Programming Assignment II

CS3331

Spring 2025

1 Merge Sort

At this point in your computer science education, you should be familiar with merge sort. We will implement a version of merge sort using multiple processes and shared memory. A typical implementation of merge sort using recursion is given in the code snippet below.

```
void MergeSort(int a[],
               int lower_bound,
               int upper_bound) {
  int middle;
  if (upper_bound - lower_bound == 1) {
    if (a[lower_bound] > a[upper_bound]) {
      swap a[lower_bound] and a[upper_bound];
      return;
    }
  }
  /* now we have more than 2 entries */
  middle = (lower_bound + upper_bound)/2;
  /* recursively sort the left section  */
  MergeSort(a, lower_bound, middle);
  /* recursively sort the right section */
  MergeSort(a, middle+1, upper_bound);

  /* merge the left and right sections  */
  Merge(a, lower_bound, middle, upper_bound);
  return;
}
```

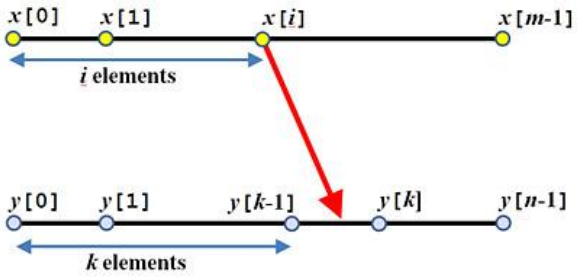
The two calls to `Mergesort()` recursively split the give array section into two sorted sections. Let us call this the split phase. Then, we enter the merge phase in which the two adjacent sorted array sections are merged into a larger sorted one.

Because the two `Mergesort()` calls are independent of each other, they can

run concurrently. This means we could create two child processes to run these two Mergesort() calls. How can we merge these two arrays concurrently? It is not a difficult task, although it is not entirely trivial either.

1.1 Concurrent Binary Merge with Unix Shared Memory

Suppose two sorted arrays x[] and y[], with m and n elements respectively, are to be merged into a sorted output array. Assume also that elements in x[] and y[] are all distinct. Consider an element x[i]. We know that it is greater than i elements in array x[]. If we are able to determine how many elements in array y[] are smaller than x[i], we know the final location of x[i] in the sorted array. This is illustrated in the diagram below:



With a slightly modified binary search, we can easily determine the location of x[i] in the output array. There are three possibilities:

1. x[i] is less than y[0]: In this case, x[i] is larger than i elements in array x[] and smaller than all elements of y[]. Therefore, x[i] should be in position i of the output array.

2. x[i] is larger than y[n−1]: In this case, x[i] is larger than i elements in x[] and n elements in y[]. Therefore, a[i] should be in position $i+n$ of the output array.

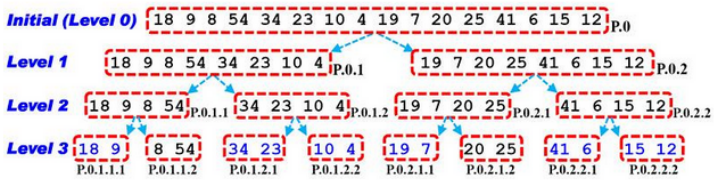
3. x[i] is between y[k−1] and y[k]: A slightly modified binary search will find a k such that x[i] is between y[k−1] and y[k]. In this case, x[i] is larger than i elements in x[] and k elements in y[]. Therefore, x[i] should be in position $i+k$ of the output array.

A program may create $m+n$ child processes, each of which handles an element in x[] or in y[]. Each of these processes determines the location of its assigned element in the sorted array using $O(\log_2 m)$ or $O(\log_2 n)$ comparisons, and writes its value into the corresponding location. After all processes complete, the output array is sorted and is a combination of the two sorted input arrays.

Note that a modified binary search can easily reach this bound, and that you will receive no points for this part if you do not modify the binary search to fit the stated purpose.

