Project Report (Prog04)

1. File structure.

1.1. Main process. (*prog04v03.c*)

The main process can be started by a *shell* script.

e.g.

./script03.sh "path to data directory" "path to output file"

When the path strings contain white spaces like in the above example, it should be surrounded by a pair of double quotes.

When the main process starts, it checks the number of arguments and the validity of each argument.

The first argument is the number of total processes, which are calculated by the shell script by adding 1 to the maximum processor index encountered within the input directory.

The second argument is the path to the input directory which may or may not contain white spaces and trailing slash. (All of these cases are handled.)

The third argument is the path to the output file which may or may not contain white spaces and trailing ".c" extension. (All of these cases are handled.)

When all of these input arguments are validated, the following procedure is followed step by step.

- Create a catalog of file paths from the given input directory.

The catalog of file paths is stored as an array of char * variables. In other words, the catalog is a 2-dimensional char array.

This is done by repeatedly expanding the array size until all data are read into the catalog.

The catalog starts from an empty array with a capacity set at 1.

When the second file path is read, the capacity of catalog is increased to 2 to hold the data.

Similarly, When the third file path is read, the capacity of catalog is increased to 4 to hold the data.

This process repeats until all data are read into the catalog.

- Distribute files to n distributor processes.

The catalog is split into n almost-equal parts using mod-based indexing.

Each part of catalog is written into a datafile. (*0.catalog, 1.catalog*, ...)

n distributor processes are executed using *execvp*() in each own child process using *fork*(). (ref. 1.2)

The main process waits until all distributor processes are finished.

- Process files using n processors.

n processors are executed using *execvp*() in each own child process using *fork*(). (ref. 1.3)

The main process waits until all processors are finished.

- Collate and output result.

Each processor outputs its processed result to a text file. (*0.result, 1.result*, ...)

The main process reads all data from each text file one by one.

It collates the data into one char array (implemented as a dynamic array with increasing array length).

The collated result is written into the given output file.

1.2. Distribute process.

A distribute process is started by *execvp*() call from the main process.

A distribute process requires 2 input arguments.

The first argument is the number of total processes.

The second argument is the process index of current distribute process.

When all of these input arguments are validated, the following procedure is followed step by step.

- Create n lists of file index from a single text file

The file paths are read from catalog files based using the current process index. (*index.catalog*)

These paths are stored in a 2-dimensional char array (*local catalog*) for later use.

n lists of file index are generated as a 1-dimensional array of n linked lists of *IntNode*. (ref Definition of *IntNode* in Documentation)

Each linked list contains data files to be processed by each processor.

Each node in the linked list is a file index. This file index is can be used to get the file path from the *local catalog*.

- Save the lists into a single text file

After n lists are created, they are written into a text file using the current process index. (*index.list*)

1.3. Processor

A processor is started by *execvp*() call from the main process.

A processor requires 2 input arguments.

The first argument is the number of total processes.

The second argument is the process index of current processor.

When all of these input arguments are validated, the following procedure is followed step by step.

- Read the list of files to be processed from n text files.

  n lists of file index are read from all list files generated by n distribute processes.

  Among these lists, only those which have the same process index as the current process are kept and merged in to a single 2-dimensional array.

- Read files to be processed.

  Each data file in 2-dimensional array is read one by one into an array of FileLine. (From each file, the line index and the line data are stored in FileLine structure.)

  Using a custom comparison function and qsort(), the array is sorted in the increasing order of line index.

  Then all lines are merged into a single string value.

  This value is written to a single text file using the current process index. (index.result)

2. Notes

2.1. Current limitations

The main process only checks if all child processes are finished.

It doesn't know the difference between any process being successfully finished and it being failed to execute.

When a distribute process fails for some reason, it won't generate its corresponding list file, which gives rise to the main process to fail due to missing input file.

The same is true for any processors.

This can be avoided by checking child status in *wait*() call and its return value.

2.2. Crash

A lot of memory allocations and deallocation are employed in this project.

Currently, there are no validation checks in the code to verify that if memory allocations are successful or not.

If the number of input files are increased, it might be possible to overflow the main memory which could lead to crashes.

Other than these, there were no crash cases observed during tests using the given data sets.