Programming Assignment II

Due on Friday, October 19, 2018 @ 11pm

100 points

Merge Sort with Unix Shared Memory

You perhaps have learned merge sort in your data structures or other courses. We will implement a version of merge sort using multiple processes and shared memory. A typical merge sort using recursion looks like the following.

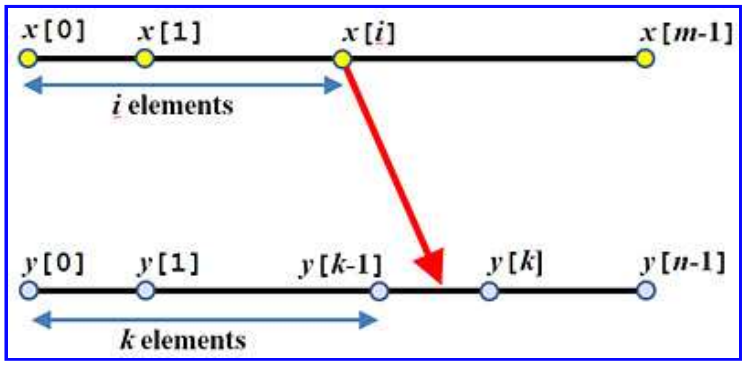
```
function MergeSort(a[ ], lower_bound, upper_bound)
{
     int middle;

     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;
     MergeSort(a, lower_bound, middle);   /* recursively sort the left section  */
     MergeSort(a, middle+1, upper_bound); /* recursively sort the right section */

     Merge(a, lower_bound, middle, upper_bound);
                                          /* merge the left and right sections  */
     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 to do merging concurrently? It is not a difficult task, although it is not entirely trivial either.

Binary Merge with Unix Shared Memory

Merging two sorted arrays can also be done concurrently. 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 larger 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:

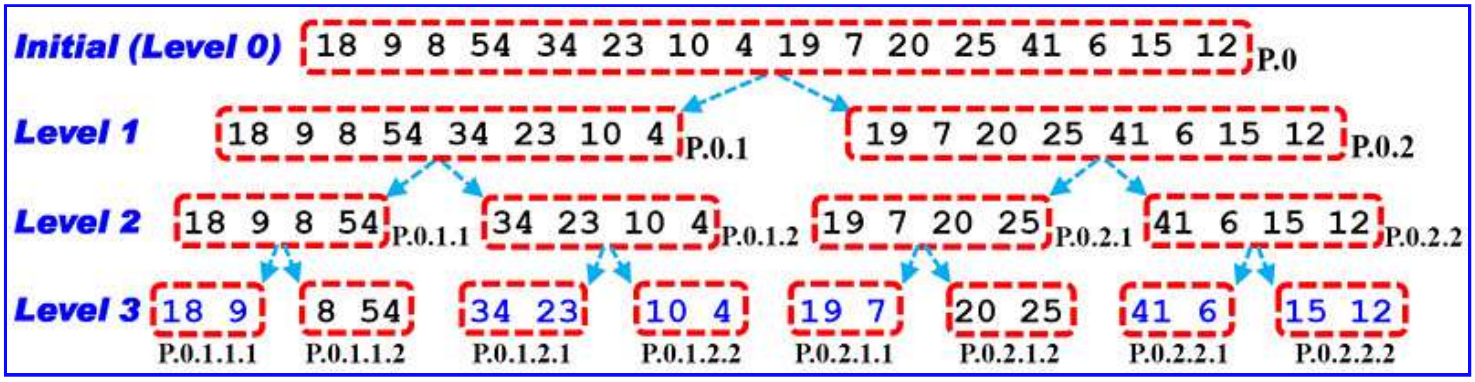
1. ***x*[*i*] is less than *y*[0]**: In this case, $x[i]$ is larger than i elements in array $x[\ ]$ and smaller than all elements of $y[\ ]$. Therefore, $x[i]$ should be in position i of the output array.
2. ***x*[*i*] is larger than *y*[*n*-1]**: In this case, $x[i]$ is larger than i elements in $x[\ ]$ 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 $x[\ ]$. Therefore, $x[i]$ should be in position $i+k$ of the output array.

Hence, 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 a zero for this part if you do not modify the binary search to fit the stated purpose.

Example -- 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.



The out-of-order pairs are shown in blue at the last level

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}, ..., $2^{k-k} = 1$.

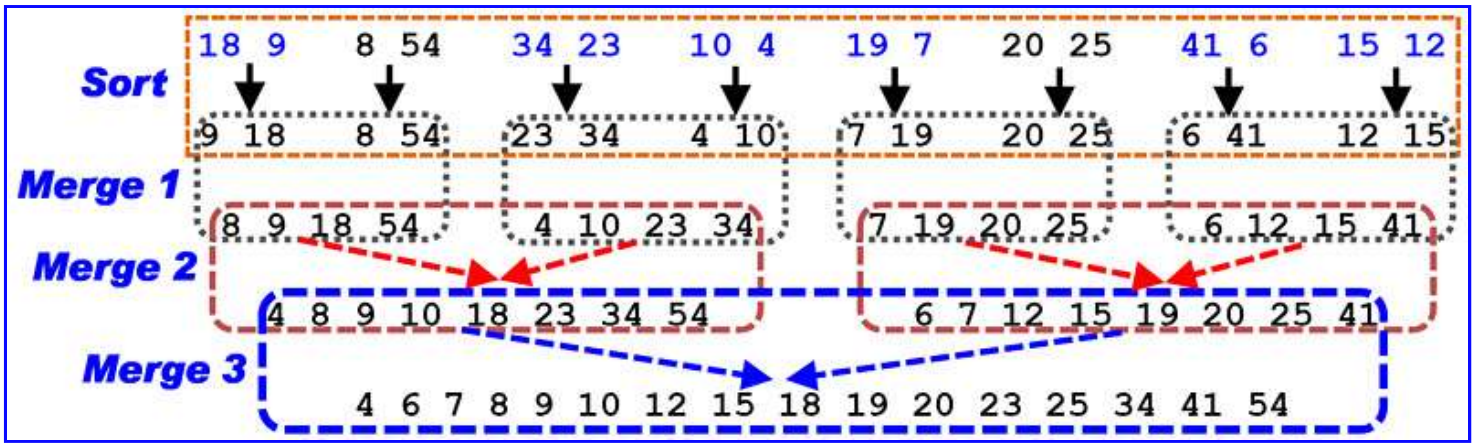
This is a geometric progression and you have learned the way of computing the total somewhere before entering this class. The following is the needed calculation.

