1. Introduction

In this assignment you are going to practice on synchronization between threads. Your task is to implement parallel merge sort on binary tree structure with threads. You will use semaphores and monitors for the synchronization between threads. There will be two implementations. The first one is to implement with semaphores. The second one is to implement with monitors.

2. Merge Sort

Sorting is a common and important problem in computing. Given a sequence of $N$ data elements, we are required to generate an ordered sequence that contains the same elements. Merge sort is a comparison-based sorting and divide-and-conquer algorithm. Conceptually, a merge sort works as follows:

- Divide the unsorted list into $N$ sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

![Figure 1: An example of merge sort](image)
Algorithmically, the sequential merge sort algorithm is as follows:

1. If the input sequence has fewer than two elements, return.
2. Partition the input sequence into two halves.
3. Sort the two subsequences using the same algorithm.
4. Merge the two sorted subsequences to form the output sequence.

The merge operation employed in step (4) combines two sorted subsequences to produce a single sorted sequence. It repeatedly compares the heads of the two subsequences and outputs the lesser value until no elements remain.

3. Parallel Merge Sort

In this section we present the parallel version of the merge sort on binary tree structure. The algorithm assumes that the sequence to be sorted is distributed and so generates a distributed sorted sequence. For simplicity, we assume that $N$ is an integer multiple of $P$, where $P$ is the number of threads. At the beginning, the $N$ data are distributed evenly among the $P$ threads. And we also assume that $P = 2^d$, where $d$ is the depth of tree.

We first describe two algorithms required in the implementation of parallel merge sort: compare-exchange and parallel merge.

3.1 Compare-Exchange

A compare-exchange operation merges two sorted sequences of length $M$ which are contained in Task A and Task B. After operation both task have $M$ data, all elements in task A is less than or equal to all elements in task B. An example is given in Figure 2.

In part a, task A and task B send their data to each other. In part b, they perform a merge operation to identify the lowest and highest $M$ elements, respectively. In part c, task A identifies the $M$ lowest

![Diagram of compare-exchange algorithm with $M=4$]
elements and discards the remainder; this process requires at least $M/2$ and at most $M$ comparisons. Similarly, task B identifies the $M$ highest elements.

### 3.2 Parallel Merge

A parallel merge algorithm performs a merge operation on two sorted sequences of length $M$ of two tasks. It determines that whether the current task gets the lowest $M$ elements after merge operation or gets the highest $M$ elements. And then call the appropriate compare-exchange function. Following operator is done at each run:

```c
If ( (myid AND 2^dim) > 0) then
    new_data = compare_exchange_high(state,message);
else
    new_data = compare_exchange_low(state,message);
endif
```

where `myid` is the id of the current thread, `state` is the partial data of the current thread, `message` is the partial data of second thread which is used in the compare-exchange operation, `new_data` is the result of the compare-exchange operation and `dim` is the current iteration number.

**Hint:** Compare-exchange process requires at least $M/2$ and at most $M$ comparisons. That is, in compare_exchange_low function it repeatedly compares the heads of the two subsequences and outputs the lesser value until $M$ elements remain and in compare_exchange_high function it repeatedly compares the tails of the two subsequences and outputs the higher value until $M$ elements remain.

### 3.3 Parallel Merge Sort with Binary Tree Template

![Diagram of parallel merge sort on binary tree](image)

Figure 3: An Example of parallel merge sort on binary tree with $P = 8$
At the beginning of the algorithm eight threads get the partial data from unsorted data sequences and sort their own data with a sequential sorting algorithm. After parallel merge operation, the left one from the two threads is continue in the next iteration. There are task dependencies between the levels of binary tree. For example: P0, P1, P2 and P3 should do their parallel merge operation and combine their data in order to be done parallel merge of the next level. There are also interaction dependencies between the threads on the same level. For example: Both P0 and P1 should sort their data (this is only needed once for all the threads at the beginning of the algorithm) and send them to each other before the parallel merge operation in the first iteration. On the next iteration, both P0 and P2 should send their data to each other before the parallel merge operation. Each thread in the computation executes the following logic:

```
procedure parallel_mergesort(thread_id)
begin
  var partial_data, n
  partial_data = sequential_sort(partial_data)
  for dim = 0 < d
    new_data = parallel_merge(thread_id, dim, partial_data, n)
  endfor
end
```

where `partial_data` is the own data of each thread which is obtained from the initial unsorted data sequences and `n` is the size of current `partial_data` which is increased at each iteration. Each thread runs the loop and calls `parallel_merge` function at most `d` times. That is, for example: P0 runs the loop `d` times whereas P2 runs `d − 1` times and P5 runs `d − 2` times.

4. An Example Run

Below an example of parallel merge sort is given. The data.txt file is as follows:

```
16
8
10
3
14
7
12
11
5
6
1
2
4
9
15
13
8
16
```

