

Counter

Counter was our Operative System project for the second year of Bachelor in Computer Science in University of Trento (Italy). The main focus of this project is to count the occurrences of chars inside one or more files (normal ASCII). Our idea was to make all as extensible and modular as possible. There are several components, every of them has a specific task. A little brief:

- **Counter:** the counter spawns the reporter and the analyzer.
- **Reporter:** the reporter creates the terminal user interface (**Tui**) and communicates with the analyzer.
- **Analyzer:** the analyzer takes all inputs from the user and finds all files given a directory. Then, while files are being discovered, analyzer sends founded ones to managers.
- **Manager:** the manager takes files and split them in several works. A work starts from a specific point of the file and ends in another. When works are ready they are sent to workers.
- **Worker:** the worker takes the file, the start point end the end point. After that he reads the portion of the file

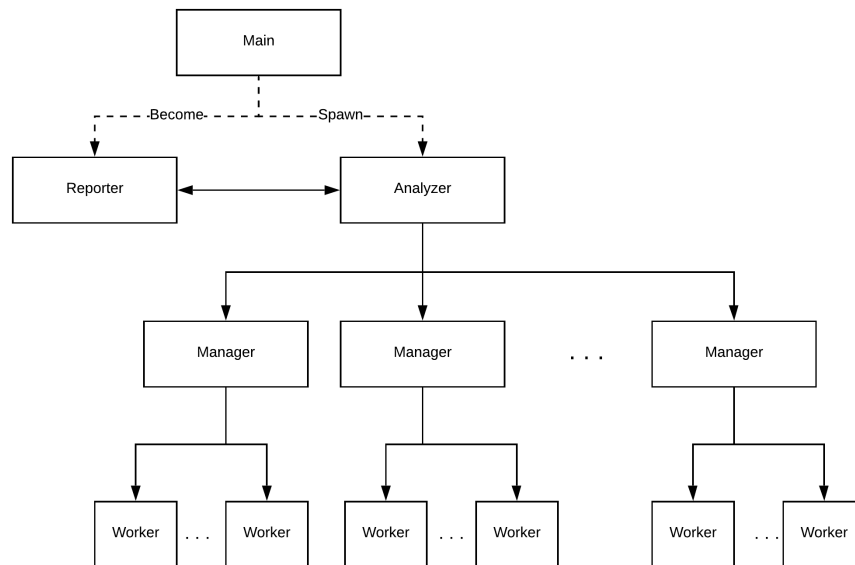


Figure 1: Structure

Obs: With a small amount of work is possible to change the lowest component of the system, the worker, to allow the system to handle different types of problems, not only counting occurrences

Goal

The goal of this project is to learn most of C system calls and to take confidence with GNU/Linux environment.

Implementation choices

There are several implementation choices inside all files in components' folder. Here are listed some common choices.

Memory

Inside all code, except worker, when we allocate some memory on the heap we always control if the memory was allocated correctly. In most cases if malloc fails we closed the program because we think that if a user saturated the RAM, he/she may prefer that some programs could free it to make the computer more usable. The only component that checks the amount of free memory is the worker. Before reading from a file it checks if there is enough free space in memory in order to read its work amount. If not it allocates 50% of the free memory and reads multiples times from the file moving the cursor. This check is made only in the worker because it is the component with the highest memory usage. Nowadays computers have enough memory to support other components and the check seems, for us, an useless overhead. We made some test to prove our decision:

Number of files	size
0	3.93 MiB
1	4.13 MiB
10	4.38 MiB
100	5.19 MiB
1000	13.52 MiB
10000	94.05 MiB

The files were empty and all inside the same folder but we think that, also with different configurations, the memory usage is similar. The amount of memory was calculated after the workers ended their tasks.

Empty folders

We decided not to store information about empty folders. If files are added later in a folder (that was previously empty) user needs to analyze it again

Changing file runtime

If a file is changed while the workers are reading there are several possibilities: * if the updated file is shorter/longer than the old one, we decided to handle the error but the statistic are not reliable, so if the user want the correct one he/she needs to analyze it again. * if the updated file has different permissions, workers are still able to read the file because permissions are only verified when a file is opened and the statistic will be saved. * if the file has been deleted, workers are still able to read the file because as long as there is an open file descriptor the file's data will not be deleted and the inode will not be freed. The statistic will be saved anyway. In any case the purpose of the program is to analyze files, so if the user changes them runtime it completely lose its original meaning (errors must be handled anyway).