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

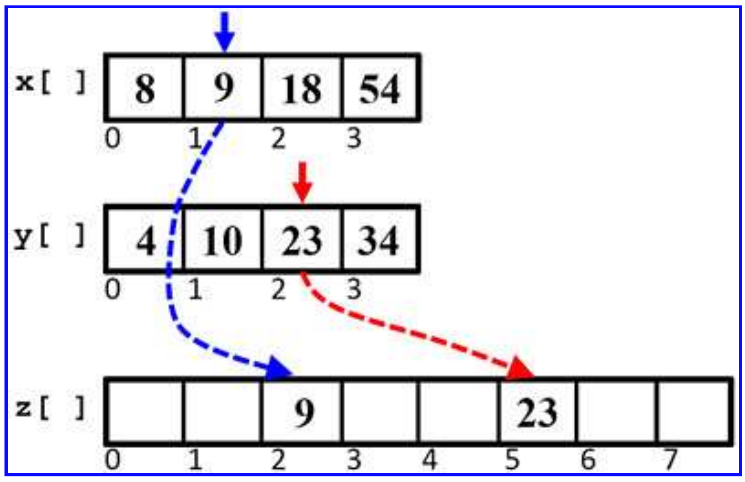
Therefore, the total number of processes needed is n-1, or $O(n)$! This is a good news!

Example -- 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 would be easier. 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 $n \times (\log_2(n) - 1)$ 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 $4 = 2^2$ entries, and $\log_2(2^2) = 2$ comparisons are required. Similarly, the fourth level merges two array sections each of which has $8 = 2^3$ 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), ..., k-1 (last level). This is an arithmetic progression and can be computed as follows:

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

Therefore, the total number of comparisons a process needed 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 $O(n \times \log_2^2(n))$. This is higher than the sequential version, which only requires $O(n \times \log_2(n))$ 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.

Program Specification

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 some other needed command line arguments

based on your program design. The job for program `main.c` to do 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` prints out the input array.
3. `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.
4. Then, `main.c` waits for its child process to complete, prints out the results, and terminates itself.

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

1. When `merge.c` runs, it receives the left and right indices `Left` and `Right` and other information 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`+1 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.

Important Notes

- **You may use multiple shared memory segments, and you must use the `execvp()` system call to solve this problem.**
- **You must use the indicated modified binary search to determine the position of `x[i]` in `y[ ]` and the position of `y[j]` in `x[ ]`. Otherwise, you will receive no point for this programming assignment.**
- **The process structure must be as specified. Otherwise, you will receive no point.**

Input and Output

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

