```
/*
 * ========================================================================
 *       Filename:  mergesort.c
 *        Version:  1.0
 *        Created:  2015/10/20
 *         Author:  Shuaiqi Cao
========================================================================
 */
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

int get_round(int length);
void *Merge(void*args);

void print_array(int length);

int number_array[4096];
/*create a global variable, integer barrier,
as a counter of current threads in each round*/
int barrier;
/*create a mutex and a condition using the POSIX API*/
pthread_mutex_t mutex;
pthread_cond_t cond;
pthread_t tid[4096];

typedef struct args{
        int left;
        int mid;
        int right;
}args;

#define DELIM " \t\r\n\a"
#define READIN_BUF_SIZE 20000
int main(int argc, char *argv[])
{
        /*Read in the file, and store the integers in array*/
        FILE *stream;
        stream = fopen("indata.txt","r");
        char input[READIN_BUF_SIZE];
        fgets(input,READIN_BUF_SIZE,stream);

        char *number;
        int i=0;
        number = strtok(input, DELIM);
```

```
while(number!=NULL){
        number_array[i] = atoi(number);
        i++;
        number = strtok(NULL, DELIM);
}

/*Initial the mutex and condition*/
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond, NULL);

/*Get length. Calculate the number of rounds
Call for loop to execute the rounds loop*/
int length=i;
int round=get_round(length);
int round_count;
int subset_length=2;
int subset_num=length;
int tid_ptr=0;
for(round_count=0;round_count<round;round_count++){

        /*Get the thread number in this round,
        and set the barrier as this number*/
        int thread_number=subset_num/2;
        barrier=thread_number;

        /*Create struct args, and assign parameters to them*/
        args arguements[thread_number];

        int subset;
        for(subset=0;subset<thread_number;subset++){
                arguements[subset].left=subset_length*subset;
                arguements[subset].mid=(subset_length/2-1)+subset_length*subset;
                arguements[subset].right=(subset_length-1)+subset_length*subset;
        }

        /*Create threads*/
        for(subset=0;subset<thread_number;subset++){
                pthread_attr_t attr;
                pthread_attr_init(&attr);
                pthread_create(&tid[tid_ptr],&attr,Merge,(void*)&arguements[subset]);
                tid_ptr++;
        }

        /*Set a barrier here to block the main thread, once all threads in
        this round are completed, it will be released. Since the barrier would
        be used as a condition here, a lock is applied to protect this section*/
        pthread_mutex_lock(&mutex);
        while(barrier>0){
                pthread_cond_wait(&cond,&mutex);
```

```
                }pthread_cond_broadcast(&cond);
                pthread_mutex_unlock(&mutex);

                /*update the round information*/
                subset_num=subset_num/2;
                subset_length=subset_length*2;
        }

        print_array(length);
        return 0;
}


int get_round(int length){
        int l=length, round=0;
        while (l!=1){
                l=l/2;
                round++;
        }
        return round;
}

void *Merge(void*args)
{
        struct args* arguements=(struct args*)args;
        int left=arguements->left;
        int mid=arguements->mid;
        int right=arguements->right;

        /*Use dynamic array to store the intermediate results*/
        int *tempArray=(int *)malloc((right-left+1)*sizeof(int));
      int pos=0,lpos = left,rpos = mid + 1;

      while(lpos <= mid && rpos <= right){
                if(number_array[lpos] < number_array[rpos]){
                 tempArray[pos++] = number_array[lpos++];
          }else{
                 tempArray[pos++] = number_array[rpos++];
          }
      }
      while(lpos <= mid){
                tempArray[pos++] = number_array[lpos++];
        }
      while(rpos <= right){
                tempArray[pos++] = number_array[rpos++];
        }
      int iter;
      for(iter = 0;iter < pos; iter++){
           number_array[iter+left] = tempArray[iter];
```

```
        }

            /*free the intermediate results once it will not be used*/
            free(tempArray);

            /*Set a barrier here to block this single thread, once all threads in
            this round are completed, the value of barrier would be 0. This thread
            would be released. Since the change of barrier and comparaion happens here,
            a mutex lock is applied to protect this section*/
            pthread_mutex_lock(&mutex);
            barrier--;
            while(barrier>0){
                    pthread_cond_wait(&cond,&mutex);
            }
            pthread_cond_broadcast(&cond);
            pthread_mutex_unlock(&mutex);

        return;
}

void print_array(int length)
{
            int index;
            for(index=0; index<length; index++){
                    printf("%d ", number_array[index]);
            }
            printf("\n");
}
```