Manager/Worker amount changes

If the manager/worker amount changes during the program execution we decided not to kill them (except the case when the user wants less processes, and rare occasions - i.e. communication errors), instead we reorganized their jobs. If one or both of those amounts are equals (or below) zero, their task are stored until the number changes.

File distribution

Files are scheduled to managers using a priority queue. In some cases this is the fairest way to assign them, but, in some occasions (few and/or small files) only a small amount of managers are involved (some/all files are analyzed before the whole distribution). This happens because the priority is based on the amount of file, assigned to the manager, which still needs to be processed. We thought about changing the priority with the total number of file assigned, but this isn't fair with big files and analyzer shouldn't access files in any way to know their dimension.

Thread

We decided to use threads in all components in order to improve the user experience, giving them the impression of running every single component in a concurrent way, just like an Operating System does with processes. Every part of a component seems to be always running, but in reality there are a lot of context switching between several threads.

Known issues

Here are listed some known issues:

- if / is given as path the program interrupt itself in some random moment. This is caused by some strange file inside system's folders. We tried to "make a rule" in order to handle them but there are too many cases (we didn't have neither the time nor the means). Here are listed some that we handled (only for information):
 - there are files that have a specific amount of space on the disk (4096 bytes) but inside the file there is only one char (i.e. /sys/kernel/mm/hugepages/hugepages-2048kB/free_hugepages). Inside manager we first use lseek to compute the dimension but if a worker fails, we try to compute the dimension reading all file and take only the amount of read chars. This is against the professor's directives but is the only way to handle it (i.e. we saw wc unix command source code and also wc use this techniques).
 - there are files that have multiple EOF or other special chars that block the read from the file descriptor before the real amount is read
- if analyzer and reporter are opened in two different terminals there is the possibility to close one and open it again. The two components keep the communication up but if reporter is closed and open again several times very quickly there is the possibility that the analyzer will die without any error message (we spread a lot of error messages inside all the code, but we didn't get the problem)
- The executable files needs to be called inside the bin directory, otherwise the program won't work (except for the reporter and worker).
- There is the possibility that the user clear the FIFO manually. If so the analyzer/reporter communicate will be compromised (we tried to delete FIFOs after creation but other major problems were found)
- If where we create the FIFOs does already exist a file with the same name owned by root with no reading/writing permission for the current user the FIFO can not be accessed

Team

[illegible]

Reporter

The reporter is the component that allows the user to communicate with the chain structure of the system. It is the gateway between the analysis part of the program and the commands of the user. When the reporter is started, depending on the size of the terminal where it is called, starts the system with the tui or with a more basic and less comfortable to use interface. The reporter can receive various commands like send new files to analyze, require the system to compute statistics on files and show the results of the computation. Depending on the command received, the reporter communicates to the analyzer to do what the user want. When the analyzer has some results of the computation, the reporter receives this data and shows them to the user.

Structure

The reporter is composed of 3 threads, but it also uses 2 thread of the tui: * readFifoLoop that reads the analyzer's computed data * writeFifoLoop that sends user's requests to the analyzer * userInputLoop that reads the input of the user and gives feedbacks to the user if necessary * inputLoop (tui) that does the same operation of userInputLoop when the tui mode is used * graphicsLoop (tui) that draw the terminal user interface

Read fifo thread

This thread is charged of checking if the “analyzerToReporter” FIFO file exists and in case of success, the FIFO is opened between the reporter and the analyzer in O_RDONLY in order to read the data sent from the analyzer. There reporter can read 2 type of data:

- with “tabl” the reporter reads the data that analyzer computed for the current requested files and directories.
- with “tree” the reporter reads the files and directory that are contained in the current work directory (used in tui mode only to populate the file system area).

