COMP3230: Principles of Operating Systems
Programming Assignment 1
Question Paper (2020)

Deadline: BEFORE **11:55pm 15th October, 2020 (Thursday)**

[Early-Bird Bonus] Gain **5%** Bonus Marks if you submit this assignment BEFORE **11:55pm 8th October, 2020 (Thursday)**

Weighting: 10% of the course assessment

Full Mark: 100

Objectives

In this assignment, you are required to write multi-process programs in C. Upon the completion of this assignment, you should get hands-on experiences on:

1) Creating processes using ***fork()*** and parent-child process synchronization using ***wait()***;
2) Inter-Process Communication between processes using ***shared memory*** and ***signals***;
3) Implementing a multi-process concurrent 4-way merge-sort algorithm;

Submission

1. You need to prepare the following files::
   - ✧ assign1_q1_1.c (based on assign1_q1_template.c)
   - ✧ assign1_q1_2.c (based on assign1_q1_template.c)
   - ✧ assign1_q2_1.c (based on assign1_q2_main_template.c)
   - ✧ assign1_q2_2.c (based on assign1_q2_main_template.c)
   - ✧ assign1_q2_funcs.h
   - ✧ assign1_q2_funcs.c
   - ✧ assign1_q2_3.pdf (For Q2.3 Performance Evaluation, you should answer this question in a document and upload it as **a pdf file)**
2. Zip the above **7 files** into one archive file, which should be named as:

   COMP3230A-YourUniversityID-yourNAME.zip
3. Submit the zip file to HKU Moodle.

Reminders

1. In order to complete this assignment, a **review of Lecture Note 2 “Process”** and **Tutorials 1 and 2 PowerPoint Slides** is highly recommended.
2. Before submitting your answers to Moodle, please make sure that **all your C source code files can be compiled and executed successfully with expected output on our Linux server (workbench)**. Otherwise, no marks will be given.
3. The template files can be downloaded from Moodle: (1) assign_q1_template.c (2) assign1_q2_main_template.c (3) assign1_q2_funcs.h (4) assign1_q2_funcs.c

Question 1: Shared Memory, Signals and wait() [35%]

Q1.1 (20%) Write a program based on the given template file (assign_q1_template.c) as shown in **Fig. 1.2**, which performs the following tasks:

1. Your main program should receive three positive integers (denoted as v1, v2, v3) from the command line. Check whether the number of command line arguments is 3. If not, print the error information and return. (The part is provided in the template file. You are NOT allowed to change it)
2. Create a shared memory segment in parent process (*shmget()*) and attach it to the integer array *differ* **with size 3** (*shmat()*).
3. Use fork() to create a child process (denoted as process A). **After fork()**, the parent process calculates and prints the differences of these three integers (i.e., differ[0] = v2-v1, differ[1]= v3-v2); **then** uses ***signal()*** to "tell" the child process (Process A) to begin to calculate "differ[2] = differ[0]+ differ[1]" and print the sum of differences "differ[2]". When the child process A finishes the calculation, it will exit (*exit()*).The parent process needs to wait for the child process to exit before it terminates.

Note: You only need to use SIGSTOP and SIGCONT in Q1.1. You will get zero mark if *sleep()* is used.

1.1 In the **parent process**, you need to print the following information:

1) The number of bytes of shared memory it successfully requested;
2) Parent's PID and the differences of the three integers;
3) The child's PID which will receive the signal and the type of the signal sent by the parent;
4) The exited child's PID (returned from *wait()*);

1.2 In the **child process**, you need to print the following information:

1) The child's PID and the signal received from its parent;
2) The sum of differences (differ[2]) calculated in the child process;

```
jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/tutorial2
jpwang@workbench:~/git_tutorial/tutorial2$ gcc assign1_q1_1.c -o main
jpwang@workbench:~/git_tutorial/tutorial2$ ./main 5 3 8
This is the BEGINNING of the program.

Apply 12 bytes.
Parent process ID: 72057.
Differences: -2, 5.
Send a SIGCONT to process 72058.

Receive a SIGCONT.
Child process ID: 72058.
Sum of differences: 3.

Exited Process ID: 72058.
This is the END of the program.
jpwang@workbench:~/git_tutorial/tutorial2$
```