```
n   <---------------------------- the number of elements of array a[ ]
a[0]  a[1]  a[2] ..... a[n-1] <--- elements of array a[ ]
```

You may assume the following:

- The value of `n` is a positive integer in the range of 4 and 32 and is always a power of 2 (*i.e.*, 4, 8, 16 or 32).
- The values for array `a[ ]` are ***distinct*** integers.

The following shows a possible program output.

```
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 <------- elements of a[0] to a[7]
       .........
   ### M-PROC(3717) created by M-PROC(4913): entering with a[4..7]
      3   8   6   5  <---------------------- elements of a[4] to a[7]
       .........
   ### M-PROC(2179) created by M-PROC(4913): entering with a[0..3]
      7   4   9   2  <---------------------- elements of a[0] to a[3]
   ### M-PROC(4756) created by M-PROC(3717): entering with a[4..5]
      3   8 <------------------------------- elements of a[4] to a[5]
       .........
   ### M-PROC(3421) created by M-PROC(3717): entering with a[6..7]
      6   5 <------------------------------ elements of a[6] to a[7]
       .........
   ### M-PROC(3421) created by M-PROC(3717): entering with a[6..7] -- sorted
      5   6 <------------------------------ sorted elements of a[6] to a[7]
       .........
   ### M-PROC(4756) created by M-PROC(3717): entering with a[4..5] -- sorted
      3   8 <------------------------------- sorted elements of a[4] to a[5]
       .........
   ### 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
```

Here are some notes:

NOTE: The number `nnnnn` in `PROC(nnnnn)` is the process ID of the process which generates the message. `M` and `B` are merge sort and binary merge processes, respevctively.

NOTE: 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
```

NOTE: 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.

NOTE: 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
```

NOTE: 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-PROCE(abcd):` and each integer must printed using four positions, right aligned. Refer to the sample output format.

NOTE: 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.

NOTE: 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 very careful 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 un-removed 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 to make mistakes.

Submission Guidelines

General Rules

1. All programs must be written in C.
2. Use the `submit` command to submit your work. You may submit as many times as you want, but only the last on-time one will be graded.
3. Your program should be named as `main.c` and `merge.c`. Since Unix filename is case sensitive, `MAIN.c`, `main.C`, `Main.c`, etc are ***not*** acceptable.
4. We will use the following approach to compile and test your programs:

```
gcc     main.c -o main        <-- compile main.c
gcc     merge.c -o merge      <-- compile merge.c
./main < input-file           <-- test your program
```

   This procedure may be repeated a number of times with different input files to see if your program works correctly. You may use the 16 integers in the **Example** sections for your testing. This is our public data.

   You must use the above command lines to compile and run your program. Otherwise, our grader may not be able to grade your program, and you will risk a low or even a 0

score.

5. Your implementation must fulfill the program specification as stated. Any deviation from the specifications will cause you to receive **zero** point.
6. A `README` file is always required.
7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to syntax errors, wrong file names, etc., **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**. **Therefore, before submitting your work, make sure your program can compile and run properly.**

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, etc. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point. Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
/* ------------------------------------------------------------ */
/* NAME : John Smith                          User ID: xxxxxxxx */
/* DUE DATE : mm/dd/yyyy                                        */
/* PROGRAM ASSIGNMENT #                                         */
/* FILE NAME : xxxx.yyyy.zzzz (your unix file name)             */
/* PROGRAM PURPOSE :                                            */
/*    A couple of lines describing your program briefly         */
/* ------------------------------------------------------------ */
```

   Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
/* ------------------------------------------------------------ */
/* FUNCTION  xxyyzz : (function name)                           */
/*     the purpose of this function                             */
/* PARAMETER USAGE :                                            */
/*    a list of all parameters and their meaning                */
/* FUNCTION CALLED :                                            */
/*    a list of functions that are called by this one           */
/* ------------------------------------------------------------ */
```

   Note that you may also use `/* ... */` for comments if the compiler will complain. The use of `//` is part of the new C standard.
2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

Program Specification

Your program must follow exactly the requirements of this programming assignment. Otherwise, you receive 0 point even though your program runs and produces correct output. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specifications. For example, your program is not divided into functions and files, etc. when the specifications say you should.
3. Any other significant violation of the given program specification.
4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. Our grader will make a decision. Hence, carefully check your program output.

Program Correctness

If your program compiles and runs, we will check its correctness. We run your program with at least two sets of input data, one posted on this programming assignment page (the public one) and the other prepared by the grader (the private one). You program must deliver correct results for both data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

Shared Memory Issues

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 un-removed shared memory segment will cost you 10 points.** Thus, if you have a correct program and if the grader finds out your program has two shared memory segments left behind, you receive no more than 80 points (*i.e.*, 20 points deduction).

The `README` File

A file named `README` is required to answer the following questions:

1. The logic of your program
2. Why does your program work?
3. Explain the allocation and use of each shared memory segment.
4. Are there potential race conditions (*i.e.*, processes use and update a shared data item concurrently) in your program and in the program specifications?
5. Why you should not save the assigned array entry back to the given array in the binary merge phase? State explicitly your reason.
6. Explain how you allocate a temporary array to hold the intermediate result in each execution of `Mergesort()`.
7. 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.

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line.**

Note that the filename has to be `README` rather than `readme` or `Readme`. Note also that there is ***no*** filename extension, which means filename such as `README.TXT` is ***NOT*** acceptable.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your `README` rather than a word processor.

Final Notes

1. Your submission should include three files, namely: `main.c`, `merge.c` and `README`. Please note the way of spelling filenames.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**