Write fifo thread

In this thread the information requested by the user are sent to the analyzer. First of all, this thread checks if the “reporterToANalyzer” FIFO file exists and in case of success, the FIFO is opened between the reporter and the analyzer in O_WRONLY mode in order to write different commands to analyzer:

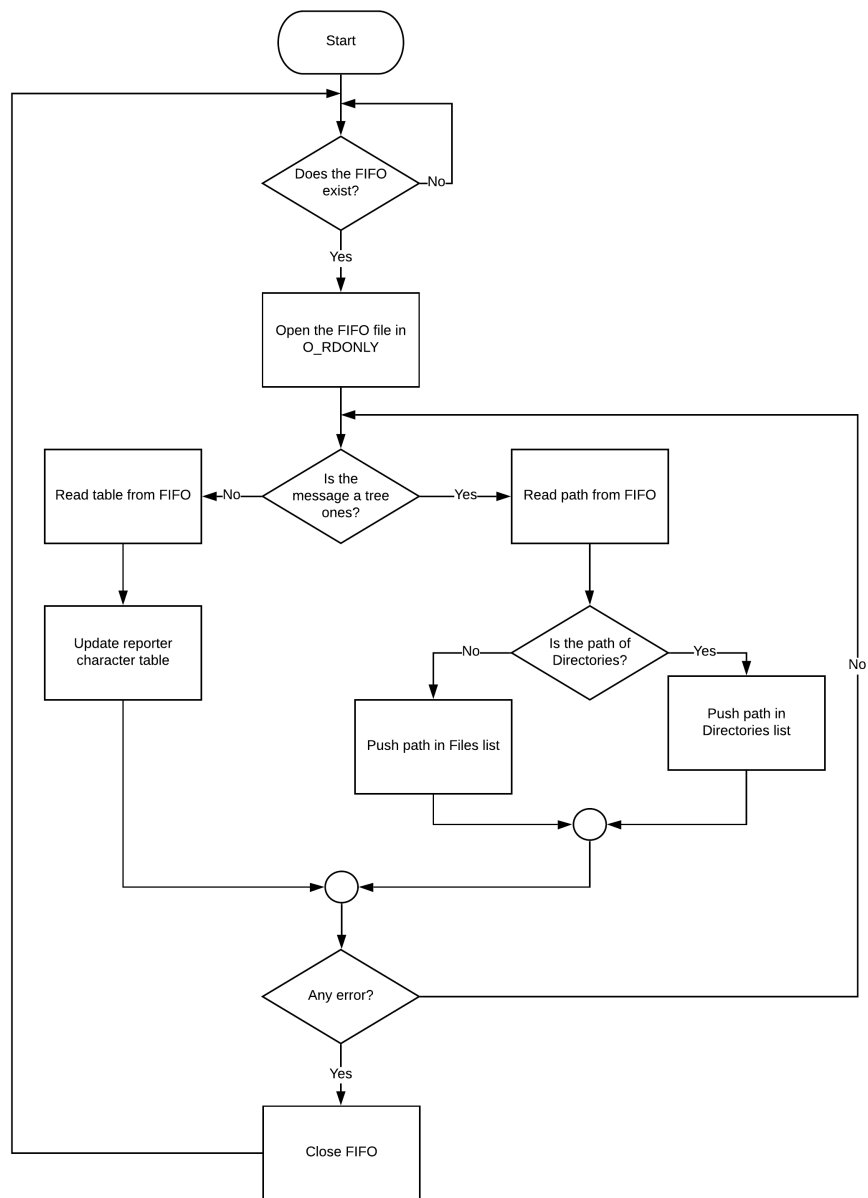


Figure 2: Read fifo thread

- send directives: the reporter sends new files or directories that the analyzer must compute, but also sends the number of managers or the new number of the workers if the user wants to change it
- send results: the reporter asks the analyzer to send back the results of the analysis of a list of typed/discovered files
- send tree (tui mode only): the reporter asks to receive the files and directories that are contained in the current working directory

