Fall 2021 CSci 4061: Introduction to Operating Systems
Project #1 – Parallel Multiway Merge Sort

Instructor: Abhishek Chandra

Interim Submission Due: 11:59 pm, Oct. 6 (Wed.), 2021
Final Submission Due: 11:59 pm, Oct. 13 (Wed.), 2021

1. Background

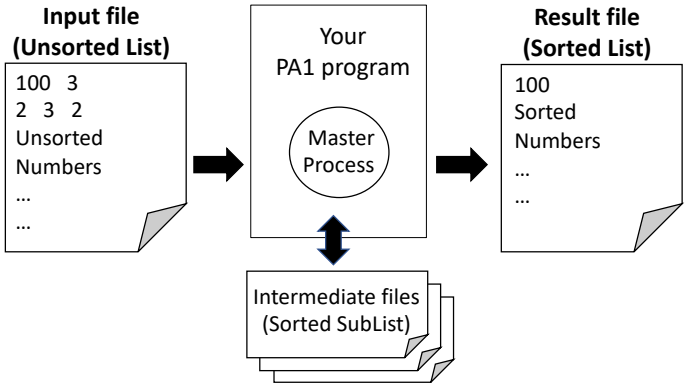
Merge sort is an effective, general-purposed, and comparison-based sorting algorithm, which is based on a divide and conquer concept. A typical merge sort divides the unsorted input into N sublists containing one element (one element is considered as sorted) and then repeatedly merge the sublists to generate sorted sublists until it has one sorted output. For better parallelism, merge sort can utilize multiple processes to perform the algorithm in parallel. In this programming assignment, you will implement a variant of parallel multiway merge sort to sort numbers in ascending order.

2. Project Overview

A typical merge sort uses a binary merge method by recursively dividing an original unsorted list into left and right sublists until the smallest unit (one element) and merging them into one final sorted list, whereas **multiway merge sort** has N merge branches instead of 2. In this project, you will use process-related functions such as fork(), exec() and wait() to make the multiway merge sort run in parallel. Each sublist is sorted by a spawned child process and merged by a parent process. Child process creation is based on "**division instruction**" provided along with input data. More details are described in Section 2.2 Process Tree Generation based on Division Instruction. Apart from merge sort algorithm, another algorithm called "**leaf node sorting algorithm**" (e.g. quicksort, insert sort, etc.) is required for leaf node because the last sublists may not always have one element unlike typical multiway merge sort. More details are discussed in Section 2.3 Process Execution.

2.1. Input and Output

Your PA1 program will have one input file and generate one or more output files as shown in Figure 1. One result file should be included in the output and there will be zero or more intermediate files depending on the number of child processes spawned by the master process (the main process of your program). Input file exists in the "input" folder, and intermediate and result files should be written in the "output" folder. You can find the details of folder structure in Section 4 Folder Structure.



