A common design pattern in systems programming is to use a thread pool to farm out the processing of incoming tasks and scale up to use as many processors as available. Each task is enqueued into a queue and processed by a worker thread.

```
typedef struct Task {
    void (*func)(void *);
    void *arg;
} Task;

Task q[QUEUE_SIZE];
int in = 0, out = 0, count = 0;
bool exitRequested = false;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void
enqueue(void (*func)(void *), void *arg) {
    Task *t = malloc(sizeof(*t));

    t->func = func;
    t->arg = arg;

    q[in] = t;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    pthread_cond_signal(&c);

}

void
workerthread(void *ignored) {
    for (;;) {
        pthread_mutex_lock(&m);

        if (count == 0)
            pthread_cond_wait(&c, &m);
        if (exitRequested)
            pthread_exit(NULL);

        Task *t = q[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        pthread_mutex_unlock(&m);

        t->func(t->arg);

        free(t);
    }
}
```

Question 1. Does the program have any data races? If so, explain the data race and fix the code.

Question 2. Are the condition variables used correctly? If not, explain the bug(s) and fix the code.

Question 3. Write a function to terminate the worker pool. Hint: You may need to fix the code to left.