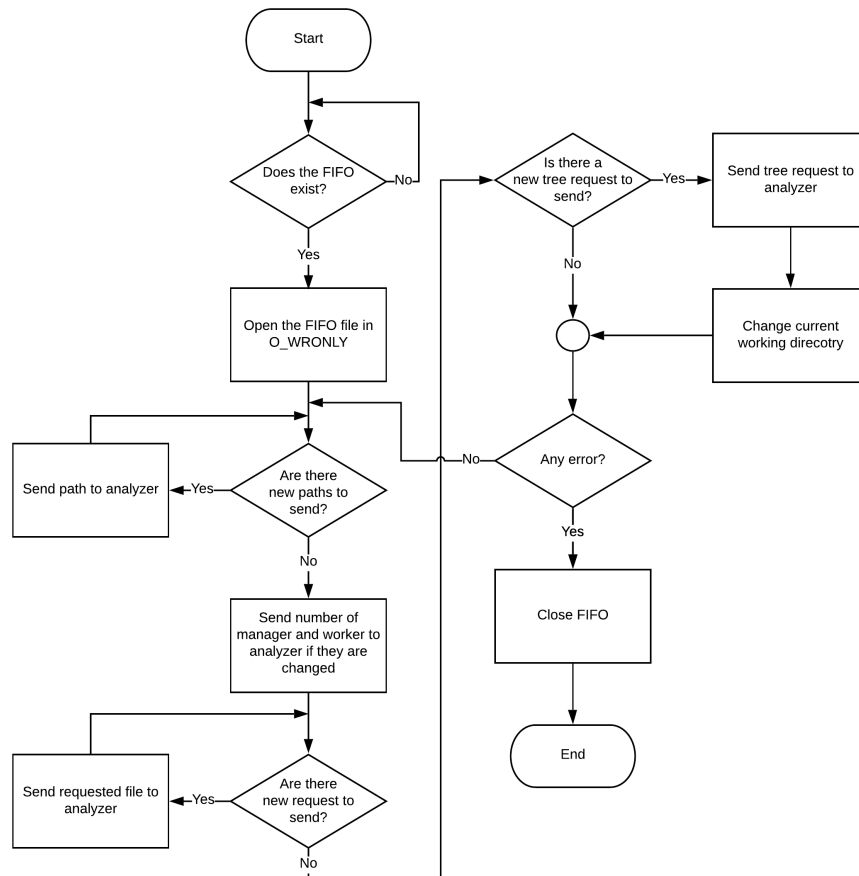


Figure 3: Write fifo thread

User input thread

This thread is only used when the program starts without tui. This thread allows user to insert command that are send later to analyzer:

- with “dire” the user can write new file or folder that the system must analyze and can change the amount of worker and/or manager
- with “requ” the user can insert a list of files to read the analysis about
- with “resu” the user can see the statistics that the system calculated during the execution
- with “quit” the user can close the program

Current working directory

The current working directory and the paths have different meaning based on the type of interface the user interacts with. With the basic mode the cwd is the directory where the reporter is executed (or where the main component is executed) and does not change during the execution, while with the tui the cwd is displayed in the dedicated block of the in interface and is changed when the user in tree mode types the name of a folder or .. (double dot).

Consideration

The reporter’s work seems to be not so important, but it is a central node because it is the bridge between the chain of processes that analyze the files and the user.

Tui

The terminal user interface is not a necessary component of the system but at the same time is the most used by the user. Our goal was to offer a good user experience despite the terminal environment. The first problem was about the low variety of libraries. We decided to use termios, a very low level library that allows to change the terminal in raw mode.

User interface

The user interface is divided in 7 specific blocks, each of them has different refresh level in order to improve the usability. A little brief:

- **User input:** user input block takes file or directory path from the user.
- **Quick help:** quick help block displays all possible commands for user input block.
- **Tree input:** tree input block allows to move in the file system.
- **Current directory:** current directory block displays the current directory.