1.2 Split

For simplicity, suppose we are given $16 = 2^4$ distinct integers: 18, 9, 8, 54, 34, 23, 10, 4, 19, 7, 20, 25, 41, 6, 15 and 12. This is the initial level (i.e., Level 0). To reach the next level (Level 1), the given array is split into two sections 18, 9, 8, 54, 34, 23, 10, 4 (left) and 19, 7, 20, 25, 41, 6, 15, 12 (right), each of which has $8 = 2^3$ entries, half of the previous level. Then, these sections are handled concurrently by two processes. Of these two processes, the first receives the left section 18, 9, 8, 54, 34, 23, 10 and 4, while the second process receives the right section 19, 7, 20, 25, 41, 6, 15 and 12. See the diagram below.



For the left process, it receives 18, 9, 8, 54, 34, 23, 10 and 4, and concurrently splits its input into two sections: 18, 9, 8 and 54 (left) and 34, 23, 10 and 4 (right). The right process receives 19, 7, 20, 25, 41, 6, 15 and 12 and concurrently splits its input into two sections: 19, 7, 20 and 25 (left) and 41, 6, 15 and 12 (right). This procedure continues until each process receives only two entries. Then, each process does a compare followed by a swap (if needed) to complete sorting the 2-entry array section, and returns.

In the diagram above, each dashed-line frame indicates a process. Each process creates two child processes until that process only has two entries. The number attached to each process shows the ancestor history of that process, where 1 and 2 represent left and right child processes, respectively. For example, $P.0.1.2.1$ is the left child of process $P.0.1.2$, which, in turn, is the right child process of process $P.0.1$.

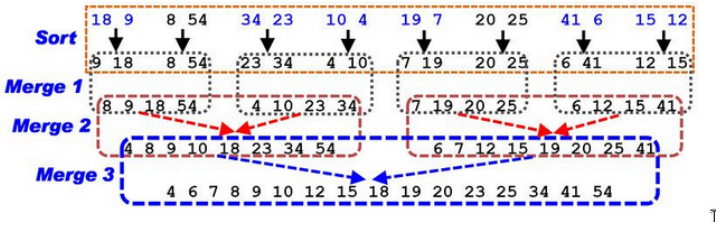
At the bottom most level, we always have $n/2$ processes, where $n = 2^k$ for some $k > 0$. How many processes are there? It is not difficult to see. Recall that we assumed $n = 2^k$ for some $k > 0$. The last level has $n/2 = 2^{k-1}$ processes. Because each level has half of the number of processes of the level below, the total number of processes is the sum of $2^{k-1}, 2^{k-2}, \ldots, 2^1$. This is a geometric progression, you should be familiar with computing the total from your discrete math course. The following is the needed calculation.

$$2^{k-1} + 2^{k-2} + \cdots + 2^0 = \frac{2^{(k-1)+1} - 1}{2 - 1} = 2^k - 1 = n - 1$$

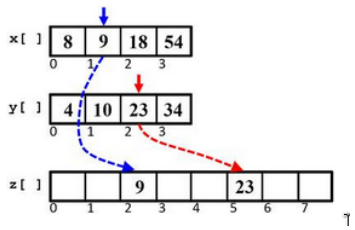
Therefore, the total number of processes needed is $n - 1$, or $O(n)$! This is a good news!

1.3 Merge

Now we turn our attention to the merge phase. As mentioned earlier, the last level processes do not have to do a merge because comparing and swapping two entries is the base case. The diagram below continues the split diagram shown earlier. Each process at the last level sorts the 2-entry array section. Then, we have eight sorted array sections 9, 18; 8, 54; 23, 34; 4, 10; 7, 19; 20, 25; 6, 41; 12,15. Note that these eight sections belong to four processes. The first process has sections 9, 18 and 8, 54; the second has 23, 34 and 4, 10; the third has 7, 19 and 20, 25; and the fourth has 6, 41 and 12, 15. The first process merges sections 9, 18 and 8, 54 into 8, 9, 18, 54; the second process merges 23, 34 and 4, 10 into 4, 10, 23, 34; the third process merges 7, 19 and 20, 25 into 7, 19, 20, 25; and the fourth process merges 6, 41 and 12, 15 into 6, 12, 15, 41.