Fig 1.1 Sample output of Q1.1

```c
int main(int argc, char* argv[])
{
    printf("This is the BEGINNING of the program.\n\n");
    if(argc-1 != 3){
        printf("Error: The number of input integers now is %d. Please input 3 integers.\n",argc-1);
        return -1;
    }// Don't modify this Error Checking part

    {
        // Write your code in this brace



    }
    printf("\nThis is the END of the program.\n");
    return 0;
}
```

```c
sleep(0.001);  // which you may need for registration of signal handlers in child process,
               //only allowd at the beginning of the parent process after fork();
printf ("Apply %d bytes.\n",***);
printf("Child process ID: %d.\n", getpid());
printf("Sum of differences: %d.\n\n", ***);
printf("Send a SIGCONT to Process %d.\n\n", ***);
```

Fig.1.2 Template file (assign_q1_template.c)

Q1.2 (15%): Based on your code of **Q1.1**, create another child process, denoted as **Process B**, using fork(). The relationship of these three processes is shown in **Fig. 1.3**. The parent process needs to finish the calculation of differences of arguments firstly (differ[0] = v2-v1, differ[1]= v3-v2), and <u>then</u> send a *signal* to its child **Process A firstly** (Refer to **Q1.1**). Child process A needs to finish the calculation of the sum of differences ("differ[2] = differ[0]+ differ[1]") and then send a signal to "tell" **child process B** to start comparing the sum (differ[2]) with 0 and print the corresponding information accordingly (Read output details specified in 2.3). When the child process A and B finish their job, they should exit (exit()) respectively. The parent process needs to wait for the two child processes and also count the number of exited child processes before it exits.

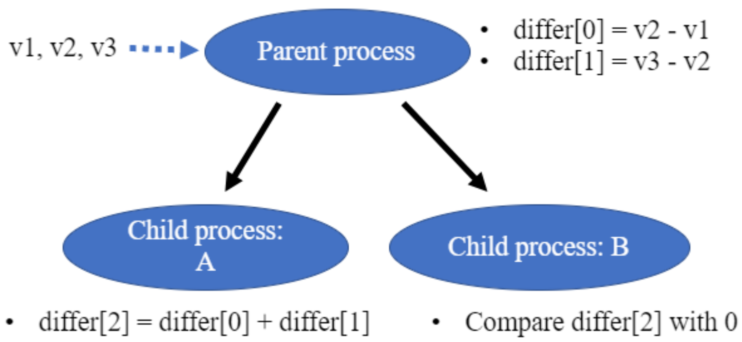


Fig. 1.3 Process tree

```
jpwang@workbench.cs.hku.hk:22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/tutorial2
jpwang@workbench:~/git_tutorial/tutorial2$ gcc assign1_q1_2.c -o main
jpwang@workbench:~/git_tutorial/tutorial2$ ./main 5 3 8
This is the BEGINNING of the program.

Apply 12 bytes.
Parent process: 33855.
Differences: -2, 5.
Send a SIGCONT to process 33857.

Receive a SIGCONT.
Child Process A ID: 33857.
Sum of differences: 3.
Send a SIGCONT to process 33856.

Receive a SIGCONT.
Child Process B ID: 33856.
The 3rd argument is larger than the 1st argument.

Exited Process ID: 33857; Count: 1.
Exited Process ID: 33856; Count: 2.
This is the END of the program.
jpwang@workbench:~/git_tutorial/tutorial2$
```

Fig 1.4 Sample output of Q1.2

2.1 In the **parent process**, you need to print the following information:

1) The number of bytes of shared memory it requested;
2) Parent's PID and the differences of the three integers;
3) The child's PID which will receive the signal and the type of the signal it sends;
4) The exited process' PID and the number of exited child processes;

2.2 In the **child process A**, you need to print the following information:

1) The PID of the child process A and the signal it received.
2) The sum (differ[2]) of differences
3) The PID of process B who will receive the signal and the type of signal sent;

2.3 In the **child process B**, you need to print the following information:

1) The PID of process B and the signal it received.
2) The comparison result: Compare the sum (differ[2]) with 0 to check if the 3rd argument (v2) is smaller or larger than or equal to the 1st argument (v1).

