Readme File | Assignment Two

Abdullah Javaid Chaudhry | AJC957

Invocation Details

My program will be compiled as the executable myhie, and this executable allows the sorting of records by either of the four attributes residentID, number of dependents, income, or zip code. My program can accept 9, 10, 11, or 12 command line parameters where the parameters can be the flags “-i”, “-k”, “-r”, “-a” “-o” “-s” in no particular order, it being only necessary that the argument that follows the flags “-i”, “-k”, “-a” “-o” “-s” be valid and appropriate. Disregarding all permutation of order, my program can be executed in one of four primary ways, as below:

```
        $ myhie -i InputFile -k NumOfWorkers -r -a AttributeNumber -o Order -s OutputFile
        $ myhie -i InputFile -k NumOfWorkers -a AttributeNumber -o Order -s OutputFile
        $ myhie -i InputFile -k NumOfWorkers -r -a AttributeNumber -o Order
        $ myhie -i InputFile -k NumOfWorkers -a AttributeNumber -o Order
```

The flags “-i”, “-k”, “-a”, and “-o” are mandatory, otherwise it does not make sense to invoke this program and sort records, because we do not know what it is we want to sort (the input file), how it is we want to sort it, in what order (ascending or descending), and using how many workers.
In the absence, of the “-r” flag, the sorters will be given uniform ranges.
In the absence, of the “-s” flag, the output file will take as a default the value, output.txt.
AttributeNumber can only be 0, 3, 4, or 5.
Order can only be ‘a’ or ‘d’ and this corresponds to ascending and descending.
OutputFile cannot begin with a “-” because that is used to indicate flags.

Program Structure

My assignment submission contains the following .c files:

- The “main.c” file contains the program implementation for the program of the root node, parses and validates the user input with the further program only invoked if the user input is valid. The processed user input and the flags set will all be displayed. Given that this is the root, it will be handling the signals generated by each of the sort nodes (SIGUSR1) and the merge node (SIGUSR2) reporting how many signals of each have been caught.
- The “coord.c” file contains the program implementation of the coordinator node, the coord first generates sorting intervals (random or not based on the user input), and then iteratively forks and invokes the mergeSort program or the bubbleSort program (the choice between the two being made depending on whether the sort node is even or not) with their specific ranges. The ranges are displayed for user verification in the format “x [y, z)” where y is the starting range for sort node number x, and z is the end range for sort node number z. The coord also invokes the merge node program.
- The “bubbleSort.c” contains the program implementation for one of the two sorter nodes, and this particular sorter node type employs the simple bubble-sort algorithm. It will send timing information using the TIMINGFIFO ("/tmp/timings") to the merge node, while all the data that has been sorted will be sent using the RESULTFIFO ("/tmp/results") to the merge node. A SIGUSR1 signal addressed to the root node will then be generated, immediately before program termination.
- The “mergeSort.c” contains the program implementation for the other of the two sorter nodes, and this particular sorter node type employs the efficient merge-sort algorithm. Identical to the bubble sort program, it will send timing information using the TIMINGFIFO ("/tmp/timings") to the merge node, while all the data that has been sorted

will be sent using the RESULTFIFO ("/tmp/results") to the merge node. A SIGUSR1 signal addressed to the root node will then be generated, immediately before program termination.

- The “merge.c” file contains the program implementation for the merger node, this will first read all the data from the RESULTFIFO into a two-dimensional array until all the records across all sorting nodes have been read. The two-dimensional array is such that the zeroth element of the array will contain the records from the zeroth sorter, and so on. This two-dimensional array will then be merged into a single array of pointers, by selecting the most appropriate (smallest or largest record using the requisite attribute number depending on whether the sorting mode is ascending or descending) at every turn. The data pointed to by this sorted array will then be written out to the output file. The timing information will also be received from the sort nodes, and displayed appropriately. A SIGUSR2 signal addressed to the root node will then be generated, immediately before program termination.

Note, that using the data file containing the million records can require a lot of time, depending on the number of sorters one has, this is because bubbleSort has O(n^2) which proves very slow on such a large amount of data. In the absence of any printing to the screen, the time required for processing (which can be up to 5 minutes) might cause the program to appear unresponsive to a user even though work is ongoing. To reassure the user that work is ongoing, I have added two print statements, one of which prints out the number of records received after the receipt of every thousand records, and the symbol “.” when the reading part of the merge node is starved of data. This starving happens only temporarily until the sending of the sorters catches up to the reading of data by the merge, this because merge can read data much quicker than the bubble-sort based sort node can send causing merge to starve. Note that we cannot choose to not wait for bubble-sort based sort node, because we must read all the data before we can hope to merge everything because if we were to operate greedily, then we would begin merging before we collect data from the slower sorters while it is possible they might have a more smaller value which would be very problematic (assuming an ascending order sort).The print calls naturally do slow the program somewhat, but their use is necessary as a comfort and a progress bar.

