CS 466, Fall 2023
Assignment #3
Orderly Queue

Due 10/20/23, in class

1. Problem statement

Mergesort is an algorithm that can be easily adapted to use multi-threading to speed up the sorting. In this assignment, you will write two mergesort implementations: one without multi-threading and one that uses a shared task list to operate with any number of threads.

2. Specifications

You must work with a partner on this assignment, but cannot work with anyone who is in your security project group.

2.1. Mergesort

Mergesort divides its input array in half, sorts the halves, and then merges the two sorted halves together.

```
MERGESORT(A)
  if LENGTH(A) > 1 then
    B = MERGESORT(first half of A)
    C = MERGESORT(second half of A)
    A = MERGE(B, C)
  return A
```

2.2. Functions of `mergesort.c`

`void mergesort(int *arr, int num_elem)`

This function uses the mergesort algorithm to sort the first `num_elem` elements of the array `arr`. If `arr` is `NULL`, the function should just return. Be aware that the merge step needs to create a deep copy of the array before merging them back into the original location.

`void mt_mergesort(int *arr, int num_elem, int chunk_size, int thread_ct)`

This function splits the array `arr` into chunks of `chunk_size` integers (via pointers, not copying) and places them on the task list (see below). It then starts `thread_ct` worker threads (whose operation is described below). The main thread waits for the workers to terminate (using `pthread_join()`). It does not do any of the actual sorting (i.e., there are `thread_ct + 1` total threads: `thread_ct` worker threads and one main thread).

3. Task List

Your implementation must use a task list, which is a shared list containing pieces of work. Each thread repeatedly pulls unsorted chunks from the list and puts sorted chunks back on the list. (A Task List is sometimes referred to as a Task Queue or a Work List.)

Chunks

Chunks have a start (index of the first contained element) and a size (number of elements). Each chunk must be properly aligned, meaning that it's start must always be a multiple of its size.

Buddies

Each chunk has a buddy which is the adjoining chunk of the same size that, when merged, forms a *properly-aligned* chunk twice as large. (Think about which mergings alignment allows and which it prevents.)

Worker Interaction

Each worker repeatedly does the following:

1. Remove the first unsorted chunk from the task list.
2. Sort the chunk using `mergesort()` and mark the chunk as ''sorted''.
3. Repeatedly merge the chunk with it's buddy if the buddy is marked sorted, then return the resulting chunk to the list.
4. Remove the next unsorted chunk form the task list.
5. If there are no more unsorted chunks in the list, the thread exits via `pthread_exit()`. (Note that when two chunks are merged into a third, the first two are removed from the task list and free'd. The third is placed in the list unless it is immediately merged with its buddy.)

4. Testing program

Your executable, `mergetest`, should take three integer arguments on its command line. The first is the number of integers to be sorted, which is given as a power of two (e.g., 10 would create an array of 2^{10} elements). The second is the seed for the random number generator used to generate the initial array of numbers. The third is the maximum number of threads to use. You may assume that the numeric values of all three arguments can be stored in a signed int without any loss of data.

After initializing the array, your program should proceed to sort it using first the non-threaded mergesort and then the multi-threaded with one through the max threads. Make sure to sort a copy of the original array each time, as sorting already sorted data will invalidate your results. After each sort, you should report to standard output the time spent sorting, in terms of wall clock time, user time, and system time, in the format shown below.

```
0 threads: 0.751018s wall; 0.735888s user; 0.002999s sys
1 threads: 0.774871s wall; 0.779882s user; 0.000000s sys
2 threads: 0.417598s wall; 0.781881s user; 0.003000s sys
...
```

For your own testing purposes, you should support using `gcc -DCHECK`, which checks that each sort produces the correctly sorted array.

What to hand in

An updated repo with your nicely formatted source code and the questions in `README.md` answered.

5. Important Points and Hints

(1) You may use global variables for the head of the task list and any locks.

(2) If `thread_ct` is less than one, or greater than 1024, or chunk size is less than 1 or greater than 2^{30}, the function `mt_mergesort()` should just return.

(3) The input array size will always be a power of two and a multiple of the chunk size. In other words, your ''buddy tree'' will be a *full binary tree.*

(4) The list is shared among all threads and hence must be protected by appropriate synchronization. You can use the `PTHREAD_MUTEX_INITIALIZER` default initialization.

(5) You may find writing a `merge()` helper function useful.

(6) Compile using the makefile, which uses ''`gcc -pedantic-errors -Wall -Werror ...`'' to force your to produce error and warning free code.