Note: "This is the END of the program." is only printed once.

Question 2: Parallel 4-way Merge-sort using Fork() and Shared Memory [65%]

In this question, you are going to write a **parallel 4-way merge-sort program** using fork() to sort **(4^n)*max_num** integers **into an ascending order**, where **n** **(n>=1)** and **max_num (>=4)** are set by users. Here **max_num** is the number of integers to be sorted by a single process using a local 4-way recursive merge-sort (done in Tutorial 1) before we start merging the sorted sub-arrays, and **n** is used to control the number of processes that participate the 4-way merge sort. For example, if **n=2**, the total number of processes that participate the 4-way parallel merge-sort will be $4^2 = 16$.

Q2.1 (10%). Sort (4^1)*max_num integers using 4 processes (i.e., **n=1**, 1 parent process and 3 child processes)**:** Write a program to implement a parallel 4-way merge-sort algorithm using 4 processes. The relationship of 4 process is shown in **Fig. 2.1**. You should create a shared memory segment to store the input integer array of size **4*max_num* (sizeof(int)),** divide the jobs equally between the 4 processes.

After the creation of 3 child processes, let each of the 3 child processes and the parent process sort **max_num** number of integers using a **local 4-way merge-sort**. At the end, let the parent process merge the sorted results.

Besides, you need to print the sorted or merged results by each process as shown in **Fig. 2.2**.

Note 1: the 4-way merge sort has been implemented in Tutorial 1 exercise. You should modify that code to make it work as a function call in your parallel 4-way merge-sort. Please read "Q2 Requirements" for more details.

Note 2: The sorted results should be verified based on a sequential bubble sort program by parent program (included in our template file assign1_q2_funcs.c) after all child processes exit to verify the correctness of the sorted results and print "**The sorted result is correct".**

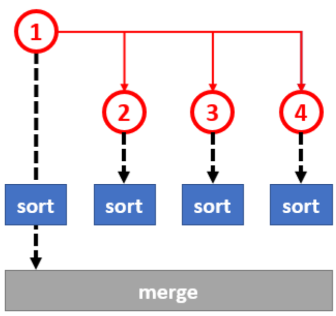


Fig. 2.1 Concurrent Merge Sort with Shared Memory (n=1). Process 1 created child process 2, 3, and 4. Each of these processes (including Process 1) will sort max_num number of integers concurrently then let Process 1 (parent) merge the results.

```
jpwang@workbench.cs.hku.hk22 - Bitvise xterm - jpwang@workbench: ~/git_tutorial/assign_q2_solutions
jpwang@workbench:~/git_tutorial/assign_q2_solutions$ gcc assign1_q2_funcs.c main.c  -o main -lm
jpwang@workbench:~/git_tutorial/assign_q2_solutions$ ./main 1 4
This is the BEGINNING of the program.
n: 1; max_num: 4.
Sort (((4^n)*max_num) = 16 integers.
Input array: 4544 28214 11246 8870 16887 2234 20162 27593 20255 30710 15445 15276 7136 14 1395 8096

Start timing...
Process ID: 86563; Sorted 4 integers: 14 1395 7136 8096
Process ID: 86564; Sorted 4 integers: 4544 8870 11246 28214
Process ID: 86565; Sorted 4 integers: 2234 16887 20162 27593
Process ID: 86566; Sorted 4 integers: 15276 15445 20255 30710
Process ID: 86563; Merged 16 integers: 14 1395 2234 4544 7136 8096 8870 11246 15276 15445 16887 20162 20255 27593 28214 30710
End timing.
The elapsed time (us) is 433.

The sort result by merge sort is corrent, verified by bubble sort.
This is the END of the program.
jpwang@workbench:~/git_tutorial/assign_q2_solutions$
jpwang@workbench:~/git_tutorial/assign_q2_solutions$
```