Now, the resulting four sections belong to two processes. Let us carefully work out what the first process will do with a modified binary search. This process has two sections 8, 9, 18, 54 and 4, 10, 23, 34. Let array x [ ] contain the first section 8, 9, 18, 54, and let array y [ ] contain the second section 4, 10, 23, 34. We also need a temporary array to hold the merged result. Why? Let this array be z [ ] .



As discussed in the binary merge section, each entry of each array is assigned a process, which performs a modified binary search to locate the correct position of the assigned entry in the result array.

Consider the process assigned to array entry x [1] . A binary search shows that the value of x [1] (i.e., 9) is between y [0] = 4 and y [1] = 10. This means x [1] is larger than one entry in array x [ ] and also larger than one entry in array y [ ] . Therefore, x [1] should be in location z [2] in the result array.

Now consider the process assigned to y [2] . In this case, y [2] is larger than two entries in array y [ ] . A binary search using y [2] to search array x [ ] tells

us that y [2] 's value (i.e., 23) is between x [2] = 18 and x [3] = 54. Therefore, y [2] is larger than 3 entries of array x [ ] . Thus, in the result array, y [2] is larger than five entries of the combined arrays x [ ] and y [ ] , and, consequently, it should be in z [5] .

How many processes are needed to complete this binary merge for each merge? It is obvious that the number of processes needed is the total number of entries of both arrays because each element requires a process. Because in each merge level there are always n elements involved, we know that n processes will be needed for each merge level. Because there will be $k - 1 = \log_2 n - 1$ merge levels, the total number of processes needed appear to be $n(\log_2(n) - 1)$ or $O(n \log_2 n)$. Well, a careful planning can allow us to only use n processes to do all the binary merge work. Combined with the MERGE phase, the total number of processes needed, assuming we know how to do the binary merge part using n processes, is $2n$ or $O(n)$ processes.

Next we shall examine the number of comparisons performed. First of all, we notice that the MERGE phase does not need any comparisons, and all comparisons are performed in the binary merge component. The first level in the merge process involves comparing two entries, and, hence, a process only executes one comparison. The second level merges two array sections each of which has two entries. Thus, the assigned process to any array entry requires $\log_2 2^1 = 1$ comparison to get the job done. The third level merges two array sections each of which has $4 = 22$ entries, and $\log_2 2^2 = 2$ comparisons are required. Similarly, the fourth level merges two array sections each of which has $8 = 23$ entries, and $\log_2 2^3 = 3$ comparisons are needed. The last level merges two array sections each of which has $n/2 = 2^k/2 = 2^{k-1}$ entries, and $\log_2 2^{k-1} = k-1$ comparisons are needed. In this way, the total comparisons a process must perform is the sum of the number of comparisons that process performs at each level. More precisely, the total number of comparison for a process to perform is the sum of 1 (first level), 1 (level 2), 2 (level 3), 3 (level 4), $\ldots$, $k - 1$ (last level). This is an arithmetic progression and can be computed as follows:

$$\begin{aligned} 1 + 1 + 2 + \cdots + (k-1) &= 1 + (1 + 2 + \cdots + (k-1)) \\ &= 1 + \frac{(1 + (k-1))(k-1)}{2} \\ &= 1 + \frac{k(k-1)}{2} \\ &= O(k^2) = O(\log_2^2 n) \end{aligned}$$

Therefore, the total number of comparisons a process needs in the MERGE phase is $O(\log_2^2 n)$. Because we have n processes each of which requires $O(log_2^2 n)$ comparisons to complete the merge sort task, the total work of all processes is $O(n \log_2^2 n)$. This is higher than the sequential version, which only requires $O(n \log_2 n)$ comparisons; however, because each process does $O(\log_2^2 n)$ comparisons concurrently, the multiple process version in theory is certainly much

faster than the sequential counterpart.

