
160  *                      insNode - the node before which
    item is added
161  *                      toStore - the char array to store
162  *         Returns: int - 0 if no insert (toStore was NULL
     or insNode
163  *                                        is NULL)
164  *                        non-zero if insert successful
165  */
166 int llInsertBefore(list* myList, node *insNode, int value
    );
167
168 /* llRemove()
169  *     Removes the node referenced.  Releases
170  *     all associated dynamic memory.
171  *     If this method is called the current head or tail
    node, changes
172  *     to the list structure may need to be made.
173  *         Parameters:  myList - the list
```

```
174  *                        rmvNode - the node prior to the
     node to be
175  *                                  removed.
176  *         Returns: nothing
177  */
178 int llRemove(list* myList, node *rmvNode);
179
180 /*
181  * Helper method which prints the list's contents to the
     console
182  *
183  * myList : pointer to the list which should be printed.
184  *
185  * return : void
186  */
187 void printList(list* myList);
188
189 /*
190  * Helper method which prints the list's head and tail to
     the console
191  * to verify the head and tail are what they should be.
192  *
193  * myList : pointer to the list which should be printed.
194  *
195  * return : void
196  */
197 void printHeadAndTail(list* myList);
198
199 /*
200  * Helper method which get's a node pointer for a supplied
      position in
201  * the linked list.
202  *
203  * myList : pointer to the list which should be printed.
204  * position : valid position in the list for which you
     would like the pointer of
205  *
206  * return : pointer to the node at position in supplid
     myList
207  */
208 node* getNode(list* myList, int position);
209
210 /*
211  * Method which "cleans up" the list before destructing it
     .
212  * Free's the memory associated with the internal mutex.
```

```
213  *
214  * myList : pointer to the list
215  */
216 void destructList(list* myList);
217
218 #endif
219
```

```
# gnu c compiler
#CC = gcc

# `Wall`    - warns about questionable things
# `Werror`  - makes all warnings errors
# `Wextra`  - enables some extra warning flags that `all` doesn't set
# `Wunused` - complains about any variable, function, label, etc. not being used
#CFLAGS = -Wall -Werror -Wextra -Wunused
# `g`           - generate source code debug info
# `std=`        - sets the language standard, in this case c99
# `_GNU_SOURCE` - is a macro that tells the compiler to use rsome gnu functions
# `pthred`      - adds support for multithreading with the pthreads lib (for preprocessor
#                 and linker)
# `O3`          - the level of optimization
#CFLAGS += -g -std=c99 -D_GNU_SOURCE -pthread -O3
# `-I` - adds directory to the system search path (for include files)
#CFLAGS += -I"$(INCDIR)"


CC=gcc
CFLAGS=-c -Wall -pthread
LDFLAGS=-pthread
SOURCES=llist.c Main.c ConstructionTraffic.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=test

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.c.o:
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -rf $(OBJECTS) $(EXECUTABLE)

```

File - C:\lwshare\Lab7\BurgerPlace.c

```c
// standard libraries
#include <stdlib.h>
#include <stdio.h>

#include <pthread.h> // for threading
#include <sys/types.h> // for wait, etc.
#include <sys/wait.h> // for wait, etc.
#include <unistd.h> // fork, sleep, etc.

// user defined
#include "llist.h"
#include "string.h"
#include "BurgerPlace.h"

CustomerArray* initBurgerPlace(const char* filename, BurgerCooks* burger_cooks, Fryers* fryers){
    FILE* input = fopen(filename, "r");
    // start reading the file
    if(input != NULL){
        // fill the burger_cooks and fryers
        fscanf(input, "%d", &(burger_cooks->number_cooks));
        fscanf(input, "%d", &(burger_cooks->cook_time));
        fscanf(input, "%d", &(burger_cooks->total_servings));
        fscanf(input, "%d", &(fryers->number_cooks));
        fscanf(input, "%d", &(fryers->cook_time));
        fscanf(input, "%d", &(fryers->total_servings));
        // create and fill the customer array with each following customer
        CustomerArray* customer_array = malloc (sizeof(CustomerArray));
        fscanf(input, "%d", &(customer_array->num_customers));
        customer_array->customers = malloc (sizeof(Customer)*customer_array->num_customers);
        for(int i = 0; i < customer_array->num_customers; ++i){
            fscanf(input, "%d", &((customer_array->customers + i)->burgers));
            fscanf(input, "%d", &((customer_array->customers + i)->fries));
            fscanf(input, "%d", &((customer_array->customers + i)->duration));
            (customer_array->customers + i)->ordersFilled = 0;
        }
```

```
36          //close the file and return the resulting customer
    array
37          fclose(input);
38          return customer_array;
39      }
40      return NULL;
41  }
42
43  void deleteCustomers(CustomerArray* customer_array){
44      free(customer_array->customers);
45      free(customer_array);
46  }
47
```

File - C:\lwshare\Lab7\ConstructionTraffic.c

```c
// standard libraries
#include <stdlib.h>
#include <stdio.h>

// user defined
#include "llist.h"
#include "string.h"
#include "ConstructionTraffic.h"

CarArray* initTrafficCircle(const char* filename, Flagger* flagger){
    FILE* input = fopen(filename, "r");
    // start reading the file
    if(input != NULL){
        int west_cars, east_cars, drive_time = 0;
        fscanf(input, "%d", &west_cars);
        fscanf(input, "%d", &east_cars);
        fscanf(input, "%d", &drive_time);
        fscanf(input, "%d", &(flagger->flow_time));
        fscanf(input, "%d", &(flagger->car_capacity));
        flagger->current_direction = 0;
        // create and fill the car array
        CarArray* car_array = malloc (sizeof(CarArray));
        car_array->num_cars = (west_cars + east_cars);
        car_array->cars = malloc(sizeof(Car) * car_array->num_cars);
        for(int i = 0; i < car_array->num_cars; ++i){
            fscanf(input, "%d", &((car_array->cars + i)->num_crossings));
            fscanf(input, "%d", &((car_array->cars + i)->sleep_time));
            (car_array->cars + i)->car_number = i;
            (car_array->cars + i)->drive_time = 0;
            if(i < west_cars){
                (car_array->cars + i)->initial_side = 0;
            } else {
                (car_array->cars + i)->initial_side = 1;
            }
        }
        //close the file and return the resulting customer array
        fclose(input);
        return car_array;
    }
    return NULL;
}
```

```
42
43 void deleteCars(CarArray* car_array){
44     free(car_array->cars);
45     free(car_array);
46 }
47
```

File - C:\lwshare\Lab7\ConstructionTraffic.h

```h
#ifndef CONTRA_H
#define CONTRA_H
#include <pthread.h>

typedef struct Flagger{
    int flow_time;
    int car_capacity;
    int current_direction;
} Flagger;

/*
 *
 * initial_side : boolean value. 0 if west, 1 if east.
 */
typedef struct Car{
    int drive_time;
    int initial_side;
    int num_crossings;
    int sleep_time;
    int car_number;
} Car;

typedef struct CarArray{
    Car* cars;
    int num_cars;
} CarArray;

typedef struct Signal {
    pthread_mutex_t m;
    pthread_cond_t cv;
} Signal;

CarArray* initTrafficCircle(const char* filename, Flagger*
flagger);

void deleteCars(CarArray* car_array);

#endif
```