QUESTION 6

For this question you are given the following definitions and functions:

```
thread_start_fp; // a function pointer type to a
                 // function that takes a (void*) as
                 // argument and returns a (void*)
thread_t;        // a thread type

// A function that, given an allocated thread_t
// pointer, a function pointer, and arguments,
// schedules a new thread
thread_create(thread_t * thread,
              thread_start_fp * fp,
              void * args);

// A function that performs a “join” on a given
// thread_t
thread_join(thread_t * thread);

mutex_t;         // a mutex type
cond_t;          // a condition variable type
// You can assume with both of these that they get
// initialized for you after you have declared them

// A function that acquires a mutex
mutex_acquire(mutex_t * mutex);
// A function that releases a mutex
mutex_release(mutex_t * mutex);
// A function that “waits” on a given condition
condition_wait(cond_t * cond, mutex_t * mutex);
// A function that “signals” a given condition
condition_signal(cond_t * cond);
// A function that “broadcasts” on a given condition
condition_broadcast(cond_t * cond);

// A blocking function that may take arbitrarily long
// to complete
BlockingNetworkCall();
```

You will design two functions. One will be a worker function and the other will create threads that run this worker function:

A. Design a function called “`worker`.” This function will be the body of the threads that you will be creating. This function must do the following
   - Get an ID (some number), as well as a callback (some function pointer) from its arguments
   - The callback should be a function that takes an `int` and returns `void`
   - It must call the “`BlockingNetworkCall()`” function
   - After “`BlockingNetworkCall()`” returns, it must call the callback with the thread ID as argument
   - Finally, it must return

B. Design a function called “`schedule_workers`.” The purpose of this function is to create the worker threads. It must do the following:
   - Get a callback and an integer num_threads as an argument
   - Create num_threads worker threads with a unique ids (assume a “`GetUniqueId()`” function)
   - Return when all the threads are done

HINTS: You will need to define two new types – these should be useful in passing arguments to your workers. Remember to follow standard C coding practices and conventions.

Sample Solution:

```
// The new types
typedef void (*callback_fp)(int);

struct start_info {
  int id;
  callback_fp callback;
};
typedef struct start_info start_info_t;

// The worker thread body
void* worker(void *arg) {
  start_info_t *start = (start_info_t*)arg;
  BlockingNetworkCall();
  start->callback(start->id);
  free(start);
}

// The function that runs the workers and waits for them
void schedule_workers(callback_fp callback,
                      int num_threads) {
  thread_t * threads =
      (thread_t*)malloc(num_threads * sizeof(thread_t));
  // create all the threads
  for (int i = 0; i < num_threads; ++i) {
    start_info_t *start =
        (start_info_t*)malloc(sizeof(start_info_t));
    start->callback = callback;
    start->id = GetUniqueId();
    thread_create(&thread[i], &worker, (void*)start);
  }

  // join on all the threads
  for (int i = 0; i < num_threads; ++i) {
    thread_join(&thread[i]);
  }

  free(threads);
}
```

C. Now, assume that the "`BlockingNetworkCall()`" function uses a limited underlying resource (for example, network connection channels.) Assume that the maximum number of available channels is given to you by the constant `MAX_CHANNELS` and assume that "`BlockingNetworkCall()`" uses exactly one channel. Redesign/enhance your worker function to use the given synchronization primitives to make sure that `BlockingNetworkCall` is never called if there are no available channels, and that it always executes as soon as possible when there are available channels.

Sample Solution:

```
int available_channels = MAX_CHANNELS;
mutex_t mutex;
cond_t available_cond;

void* worker(void *arg) {
  start_info_t *start = (start_info_t*)arg

  mutex_lock(&mutex);
  while(available_channels <= 0) {
    cond_wait(&available_cond, &mutex);
  }
  // We are guaranteed to hold the mutex and have channels
  --available_channels; // use a channel
  mutex_unlock(&mutex);

  // Make the long blocking call outside critical sections
  BlockingNetworkCall();

  mutex_lock(&mutex);
  ++available_channels; // relinquish a channel
  mutex_unlock(&mutex);
  cond_signal(&available_cond); // let a waiting thread in

  free(start);
}
```