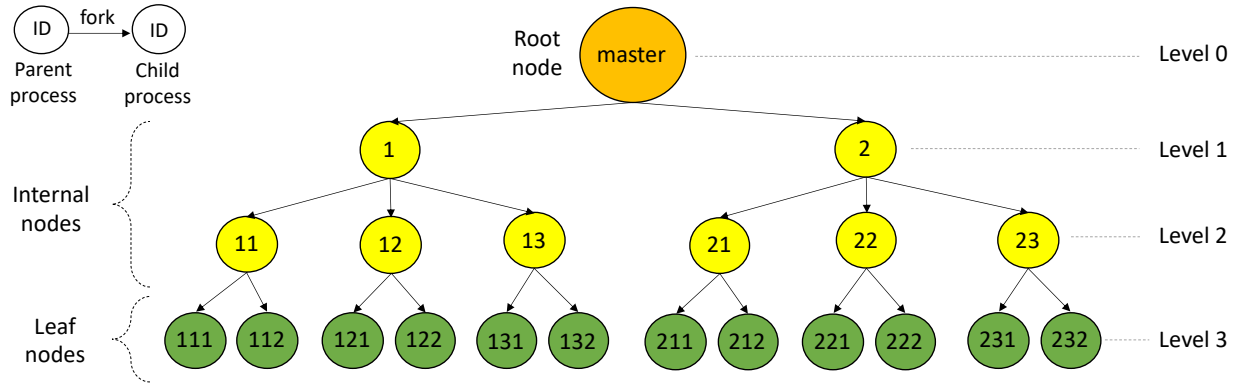
< Figure 1: Input and output files >

Input file contains essential information regarding **division instruction** as well as unsorted input data. The first line of the input file consists of two numbers, **the total number of input data** and **the depth of process tree**. The second line of the file contains **the degrees of each level** of the tree. If the depth is three, then there will be three numbers in the second line as shown in Figure 1. If the depth is zero, then the second line will be empty. The numbers in the first and second lines are separated by space. Unsorted input data are stored from the third line and to the end of file (one value per line).

Intermediate files are generated by child processes to store sorted results for their portion of input data if any, and the result file is generated by the master process to store sorted results for whole input data. **The first line of these files should contain the number of data that each process has sorted on.** Sorted numbers are stored one per line from the second line. That is, the number of data of input and result files should be the same, and the numbers of intermediate files should be the same or smaller than the number of input data.

2.2. Process Tree Generation based on Division Instruction

The division instruction stated in the example input file in Figure 1 is **depth** 3 and **degrees per level** 2/3/2. The master process is the sole **root** node at level 0. Based on the instruction, it spawns 2 child processes (**internal nodes** at level 1), and each child process at level 1 spawns 3 their own child processes (internal nodes at level 2), and the processes at level 2 spawn their own child processes (**leaf nodes** at level 3). Figure 2 depicts the final process tree generated as per the given division instruction.



< Figure 2: Process tree example
generated as per a division instruction (depth 3, 2/3/2 degree per level).
Circles are processes, strings in the circles are their ID, and
arrows indicate parent-child process relationships created by using fork() >

Each process has their ID, which is not a pid of a process. You should assign a unique ID for each process when they are spawned according to the direction described in this paragraph. Figure 2 will help you to understand how to generate an ID for a process. Master process's ID is "master", and its child processes have a unique number. The IDs of processes at level 1 starts from 1 and increments by 1. The IDs of their child processes can be generated by concatenating their parent process ID with a unique number starting from 1 and incrementing by 1.

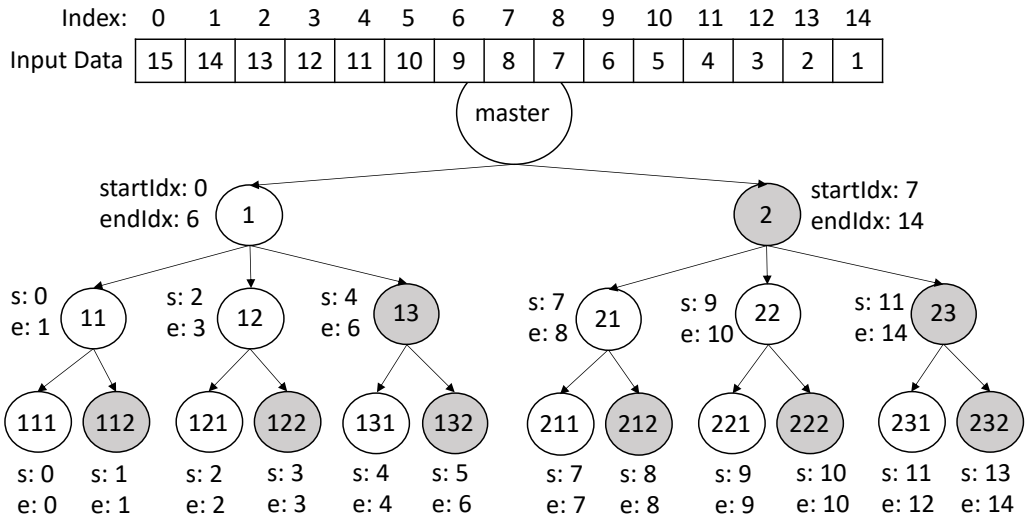
2.3. Process Execution

A parent process spawns their child processes using fork() function and **wait until all their child processes finish their tasks**. A new program called "**childProgram**" is **launched in the child processes by using an exec() function.** The "childProgram" program sorts the assigned subset of input data and stores the sorted results into their own intermediate files in the "output" folder. Intermediate file name is "**<their ID>.out"**