Fig. 2.2 Sample output (n=1, max_num=4)

Q2.2 (35%). Sort (4^n)*max_num integers using (4^n) processes. In this case, **n (n>=1)** and **max_num (>=4)** are specified by users. Same as Q2.1, you should create a shared memory segment to store the input integer array containing **(4^n)*max_num** integer values. The shared array should be divided into (**4^n**) smaller segments of the same size, which is equal to **max_num**. Each small segment is first sorted by a single process, and iteratively merged by their parent process using 4-way merge-sort until only a single array is left.

As an example, **Fig. 2.3** shows the relationship of the processes. With **n=2**, we have a total of **4^2 =16** processes (You should manage to create the rest of 15 child processes). All the 16 processes will start with sorting **max_num** number of integers using a local 4-way merge-sort (Discussed in Q2.1. Also read "Q2 Requirements" for more details). Then Process 1, 2, 3, and 4 should merge the sorted segments generated by their 3 child processes and themselves. For example, Process 1 should merge the sorted segments from process 5,6,7, and the part sorted by its self to generate a sorted array of size (**4x max_num**). Finally, Process 1 merges the four segments of size (**4x max_num**) to generate the final sorted array.

Note: you should keep all the processes do their local sort concurrently to improve the processing speed (as these processes could potentially be assigned to different cores for execution). Otherwise, some marks will be deducted (e.g., local sorting on process 5, 6, 7 are done one after the other (sequentially); or the merge steps at process 1, 2,3, and 4 cannot be done concurrently.)

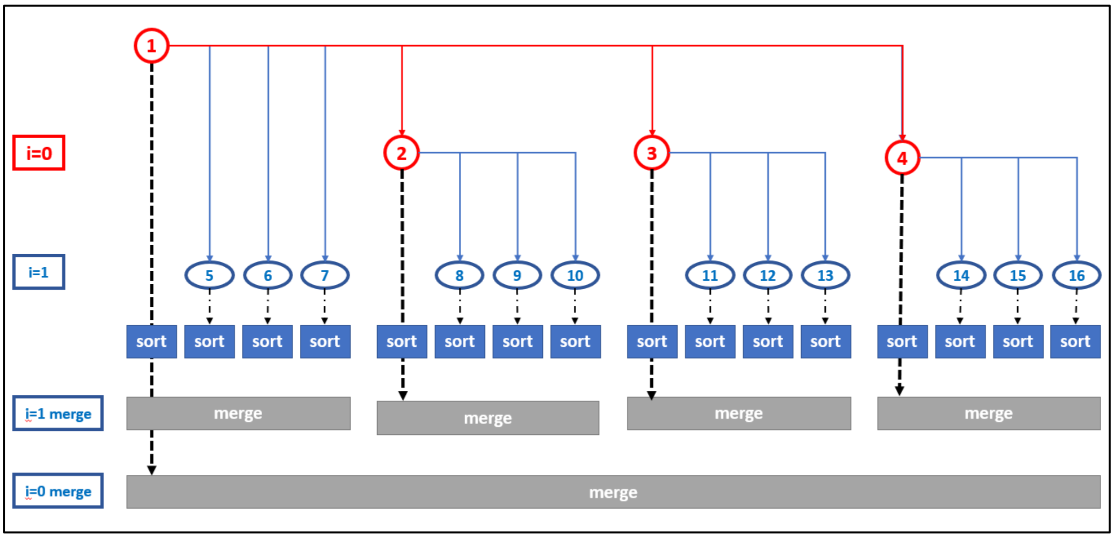
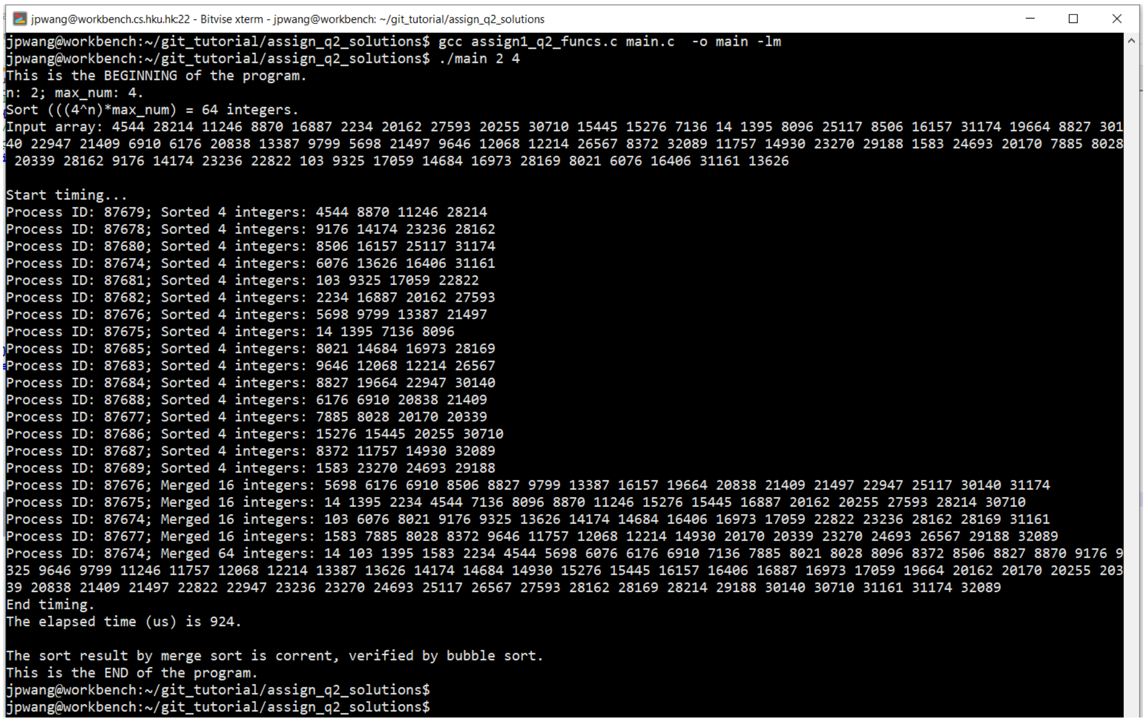