2 Requirements

Write two programs `main.c` and `merge.c`. Program `main.c` does not require any command line arguments, and `merge.c` takes at least two command line arguments Left and Right, plus any other command line arguments based on your program implementation. The job of program `main.c` consists of the following:

1. Program `main.c` reads an array a [ ] into a shared memory segment. Let the number of elements of a [ ] be $n = 2^k$ for some $k > 1$.
2. `main.c` creates a child process to run program `merge.c` using the execvp() system call and passes the assigned Left, Right and other needed information to program `merge.c`. Initially, Left and Right are 0 and $n - 1$, respectively.
3. Then, `main.c` waits for its child process to complete, prints out the results, and terminates itself.

The job for program `merge.c` to perform is the following:

1. When `merge.c` runs, it receives the left and right indices, Left and Right, and any other information you find necessary from its command line arguments.
2. Then, it splits the array section a [Left .. Right] into two at the middle element a [M] . After the split is obtained, two child processes are created, each of which runs `merge.c` using execvp ( ). The first child receives Left and M–1, while the second receives M and Right. In this way each child process performs a merge sort on the given array section. The parent then waits until both child processes complete their job.
3. After this, program `merge.c` uses the modified binary merge method to merge the two sorted sections as shown below.
   (a) `merge.c` creates a process for each entry in the two array sections, and waits for the completion of all of these child processes.
   (b) Each child process uses the assigned array entry to search the other array in order to find its final position in the merged array. Note that these child processes should not store the assigned entry back to the given array. In other words, the result array must be elsewhere. Why? Use your creativity to find a place to store this result array.
4. After all child processes complete, the merged (and sorted) array must be copied back to shared memory, overwriting the original. At this point don't forget to remove those temporary arrays. Then, `merge.c` exits.

You may use multiple shared memory segments. You must use the `execvp()` system call. You must use the modified binary search to determine the position of x[i] in y[], or y[j] in x[]. Programs that do not do this will receive 0 points. The process structure must be as specified above.

2.1 Input

The input to the `main.c` program is a file with the following format:

```
n
a[0] a[1] a[2] ... a[n-1]
```

where n is the length of the input array a and $a[i]$ is the i-th element of said array.

2.2 Output

Below is an example program output. The `.....` in-between lines are to indicate that it is possible print statements from other processes may occur here. **Please do not print the .....!**

```
Merge Sort with Multiple Processes:

*** MAIN: shared memory key = 1627930027
*** MAIN: shared memory created
*** MAIN: shared memory attached and is ready to use

Input array for mergesort has 8 elements:
   7   4   9   2   3   8   6   5

*** MAIN: about to spawn the merge sort process
   ### M-PROC(4913): entering with a[0..7]
      7   4   9   2   3   8   6   5
       .........
   ### M-PROC(3717) created by M-PROC(4913): entering with a[4..7]
      3   8   6   5
       .........
   ### M-PROC(2179) created by M-PROC(4913): entering with a[0..3]
      7   4   9   2
   ### M-PROC(4756) created by M-PROC(3717): entering with a[4..5]
      3   8
       .........
   ### M-PROC(3421) created by M-PROC(3717): entering with a[6..7]
      6   5
       .........
   ### M-PROC(3421) created by M-PROC(3717): entering with a[6..7] -- sorted
      5   6
```

```
    ..........
### M-PROC(4756) created by M-PROC(3717): entering with a[4..5] -- sorted
   3   8
    ..........
### M-PROC(3717) created by M-PROC(4913): both array section sorted. start merging
    ..........
   $$$ B-PROC(6421): created by M-PROC(3717) for a[4] = 3 is created
    ..........
   $$$ B-PROC(6421): a[4] = 3 is smaller than a[6] = 5 and is written to temp[0]
    ..........
   $$$ B-PROC(6423): created by M-PROC(3717) for a[7] = 6 is created
    ..........
   $$$ B-PROC(6423): a[7] = 6 is between a[4] = 3 and a[5] = 8 and is written to temp[2]
    ..........
   $$$ B-PROC(6421): created by M-PROC(3717) for a[4] = 3 is terminated
    ..........
   $$$ B-PROC(6428): created by M-PROC(3717) for a[5] = 8 is created
    ..........
   $$$ B-PROC(6428): for a[5] = 8 is larger than a[7] = 6 and is written to temp[3]
    ..........
### M-PROC(3717) created by M-PROC(4913): merge sort a[4..7] completed:
   3   5   6   8
    ..........
### M-PROC(2719) created by M-PROC(4913): merge sort a[0..3] completed:
    ..........
   2   4   7   9
    ..........
### M-PROC(4913): entering with a[0..7] completed:
   2   3   4   5   6   7   8   9
    ..........
*** MAIN: merged array:
    2   3   4   5   6   7   8   9

*** MAIN: shared memory successfully detached
*** MAIN: shared memory successfully removed
*** MAIN: exits
```