Figure 4: Sample input file where $N = 16$ and $P = 8$. The first two line is the value of $N$ and $P$ respectively. The remains are the initial unsorted data.
Now we show the algorithm on a sample binary tree structure:

```
    10 3 14 7 12 11 5 6 1 2 4 9 15 13 8 16
```

```
Parallel Merge
3 10
7 14
11 12
5 6
Parallel Merge
3 7 10 14
5 6 11 12
Parallel Merge
1 2 4 9
8 13 15 16
Parallel Merge
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

Figure 5: Execution of parallel merge sort algorithm for sample input file where \( N = 16 \) and \( P = 8 \)

5. Tasks

Implement the following procedures whose name and parameters are given below. Write procedure name and parameters as it is given. No additional parameter is allowed. Use semaphores or monitors for the synchronization of the threads. No other structures are allowed.

5.1 void main() procedure

- Read the unsorted sequences of initial data from data.txt file. The first two line of the data.txt file is the size of the data (\( N \)) and number of threads (\( P \)) respectively. You can generate your own data with different combination of numbers (the type of the data will be integer). Define an array variable called buffer whose size is \( N \), to store the initial unsorted data. You can use calloc command in order to avoid stack problems for the huge \( N \) values.
- \( P \) should be the power of 2 and \( N \) should be the multiple of \( P \). You may take \( P = 8 \) or 16 and \( N = 80 \) or 160 for checking your algorithm. But no matter what the value of \( P \) and \( N \) is, your program should run the algorithm correctly for different values of \( P \) and \( N \). In other words, your program should be fully modular, that is your variables should be defined by using \( P \) or \( N \).
- Initialize your semaphore or monitor variables. Create \( P \) threads and call parallel_merge_sort procedure which is given below for each thread.
Print the sorted data, which should be stored in the array variable buffer, at the end of the parallel merge sort operation. The printf operation should be as follows:
printf("%d\n",buffer[i]); where i is the loop variable. No any other printf will be used like printf("The Result = ") or printf("The Merge operation finished") in any step in your program.
Main procedure should not be terminated until all created threads terminate.

5.2 void *parallel_merge_sort(void *thread_id) procedure

- Parameter thread_id is the id of the thread where id can be between 0 and P-1.
- Each thread calls this procedure after they are created.
- Each thread gets their partial data from the initial unsorted data. Use the following index values to get the partial data. Partial data of a thread is buffer[ (thread_id*N/P) + index] where index = 0,1, ..., N/P and thread_id is the id of that thread.
- Copy and use the heap sort algorithm which is given in homework file for the sequential sort of the partial data.
- Call parallel_merge procedure at each iteration and get the results. Remember that iteration number of threads can be different according to their position in the binary tree and the size of the partial data are changed at each iteration.

5.3 void parallel_merge(int myid,int dim,int *state,int size) procedure

- Parameter myid is the id of calling thread, dim is the current iteration number of calling thread, state is the partial data of calling thread and size is the size of partial data.
- Each thread sends and receives their partial data according to the binary tree communication structure.
- Shared memory access communication is used. To do this, you can use array variable buffer which is used to store the initial unsorted data sequence and which is used to print the sorted data sequence at the end of parallel merge sort algorithm.
- Call the compare_exchange_high or compare_exchange_low procedures according to the thread’s positions in the binary tree. The threads which are in the left position call the compare_exchange_low procedure and which are in the right position call the compare_exchange_high procedure.

5.4 void compare_exchange_low(int *state, int *message,int size) procedure

- Parameter state is the partial data of calling thread, message is the partial data of other thread in the parallel merge operation and size is the size of partial data.
- Compare subsequences of two threads and outputs the lesser value until M elements remain.

5.5 void compare_exchange_high(int *state, int *message,int size) procedure

- Parameter state is the partial data of calling thread, message is the partial data of other thread in the parallel merge operation and size is the size of partial data.
- Compare subsequences of two threads and outputs the higher value until M elements remain.
6. Specifications

- Please read the specifications carefully!
- Your homework must be written in C. No C++ codes are accepted.
- You are allowed to use Posix Threads (pthread) library for threads, semaphores and monitors. And you can also use semaphore library. No other libraries are allowed.
- Some part of your solutions will be evaluated using black box technique. So be careful that your printf statement is true!! You will also get partial points from your implementations.
- Try to define your global and common variables in the heap region of the memory. You can use calloc function. For example: int *data = (int *)calloc(S,sizeof *data); where S is the variable which stores the number of elements of variable data.
- Everything you submit should be your own work.
- Please follow the course page on newsgroup (cow) for any update and clarification.
- Please ask your questions related to the homework on cow instead of email in order to your friends, who may face with same problem, can see your questions and answers.
- Your programs will be compiled with gcc and run on the department inek machines. No other platforms/gcc versions etc. will be accepted so check that your code works on ineks before submitting it.
- Sharing and copying any piece of code from each other and INTERNET is strictly forbidden. Both the sharing one and the copying one will be counted as cheated.
- After printf, use fflush(stdout); command so that you can be sure output is not buffered.
- Use sufficient comment lines in your algorithm in order to explain your solution clearly.
- Below some situations that your homework will not be graded:
  - Implementations without the procedures that are given before or with different procedures or different parameters.
  - Using any sleep procedure at any time.
  - Not printing the sorted data at the end of the execution. And not using the printf command as it is wanted before.
  - Reading the input data file with different names. The name of the input file should be data.txt.
  - Just send hw2_semaphore.c, hw2_monitor.c and Makefile. Different file names are not allowed so be careful!! And no any other files are allowed.
  - The Makefile should create two executable called hw2_semaphore and hw2_monitor. Different executable names are not allowed so be careful!!
  - Violation of any non-allowed situations which are given in the text file before.

7. Submission

You will do to two different implementations. The first one is semaphore implementation which should be named as hw2_semaphore.c and the second one is monitor implementation which should be named as hw2_monitor.c.

You will submit a single tar file (hw2.tar.gz) including a Makefile and your source files. The Makefile should create two executable called hw2_semaphore and hw2_monitor. The content of the Makefile should be as follow:
hw2: hw2_semaphore.c
   gcc -o hw2_semaphore hw2_semaphore.c -lpthread -g
   gcc -o hw2_monitor hw2_monitor.c -lpthread -g

The tar file should not contain any directories!! The following command sequence is expected to run your program:

$ tar -xvf hw2.tar.gz
$ make
$ ./hw2_semaphore
$ ./hw2_monitor

May it be easy.