Fig. 2.3 Parallel Merge Sort with Shared Memory (n=2).



```
jpwang@workbench:~/git_tutorial/assign_q2_solutions$ gcc assign1_q2_funcs.c main.c  -o main -lm
jpwang@workbench:~/git_tutorial/assign_q2_solutions$ ./main 2 4
This is the BEGINNING of the program.
n: 2; max_num: 4.
Sort (((4^n)*max_num) = 64 integers.
Input array: 4544 28214 11246 8870 16887 2234 20162 27593 20255 30710 15445 15276 7136 14 1395 8096 25117 8506 16157 31174 19664 8827 301
40 22947 21409 6910 6176 20838 13387 9799 5698 21497 9646 12068 12214 26567 8372 32089 11757 14930 23270 29188 1583 24693 20170 7885 8028
 20339 28162 9176 14174 23236 22822 103 9325 17059 14684 16973 28169 8021 6076 16406 31161 13626

Start timing...
Process ID: 87679; Sorted 4 integers: 4544 8870 11246 28214
Process ID: 87678; Sorted 4 integers: 9176 14174 23236 28162
Process ID: 87680; Sorted 4 integers: 8506 16157 25117 31174
Process ID: 87674; Sorted 4 integers: 6076 13626 16406 31161
Process ID: 87681; Sorted 4 integers: 103 9325 17059 22822
Process ID: 87682; Sorted 4 integers: 2234 16887 20162 27593
Process ID: 87676; Sorted 4 integers: 5698 9799 13387 21497
Process ID: 87675; Sorted 4 integers: 14 1395 7136 8096
Process ID: 87685; Sorted 4 integers: 8021 14684 16973 28169
Process ID: 87683; Sorted 4 integers: 9646 12068 12214 26567
Process ID: 87684; Sorted 4 integers: 8827 19664 22947 30140
Process ID: 87688; Sorted 4 integers: 6176 6910 20838 21409
Process ID: 87677; Sorted 4 integers: 7885 8028 20170 20339
Process ID: 87686; Sorted 4 integers: 15276 15445 20255 30710
Process ID: 87687; Sorted 4 integers: 8372 11757 14930 32089
Process ID: 87689; Sorted 4 integers: 1583 23270 24693 29188
Process ID: 87676; Merged 16 integers: 5698 6176 6910 8506 8827 9799 13387 16157 19664 20838 21409 21497 22947 25117 30140 31174
Process ID: 87675; Merged 16 integers: 14 1395 2234 4544 7136 8096 8870 11246 15276 15445 16887 20162 20255 27593 28214 30710
Process ID: 87674; Merged 16 integers: 103 6076 8021 9176 9325 13626 14174 14684 16406 16973 17059 22822 23236 28162 28169 31161
Process ID: 87677; Merged 16 integers: 1583 7885 8028 8372 9646 11757 12068 12214 14930 20170 20339 23270 24693 26567 29188 32089
Process ID: 87674; Merged 64 integers: 14 103 1395 1583 2234 4544 5698 6076 6176 6910 7136 7885 8021 8028 8096 8372 8506 8827 8870 9176 9
325 9646 9799 11246 11757 12068 12214 13387 13626 14174 14684 14930 15276 15445 16157 16406 16887 16973 17059 19664 20162 20170 20255 203
39 20838 21409 21497 22822 22947 23236 23270 24693 25117 26567 27593 28162 28169 28214 29188 30140 30710 31161 31174 32089
End timing.
The elapsed time (us) is 924.

The sort result by merge sort is corrent, verified by bubble sort.
This is the END of the program.
jpwang@workbench:~/git_tutorial/assign_q2_solutions$
jpwang@workbench:~/git_tutorial/assign_q2_solutions$
```