like 1.out, 2.out, …, 231.out and 232.out. If child processes are leaf nodes, they use a **leaf node sorting algorithm** such as quicksort, whereas the other processes perform merge sort based on multiple data streams from the intermediate files generated by their child processes once the child processes are terminated. You can choose any sorting algorithm for the leaf node sorting. The final sorted results are written in "**master.out**" file by the master process. The number of files in the "output" folder is equal to the number of the spawned child processes plus one (master.out). Note that, if depth is zero in the division instruction, then master does not spawn any child processes, so it becomes a leaf node. Therefore, the master sorts input data using the leaf node sorting algorithm and there is no need to merge at all.

2.4. Data Partition

When a process spawns child processes, its unsored input data is evenly distributed to their child processes by sending start and end indices of a portion of input data assigned to each child process. A very simple data partition algorithm is used for this project. When a process has data size of M, and K child processes, all child processes except the last one are assigned **floor(M/K)** data. The last child is assigned the **residual data (M – (K-1) * floor(M/K))**. Figure 3 shows an example. Last child processes are 2, 13, 23, 112, 122, 132, 212, 222 and 232 processes colored in grey, and it shows start and end indices of partitioned data for each process. Data size of process 2 is 8 (= 15 – (2-1) * floor(15/2)). It has 3 child processes and last process is process 23. Process 21 and 22 are assigned data size of 2 (= floor(8/3)) and process 23 is assigned the residual data size of 4 (= 8 – (3-1) * floor(8/3)).



< Figure 3: Data partition >

3. Log Printout

Print the following logs to terminal. You need to replace "Quick Sort" with the leaf node sorting algorithm that your group uses.

- **When a process P spawns a child process C,**
  - `printf("Parent [%s] – Spawn Child [%s, %s, %s, %s, %s]\n", process P's ID, Level of process C, process C's ID, start index, end index, and the size of the data assigned to process C)`
- **When a merge sort finishes,**
  - `printf("Process [%s] – Merge Sort – Done, the process's ID)`
- **When a leaf node sorting finishes (Quick Sort is used for an example),**
  - `printf("Process [%s] – Quick Sort – Done, the process's ID)`

For example, when the master has input data size of 23 and a division instruction (depth 1, degree 3), logs printed to the terminal are like below.

```
Parent [master] - Spawn Child [1, 1, 0, 6, 7]
Parent [master] - Spawn Child [1, 2, 7, 13, 7]
Parent [master] - Spawn Child [1, 3, 14, 22, 9]
Process [1] - Quick Sort - Done
Process [3] - Quick Sort - Done
Process [2] - Quick Sort - Done
Process [master] - Merge Sort - Done
```

4. Folder Structure

Please strictly conform with the folder structure. The conformance will be graded.
You should have a project folder named "PA1_G[Your group number]" (e.g. PA1_G23 for project group 23). The project folder contains "**include**", "**lib**", "**src**", "**input**", "**output**", and "**expected**" folders, as well as Makefile, README and two executables.

- "**include**" folder: .h header files
- "**lib**" folder: .o library files
- "**src**" folder: .c source files
- "**input**" folder: 9 input examples (input*.file).
- "**output**" folder: a result file (master.out) and intermediate files (<ID>.out) if any.
- "**expected**" folder: 9 subfolders containing the expected results of 9 input examples.
- Two executables: master and childProgram

5. Execution Syntax

The usage of your pa1 program is as follows. The executable name should be "**master**" and it should be in the same folder where the "input" folder exists.

```
./master <Input File Name>
```

- <Input File Name>: An input file name (no file path). The input file should be in the "input" folder.

The name of executable launched in child processes by exec() function call should be "**childProgram**". You can design your own argument passing through exec() function to decide which information should be passed to child processes through the arguments of the program.

