```c
/*
 * Implement a solution to the bounded buffer problem
 *  spawn off num_prods producers and num_consumers, each
 *  producer tid produces num_items items, each consumer tid
 *  consumes num_items items
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// use global variables for shared buffer state
static char *buff;          // the buffer
static int  N;              // total capacity
static int  size;           // current num elements
static int  next_in;        // next insertion index
static int  next_out;       // next remove index
static int  num_items;      // number of items each tid should produce or consume

//sync primitives here
pthread_cond_t full  = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mux  = PTHREAD_MUTEX_INITIALIZER;



// the thread main routine must match this prototype:
void *producer(void *arg);
void *consumer(void *arg);



// these functions are implemented for you
static void *my_malloc(size_t size);
static void add_to_queue(char item);
static char remove_from_queue(void);
static void print_buffer(void);     // for debugging
```

```
/*****************************************************************/
int main(int argc, char **argv) {
  pthread_t *ptids, *ctids;
  int i, num_prods, num_cons;
  //TODO: add any local variables you need




  num_prods = num_cons = atoi(argv[1]);
  num_items = atoi(argv[2]);
  N = atoi(argv[3]);
  printf("%d Producer & Consumer tids, each producing %d items, buff size %d\n",
     num_prods, num_items, N);
  ptids = (pthread_t *)my_malloc(sizeof(pthread_t)*num_prods);
  ctids = (pthread_t *)my_malloc(sizeof(pthread_t)*num_cons);
  buff = (char *)my_malloc(sizeof(char)*N);
  size = 0;
  next_in = 0;
  next_out = 0;
  // TODO:  create the producer and consumer threads
  //       initialize any synchronization primitives prior to this
  for (i=0; i < num_prods; i++) {




  }
  for (i=0; i < num_cons; i++) {




  }
```

```
// wait for threads to exit
  for (i=0; i < num_prods; i++) {
    pthread_join(ptids[i], 0);
  }
  for (i=0; i < num_cons; i++) {
    pthread_join(ctids[i], 0);
  }



  free ((char *)buff);
  free ((pthread_t *)ptids);
  free ((pthread_t *)ctids);
  exit(0);
}
/*****************************************************************/
void *producer(void *arg){
  // TODO: implement this  (feel free to change the return value)









  return NULL;
}
```

```
/*****************************************************************/
void *consumer(void *arg){
  // TODO: implement this

  return NULL;
}

/*****************************************************************/
//
// Add an item to the circular buffer
// note: this function does no checking that there is enough
// space to add, the caller is responsible for that
//   item: the value to add
//
static void add_to_queue(char item) {
  buff[next_in] = item;
  next_in = (next_in + 1) % N;
  size += 1;
}
```

```
/*****************************************************************/
//
// Remove an item to the circular buffer
// note: this function does no checking that there is something
// in the queue to remove, the caller is responsible for that
//   returns: the next item to remove
//
static char remove_from_queue() {
  char item;
  item = buff[next_out];
  next_out = (next_out + 1) % N;
  size -= 1;
  return item;
}



/*****************************************************************/
// print out the contents of the buffer
static void print_buffer() {
  int i, index;
  printf("Buffer size %d ************************\n", size);
  for(i=0; i < size; i++) {
     index = (next_out + i) % N;
     printf("%2d:%2d ", index, buff[index]);
  }
  printf("\n next in = %d  next out %d\n", next_in, next_out);
  printf("****************************************\n");
}



/*****************************************************************/
// a wrapper function around malloc that calls perror and exit
// on error
static void *my_malloc(size_t size){
  void *ret;
  ret = malloc(size);
  if(!ret) {
    perror("malloc array error");
    exit(1);
  }
  return ret;
}
```

pthread Synchronization

Mutex: mutually exclusive access

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_[un]lock(&mutex);
```

Condition variables: block if condition is true

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
pthread_cond_wait(&cond, &mux);
pthread_cond_signal(&cond);
```

Barrier mutex: block until all threads have called wait

```
pthread_barrier_t mybarrier;
pthread_barrier_init(&mybarrier,NULL,numtids);
Pthread_barrier_wait(&mybarrier);
```