The creation of all the above files was required in by the assignment and with the exception of certain design choices (elaborated later) follow largely the spirit and structure of the assignment prompt, meeting all requirements therein, rendering explanations of their outright existence somewhat superfluous. That said, there are additional .c files, namely “helper.c” and “sortHelper.c” whose existence could bear greater explanation.

The file “helper.c” is not a program in of itself and instead represents a useful collection of functions and structures. It is only partially compiled to a “.o” file, and is not an executable. The functions included in “helper.c” are those functions that are helpful across all the formerly mentioned programs such as the “integerToString” function which converts integers into strings (and is essential for when invoking programs using exec and passing integer data), or the “comparator” function. The “comparator” function enables comparison of two records and their information across the specified attribute number using the specific comparison of any of the six comparison modes from “==”, “!=”, “>”, “>=”, “<”, “<=”. This is incredibly helpful because sorting must be performed frequently throughout the assignment and it is easier to abstract away the comparison rather than acutally implement it every time. The “helper.c” additionally also contains the following structures:

- “UsrRecord” – used to store all the data regarding an individual namely his resident ID, first name, last name, number of dependents, income, and postal code.
- “Message” – used to store certain details of a sorter, its worker ID, and the total number of records that that sorter has sorted. The worker ID is imagined to be allocated as an identification number of each sort process based on its invocation number in the coord process, in that the zeroth sort process will be allocated worker ID equal to 0, and so on.
- “Timing” – used to stored certain details of a sorter, its worker ID, the real time the sort process took, and the CPU time the sort process took. Note, that the time record is only the time required in the sorting, and excludes that taken in the sending of the data by the sorter, and this is in line with the assignment specifications. This is used for inter-process communication between the sorter nodes and the merge node specifically for the transfer of data containing timing information, which is supposed to be and is later displayed.
- “DataTransferRecord” – used for inter-process communication between the sorter nodes and the merge node specifically the transfer of the data of records after they have all been sorted by a particular sort node. This structure stores the sorterIndex (an alias for worker ID) so as to identify from which sorter node is it specifically that the current “DataTransferRecord” is received, totalRecordCount so as to identify how many records has the sorter sorted in total (this will be used to create an array for storing the partial sort results from each sorter in the merge node), recordIndex to identify what entry (the present UsrRecord whose data is being sent) is in the original sort process, and lastly tRecord which is a UsrRecord and stores the actual data whose sending is required.

Alternately, the “sortHelper.c” stores the common functions across both sorter programs, such as the very important arrayMake function which generates an appropriate array of UsrRecords from the input file where the first record lies at the starting range point of the sort function until the last record that immediately precedes the ending range point.

Some Design Considerations:

My two sorting algorithms are merge-sort and bubble-sort, the choice was deliberate to highlight the poor performance of $O(n^2)$ sorting algorithms relative to $O(n\log(n))$ sorting algorithms.

The most important thing in this assignment is that the sorters be able to operate independently of one another, non-sequentially. Now, whether this is possible or not, depends on the structure of one’s program, specifically that structure involving named pipes since these can block program execution until both ends of the pipe are open. I had earlier used non-blocking pipes with each sorter node having its own pipe in which it was to write out its records, these pipes were then being read from the merge node with the help of the select system call. This enabled asynchronous execution of all the sort processes, however, when using the larger dataset, not all records sent from the sorters would reach to the merge node, and this was I later learnt due to 4 KiB atomic write size limitation. This attempt can be found at the end of my merge.c file.

Now since, individual non-blocking pipes did not work out, I have presently shifted to the use of a single pipe, RESULTFIFO. Each sorter writes a record as soon as it can to this single pipe with no sorter having to wait for the other to finish its writing, and in this no single sorter is hung up because of the other. To prevent data from one sorter being mixed with that of another in the pipe, an augmented record is actually written. This augmented record contains information about the sorter which sends the record, and this information includes a workerID x which is used to identify the sorter, a recordID y to identify that the data sent is actually record number y from sort node x. Using this, the data for the record is appropriately added to the correct array in the merge node process. Timing information is handled similarly for similar reasons. Note, however that the timing information only includes the time "each sorter took to sort its batch of record" [Assignment Prompt p. 3]

There, is one important remark that I would like to make in parting and that is that not all SIGUSR1 signals are being caught by the root node when a uniform range of records is used, this because when a uniform range, programs of a particular sorting algorithm time will all end at approximately the same time since they work concurrently. Accordingly, multiple SIGUSR1 signals will all be sent at the same time. Reasonably speaking, this should not cause any issue, but the issue is that in C, when processing one signal, there can be at most one other signal instance of the same type since signals are not queued and the buffer of signals can only contain one signal other than that being processed. Hence, the count of signal received at the end of root, does not correspond with the actual number of sorters and SIGUSR1 signals sent. This issue will not occur in a less concurrent approach to things, and Professor Delis has indicated that at this level, the issue is acceptable.