Fig. 2.4 Sample output (n=2, max_num=4)

Q2.3 (20%) Performance Evaluation: Compare the execution time of the parallel 4-way merge-sort algorithm you implemented in Q2.2 with that of a sequential bubble sort (code is given at the end of assign1_q2_funcs.c). You should include the four test cases: **(1)** n=1, max_num= 256 (sorting 1024 integers), **(2)** n=2, max_num=4096 (64K integers), **(3)** n=3, max_num= 1024 (total 64K integers). **(4)** Can you find a proper size of **n** and **max_num** to show your parallel 4-way merge-sort can **run faster than the sequential bubble sort**. Report the speedup (= T(sequential bubblesort)/ T(your parallel 4-way merge-sort)). Add screenshots to show the execution time for both sorting algorithms and give your discussion and analyses. If you cannot find it, please discuss the key reasons. **[An example of measuring execution time is also provided in** assign1_q2_main_template.c**. Don't forget to remove all the printf() commands in your code before the time measurement as the printing time should not be counted.**

Q2 Requirements:

- For Q2.1, encapsulate your solution to the function in assign1_q2_funcs.c:

  void mergesort4Way4Processes(int* array, int low, int high)

  Write a program, for **Q2.1** based on the given template file (assign1_q2_main_template.c as shown in Fig. **2.7)** which receives two integers (which are **n=1**, max_num) from the command line, and call ***mergesort4Way4Processes*** in this program to finish the sort job.

- For **Q2.2**, encapsulate your solution to the function in assign1_q2_funcs.c:

  void recursiveMergesort(int* array, int low, int high, int max_num)

  Write a program for **Q2.2** based on the given template file (assign1_q2_main_template.c as shown in Fig. **2.7)** which receives two integers (which are n, max_num) from the command line, and call ***recursiveMergesort*** in this program to finish the sort job.