6. Assumptions and Hints

- All processes can access to the input file.
- 1 <= Input data size <= 1000.
- 0 <= Input data value <= 10000.
- The number of input data is equal to or greater than the number of leaf nodes.
- Minimum depth is 0 and maximum depth is 9.
- Minimum degree is 1 and maximum degree is 9.
- You are given utils.o library including file I/O related functions. You can find function signatures in utils.h and code snippets in master.c and myutils.c to see how to use the given functions.
- You can modify the given code skeleton, files, functions, and code snippets as you need. Just make sure to modify headers and Makefile accordingly.
- Your program should run successfully using execution commands "make run*" stated in Makefile

- We will use the GCC version installed on the CSELabs machines(i.e. 9.3.0) to compile your code. Make sure that the code you submit compiles and run on CSELabs: your code compile simply by running make.
- Please make sure that your program works on the CSELabs machines e.g., KH 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

7. Submission

7.1 Interim Submission

You need to implement 3 level-1 child process creation. Assume that you are given a division instruction (depth 1, degree 3). It is okay to hardcode the numbers for interim submission.
More specific requirements are:

- Master process spawns 3 child processes as leaf nodes at level 1 by using fork() function.
- A program named “childProgram” is launched in the child processes by using an exec() function.
- The “childProgram” program simply prints “[Leaf] Quick Sort”; the main function has printf(“[Leaf] Quick Sort\n”);
- Master process waits all 3 child processes to terminate by using a wait() function, and then prints “[Master] Merge Sort” (printf(“[Master] Merge Sort\n”);) before the main function returns.

Interim submission should be a zip containing two .c files (master.c and childprogram.c) and an interim report (pdf). Interim report should include:

- Project group number
- Members’ name and X500.
- Plan on individual contributions
- Leaf node sorting algorithm that your group will use
- Captured screen shot showing terminal output. If the requirements are correctly implemented, three “[Leaf] Quick Sort” and one [Master] Merge Sort” messages are printed on terminal.

Due for the interim submission is **Oct. 6, 2021 (Wed), 11:59pm.**

7.2 Final Submission

One student from each group should upload to Canvas, **a zip file** containing the project folder. README should include the following details:

- Project group number
- Group member names and x500 addresses
- Members’ individual contributions
- Leaf node sorting algorithm that your group uses
- Any assumptions outside this document
- How to compile your program
- How to run your program

The README file does not have to be long, but must properly describe the above points. Your source code should provide **appropriate comments** for functions. At the top of your README file and each C source file please include the following comments:

```
/*test machine: CSELAB_machine_name
* group number: G[Group Number]
* name: full_name1 , [full_name2]
* x500: id_for_first_name , [id_for_second_name] */
```

Your project folder may have all folders and files mentioned in Section 4 Folder Structure. However, **please DO NOT include "input", "output", and "expected" folders in your final deliverable**. You can modify the provided Makefile, but just make sure your programs should be successfully compiled using Makefile.

Due for the final submission is **Oct. 13, 2021 (Wed.), 11:59pm.**

8. Grading Policy (tentative)

1. (10%) Interim submission
    a. Correct report content
    b. Completeness of minimum implementation requirements
2. (10%) Appropriate code style and comments
3. (5%) Conformance check
    a. Correct README content
    b. Folder structure and executable names
    c. Log format
    d. The file names of Intermediate and final files
4. (50%) Testcases
    a. input1.file: Depth 0 process tree (no child process spawned)
    b. input2.file: Degree 9 process tree
    c. input3.file: Depth 3 process tree
    d. input4.file: Depth 9 process tree
    e. input5.file: 1000 input data
5. (10%) Code
    a. Correct use of process-related functions such as fork(), wait() and exec()
6. (15%) Error Handling – You need to print any error message for each error case and terminate your pa1 program. Refer to the expected output provided.
    a. Input6.file: No input data
    b. input7.file: depth is greater than 9
    c. input8.file: # of leaf nodes is greater than # input data
    d. input9.file: degree of a level is not between 1 and 9
    e. Error handling when process-related functions fail