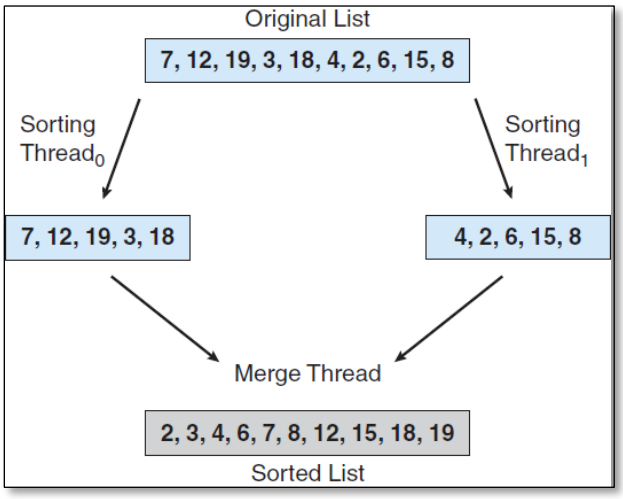
CPSC 351, Operating Systems Concepts

Homework, Threads

Inspired by Chapter 4 Programming Project #2 in the textbook

1) (*20 pts*) Write a POSIX multi-threaded sorting program (in C) called sorting.c that works as follows: A list of integers is divided into two smaller lists of (nearly) equal size. Two separate worker threads (which we will term sorting threads) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third worker thread - a merging thread - which merges the two sorted sublists into a single sorted list.

   Because we are striving for Data Parallelism (same algorithm on different data) and because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global (file scoped) array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured as show on the right.



   This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting address from which each thread is to begin sorting, and the number of items to be sorted. See Hints below.

   The parent thread will output the sorted array once all sorting has completed.

   Other information:

   1. sorting.c has been provided to get you started
   2. The list of integers to be sorted is (use the exact values and order provided below):
      `{5, 10, 17, 1, 16, 2, 0, -7, 4, 13, 6}`
   3. Pass to each sorting thread a pointer to an element within the original list and the number of elements to be sorted. The contents of the original list must not be moved or copied before invoking the sorting threads. Each half of the original list is to be sorted "in-place". For example, if the original list `L` is initialized to `{G, F, C, J, B}`, then two sorting thread will be created, one with `(&L[0], 3)` and the other with `(&L[3], 2)` as passed data. When both sorting threads complete, the list `L` will then contain `{C, F, G, B, J}`. Notice each half of the original list has been sorted. Remember not to wait for one sorting thread to complete before starting the other, and that the merging thread cannot be started until both sorting threads have completed.
   4. Use the structures for passing data to worker threads as provided in the hints below.
   5. Using the Build.sh script from your first homework, compile and link your program like this:
      `./Build.sh sorting`
   6. `./sorting | tee sorting_output.txt` is an example command to run your program while collecting your program's output to a text file and seeing the output on the console.
   7. Submit (only) the following 2 files to Canvas.
      - sorting.c, sorting_output.txt (both files must be present to earn credit)

Hints:

1) The parent thread initializes an unsorted array with integer numbers. The easiest approach for sharing data is to create global data. For example, at file scope:

```
#define SIZE           ( sizeof(list)/sizeof(*list) )
int     list[]       = {5, 10, 17, 1, 16, 2, 0, -7, 4, 13, 6}; // array initially filled with unsorted numbers
int     result[SIZE] = {0};                                     // same contents as unsortedarray, but sorted
```

2) The parent thread then creates sorter threads, passing each a parameter containing the location (pointer) and size (unsigned int) of the unsorted array. The easiest approach for passing data to threads is to create a data structure using a `typedef` and `struct`. For example,

```
/* structures for passing data to worker threads */
typedef struct
{
  int *        subArray;
  unsigned int size;
} SortingThreadParameters;

typedef struct
{
  SortingThreadParameters left;
  SortingThreadParameters right;
} MergingThreadParameters;
```

3) The parent creates worker threads using a strategy similar to:

```
SortingThreadParameters * paramsLeft = malloc( sizeof( SortingThreadParameters ) );
paramsLeft->subArray                 = list;
paramsLeft->size                     = SIZE/2;
/* Now create the first sorting thread passing it paramsLeft as a parameter */
...

SortingThreadParameters * paramsRight = malloc( sizeof( SortingThreadParameters ) );
paramsRight->subArray                  = list + paramsLeft->size;
paramsRight->size                      = SIZE - paramsLeft->size;  // taking difference addresses an add number of elements
/* Now create the second sorting thread passing it paramsRight as a parameter */
...


// wait for the sorting threads to complete
...

MergingThreadParameters * paramsMerge = malloc( sizeof( MergingThreadParameters ) );
paramsMerge->left  = *paramsLeft;
paramsMerge->right = *paramsRight;
/* Now create the merging thread passing it paramsMerge as a parameter */
...
```

4) The pointer-to-parameter (e.g. `paramsLeft`, `paramsRight`, and `paramsMerge`) is passed to the `pthread_create()` function, which in turn is passed as a parameter to the function that is to run as a separate thread.

5) The sorting threads perform an in-place sort. The merging thread does not.