The above output of `merge.c` are shown in some specific order; however, this usually is not the case in reality. The output lines can mixed, and the order of output lines from different processes may also be very different.

Some subtleties to note:

- The number `nnnn` in `PROC(nnnn)` is the process ID of the process which generates the message. `M` and `B` are prefixes for the merge sort and binary merge processes, respevctively.
- If an output line from a process includes data values, there should not be

any output between the message and its corresponding data values. For example, the following two output lines from process 3717 must be printed next to each other.

```
### M-PROC(3717) created by M-PROC(4913): entering with a[4..7]
   3   8   6   5
```

- If you use more than one shared memory segments, you should repeat the output of shared memory segment and indicate the use of each. For example, if you allocate two shared memory segments, one for quicksort and the other for binary merge, your output should look like the following, where XXXXX is a meaningful and short description of the purpose of the created shared memory.

```
*** MERGE: shared memory key = 1627930027
*** MERGE: shared memory created
*** MERGE: shared memory attached and is ready to use for XXXXX purpose.

<<<<<<<<<< Other Output >>>>>>>>>>

*** MERGE: XXXXX shared memory successfully detached
*** MERGE: XXXXX shared memory successfully removed
```

- The output lines from `main.c` always starts with `*** MAIN:` from column 1 (i.e., no leading space). Messages from `merge.c` have an indentation of three spaces and started with `### M-PROC(abcd):`, where `abcd` is the PID of this particular merge sort process. Data values printed by a merge sort process has the same indentation as that of `### M-PROC(abcd):` and each integer must printed using four positions, right aligned. Refer to the sample output format.

- Each process created for binary merge has an indentation of six spaces and must start with `$$$ B-PROC(abcd):`, where `abcd` is the PID of this particular binary merge process. Refer to the out from binary merge processes shown above for the three possible cases when doing a modified binary search.

- The temporary array temp [ ] can be created by a merge process or elsewhere, which is your decision. If it is a shared memory, you must be prudent so that these temporary arrays will be removed properly. As a rule of thumb, minimize the use of shared memory segments because your peers are also using it. If you use too many, you may actually cause problems for your peers. Moreover, one unremoved shared memory will cost you 10 points.

  A temporary array temp [ ] always start with 0 (i.e., temp [0] being the first entry of temp [ ]) rather than using the same index/subscript as that of the input array. This may help you reduce the chance of making a mistake.

3 Submission Guidelines

The following sections elaborate submission requirements not directly related to the semantics of the program itself.

3.1 General Rules

1. The program must be written in C, programs written in other languages will receive a 0.
2. Your work must be submitted as a `.zip` file via canvas. This `.zip` will contain your source code, your `Makefile`, and your `README`.
3. Your `Makefile` should produce an executable file named `prog2`.
4. We will use the following approach to compile and test your program:

   ```
   make
   ./prog2 < input-file
   ```

   This procedure may be repeated a number of times with different input configurations to see if your program behaves correctly. If your program does not compile or run in this way, you risk receiving a 0 on this program. As always, you should test your program's output when performing `stdout` redirection as well.
5. Your implementation should fulfill the program specification as stated.

