OPERATING SYSTEMS

Assignment 2

Pavlos Ntais

Contents

1 General Information

The assignment has been implemented in C. The functionalities described in the assignment, have been implemented. I have also thoroughly tested the program, and it works as intended.

2 Explanation of Directories & Files

I have chosen the following organization for the files:

- **src/coordinator.c**: Implementation of the first level - Coordinator-Splitter/Merger-Reporter.
- **src/splitter.c**: Implementation of the second level - Splitter/Merger.
- **src/quick_sort.c**: Implementation of the quicksort algorithm.
- **src/heap_sort.c**: Implementation of the heapsort algorithm.
- **src/utilities.c**: General utility functions needed at all levels.
- **src/signal_handler.c**: Functions for the signal handling.
- **include**: Header files of the program.

3 Coordinator-Splitter/Merger-Reporter

The coordinator serves as the anchor of the k-way tree hierarchy. It is responsible for the final output of information and coordinates the second level (splitters). Additionally, it is responsible for printing the final, sorted file. The following steps are followed:

1. Read the appropriate command line arguments and perform error checking. Then, assign them to the corresponding variables.
2. Create the necessary pipes so that the splitters pass the sorted records and times (CPU/Real).
3. Calculate the range each splitter will handle and create the necessary arrays to store the records and times.
4. Deploy the splitters (using **fork** and **execvp**) with the appropriate arguments - the number of sorters they will create and the range they will sort.
5. Read the sent records from the splitters (from the pipes) and store them in the arrays.
6. Merge the records and print the final sorted file, the time needed for sorting, and the number of received signals.
7. Release the allocated memory and exit.

4 Splitter/Merger

The splitter is called by the coordinator, and the following steps are followed:

1. Receive the necessary information about the splitter's sorting rage from the command line.
2. Create communication pipes with the sorters to pass the sorted records and sorting times (CPU/Real).
3. Calculate the range the splitter will handle and create the necessary arrays to store the records and times.
4. Deploy the sorter (using **fork** and **execvp**) with the file range to sort.
5. Read the sent records from the sorters (using the pipes) and store them in the arrays.
6. Merge the records, and send them to the coordinator, as well as a **SIGUSR1** signal of completion.
7. Release the allocated memory and exit.

5 Sorter

The sorter, in reality, is either quicksort or heapsort. In the splitter, we fork and execute the appropriate program. The switch between them occurs every time we change the splitter, so we always start with the first sorting algorithm. The sorter reads the part of the file it will sort, performs the sorting, and then sends the result of the sorting to the splitter. Finally, it sends a **SIGUSR2** signal to the coordinator indicating that the execution has completed.

Quicksort: The basic idea is to select an element from the array (called the pivot) and place it in the correct position. Thus, elements with smaller values are on its left, and elements with larger values are on its right. The algorithm is then recursively applied to the two subsets resulting from the partition until the array is sorted. The time complexity of quicksort is O(n logn) on average, while it has a worst-case complexity of O(nˆ2).

Heapsort: Heapsort relies on using a binary tree, known as a heap, to ensure that the smallest element is always at the root. Thus, each time the element at the root is removed, and recursively heapsort is called on the new heap until we create the new sorted array. The time complexity of heapsort is O(n logn) on average, while it has a worst-case complexity of O(n logn).

6 Example

Here we can see an example of the hierarchy, with k = 4.

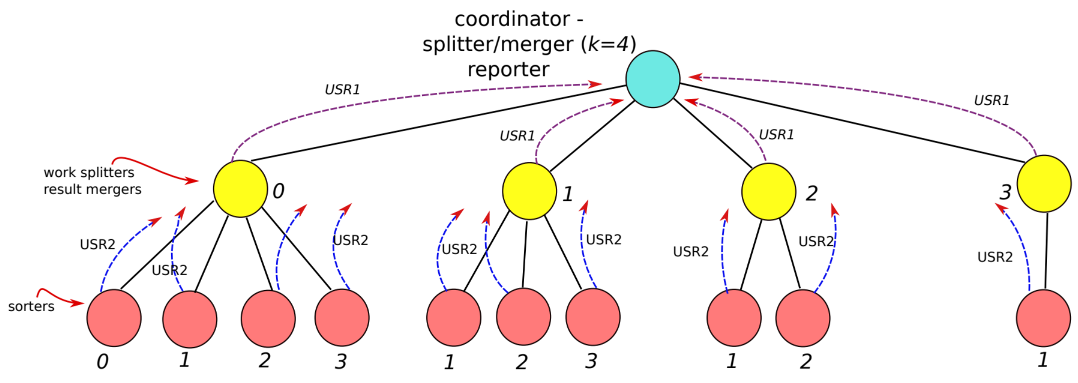

Figure 1: Hierarchy with k = 4

7 Technical Details

1. Each level of the tree represents a separate program. Therefore, there are a total of 3 programs.

2. Reading (**safe_read**) at each level is done using poll to read the results of the child that has finished and sends using write its results via a pipe. Essentially, reading is done asynchronously, and the program does not block waiting to read the results sequentially. The way safe_read works is by iteratively reading as many bytes as are available at that moment in the pipe since a pipe can only "store" a predetermined number of bytes at any given time.

3. The source **utilities.c** file contains useful functions used in all levels. This choice was made to make the code of the other programs more readable, the result of that is somewhat sacrificing the ability of finding specific functions located in a level.

4. I pass the file's name up to the last level of the sorters. I made this decision so that each sorter can independently read its corresponding part of the file by opening it and closing it immediately after reading.

5. Regarding the signals, I chose to use the sigaction function because the kernel interrupts system calls to serve a signal. However, sigaction allows these system calls to run again (except for poll).

6. It is assumed that in cases where a fair distribution cannot be achieved while calculating the range (e.g., having 10 elements and needing to distribute them among 3), the last range takes the additional weight (i.e., 3/3/4).

8 Usage

Execution is done using the command:
./bin/mysort -i **file**.bin -k **number_of_children** -e1 **path_to_sort1** -e2 **path_to_sort2**

In parameters e1 and e2, the path to the appropriate sorting executable file is passed.

The order in which they are passed is important for the alternation of algorithms. If no executable is passed, we default to quicksort.

Though, a makefile is provided with the following parameters:
make - Compile and link the program.
make run - Run the program (with default command line arguments).
make file - Run the program and pass the results to an output.txt file (with default command line arguments).
make help - Run the program using valgrind.
make clear - Delete the object files and the executable generated during compilation.