- Submit 5 files for Q2:
  - ✧ assign1_q2_1.c (for Q2.1, based on assign1_q2_main_template.c)
  - ✧ assign1_q2_2.c (for Q2.2, based on assign1_q2_main_template.c)
  - ✧ assign1_q2_funcs.h
  - ✧ assign1_q2_funcs.c
  - ✧ assign1_q2_3.pdf (For Q2.3 Performance Evaluation, you should answer this question in a document and upload it as a pdf file)

```c
int main(int argc, char* argv[])
{
    printf("This is the BEGINNING of the program.\n");
    if(argc-1 != 2){
        printf("Error: The number of input integers now is %d. Please input 2 integers.\n",argc-1);
        return -1;
    }// Don't modify this Error Checking part
    else{
        printf("n: %s; ", argv[1]);
        printf("max_num: %s.\n", argv[2]);
    }
    const int n = atoi(argv[1]);
    const int max_num = atoi(argv[2]);

    int num_integers = pow(4, n) * max_num;
    printf("Sort (((4^n)*max_num) = %d integers.\n", num_integers);
    int* pInputArray = generateIntArray(num_integers);
    printf("Input array: ");
    printArray(pInputArray, 0 , num_integers);
    printf("\n");
```

Don't modify this part.

```c
    struct timespec start, end;
    printf("Start timing...\n");
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    sleep(1);  // This line is only for test. Remove this line when you implement your solution

    //mergesort4Way4Processes();
    //recursiveMergesort();

    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
    printf("End timing.\n");
    uint64_t delta_ms = (end.tv_sec - start.tv_sec) * 1.0e3 + (end.tv_nsec - start.tv_nsec) * 1.0e-6;
    printf("The elapsed time (ms) is %lu \n", delta_ms);
    // uint64_t delta_us = (end.tv_sec - start.tv_sec) * 1.0e6 + (end.tv_nsec - start.tv_nsec) * 1.0e-3;
    // printf("The elapsed time (us) is %lu \n", delta_us);
    // uint64_t delta_s = (end.tv_sec - start.tv_sec);
    // printf("The elapsed time (s) is %lu \n", delta_s);


    bubble_sort(pInputArray, num_integers);
    //verifySortResults(pInputArray, YOUR_ARRAY, num_integers); // Replace YOUR_ARRAY by your array name

    free(pInputArray);
    printf("This is the END of the program.\n");
    return 0;
}
```

Fig. 2.7 assign1_q2_main_template.c

```c
#ifndef _ASSIGN1_Q2_FUNCS_H_
#define _ASSIGN1_Q2_FUNCS_H_

#include <stdbool.h>

// array utils
int  rand();
int* generateIntArray(int size);
void printArray(int* array, int low, int high);

// merge sort: tutorial 1
void merge_4_way(int* array, int low, int mid1, int mid2, int mid3, int high);
void mergesort_4_way_rec(int* array, int low, int high);

// bubble sort
void bubble_sort(int *array, int size);
bool verifySortResults(int* array_bubble, int* array_mergesort, int size);  // verify merge sort by bubble sort

/*****
    Don't modify functions above.
*****/

// merge sort
void recursiveMergesort(int* array, int low, int high, int max_num);
void mergesort4Way4Processes(int* array, int low, int high);


#endif
```

Fig. 2.8 assign1_q2_funcs.h

```c
bool verifySortResults(int* array_bubble, int* array_mergesort, int size)
{
    int num_unequal = 0;
    for(int i = 0; i< size; i++){
        if(array_bubble[i] !=   array_mergesort[i])
            num_unequal++;
    }

    if(num_unequal!=0){
        printf("The sort result by merge sort is not correct. The number of unequal values: %d.\n", num_unequal);
        return false;
    }
    else{
        printf("The sort result by merge sort is corrent, verified by bubble sort.\n");
        return true;
    }
}


void mergesort4Way4Processes(int* array, int low, int high)
{
    // Q2.1: Write your solution



}

void recursiveMergesort(int* array, int low, int high, int max_num);)
{
    // max_num: the maximum number of integers a process can handle
    // Q2.2 Write your solution


}
```

Fig. 2.9 Part of assign1_q2_funcs.c