3.2 Cleaning Shared Memory

Since shared memory segments are system-wide entities and will stay in the system after you logout, you are responsible to remove **all** shared memory segments whether your program ended normally or abnormally. **Each unremoved shared memory segment will cost you 10 points.** Thus, if you have a correct program and if we find out your program has two shared memory segments left behind, you receive no more than 80 points (i.e., a 20 point deduction).

3.3 Program Style and Documentation

- At the top of each `.c` file, the first comment should be a program header to identify yourself like so:

  ```
  // ------------------------------------------------------------
  // NAME : John Smith                          User ID: xxxxxxxx
  // DUE DATE : mm/dd/yyyy
  // PROGRAM ASSIGNMENT #n
  // FILE NAME : xxxx.yyy
  // PROGRAM PURPOSE :
  // A couple of lines describing your program briefly
  // ------------------------------------------------------------
  ```

Here, User ID is the one you use to log in to MTU systems.

- Your programs must contain a reasonable amount of concise and to-the-point comments. Do not write a novel.
- Your program should have good indentation.

3.4 The `README` File

A `README` file should be included with your submission. This file must either be a plaintext file named `README`, or a PDF file named `README.pdf`. The `README` should answer the questions enumerated below.

1. Explain the allocation and use of each shared memory segment.
2. Are there potential race conditions (i.e., processes use and update a shared data item concurrently) in your program?
3. Why you should not save the assigned array entry back to the given array in the binary merge phase? State explicitly your reason.
4. Explain how you allocate a temporary array to hold the intermediate result in each execution of Mergesort().
5. We assigned a process to each array entry every time when Mergesort() is executed to perform binary merge. Then, Mergesort() waits for all of these processes before terminates itself. As a result, we repeatedly create and terminate n processes $\log_2 n$ times, which is a waste of time and system resource. Suppose you are allowed to use busy waiting. Can you only create n processes at the beginning of the MERGE phase so that they can be used in each binary merge? State your answer as clearly as possible.

Please ensure you clearly indicate which question is being answered at a given point within your `README`. Make sure you have a convincing argument for each question!

3.5 Submission File Structure

Your submission should follow this file structure:

```
submission/
  main.c  # main program implementation
  merge.c # merge sort implementation
  README{.pdf,.txt,}
  Makefile
```

Please do not write code in files outside the ones listed above, as you may lose points for not following the program file structure. Remember that file names in UNIX-like systems are case-sensitive, so `README.txt` is a different file from `readme.txt`!

4 Final Remarks

It can be incredibly cumbersome working with UNIX shared memory! I recommend you work on creating a correct implementation of merge sort using the binary search merge, and then work on parallelizing the computation from that base.

As always, start early, so that if you run into an issue you can seek help before the program is due!

Finally, run through this quick checklist before and after you submit!

- My submission consists entirely of my own work, and I know I risk failing the entire course if I have plagiarized from any source.
- I am submitting the correct program to the correct assignment on canvas.
- My program follows the program file structure described in 3.5.
- I have included a `Makefile`.
- I have included my `README` file, and it is either a plaintext or PDF file.
- I have tested to make sure my `Makefile` works correctly, and successfully compiles the program using the command given in section 3.1.
- I have tested my program to ensure it works with several inputs, including the given example input.
- My program's output conforms to the output requirements described in section 2.2.
- My program takes input from `stdin`, as described in section 2.1.
- My program is written in C.
- My program uses only ThreadMentor for concurrency and synchronization. I do not use pthreads or any other threading library.
- I have tested both compiling and running my program on a CS lab machine, or `colossus.it.mtu.edu`.
- My program runs the threads of a pass concurrently, I haven't accidentally sequentialized the computation!