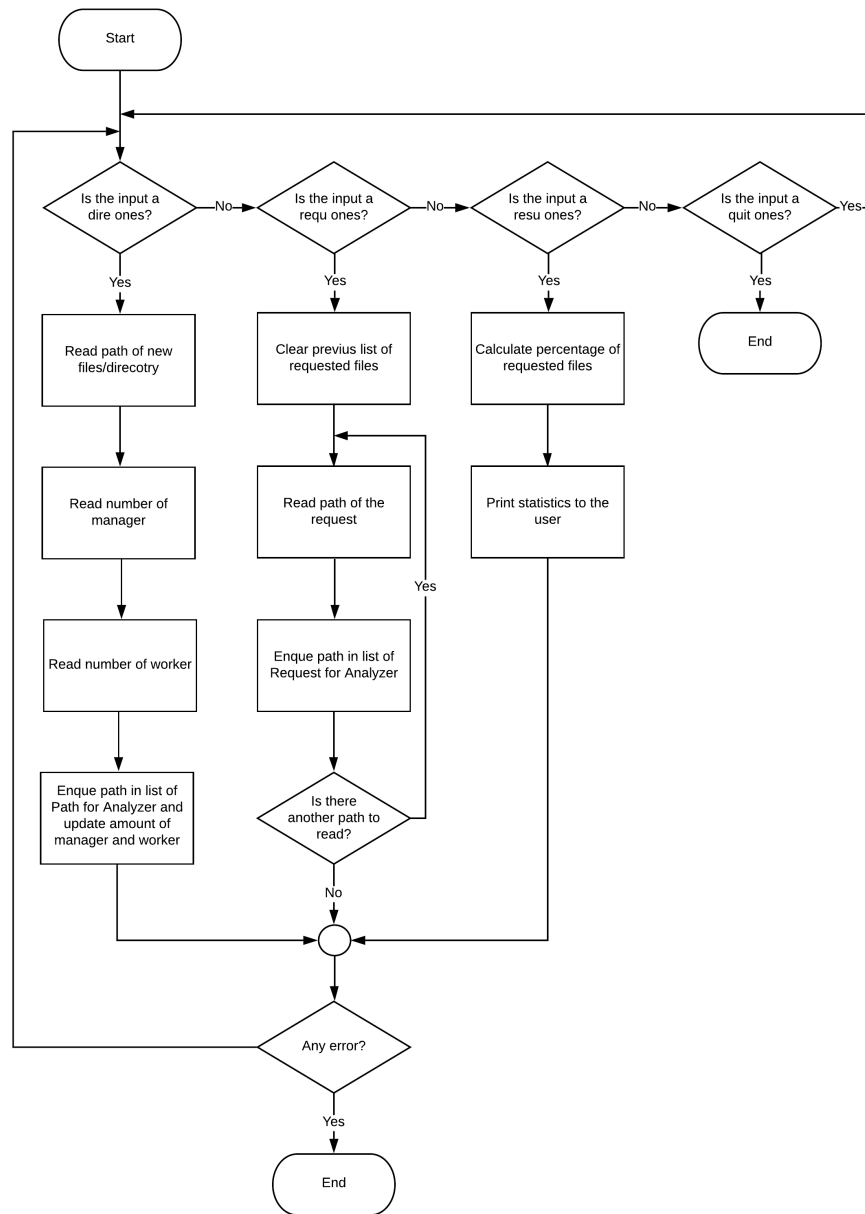


Figure 4: User input thread

- **File system:** file system block displays files and directories inserted in user input block of the current directory.
- **Percentages:** percentages block displays the percentages of some groups of char over the total.
- **Help message/Table presentation:** this block displays the help message or the table that contains the count

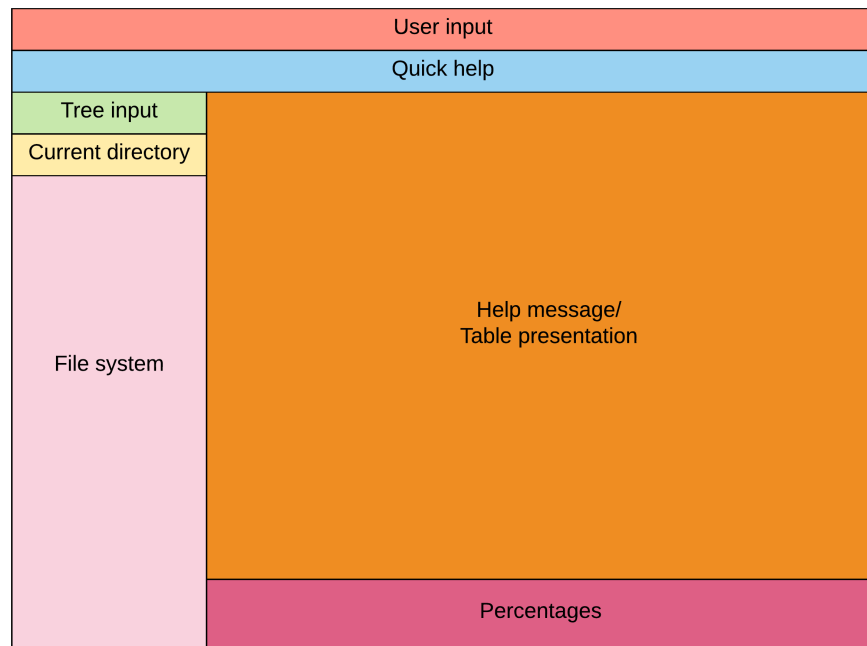


Figure 5: User interface

Implementation

In order to draw everything on the screen we take the current width and height of the terminal and we create a chars grid width*height. Then there is a thread that:

1. cleans the screen
2. prints all the grid
3. sleeps for a specific amount of time
4. repeat

A huge difference was made by the raw terminal mode and by the ansi escape code. Raw terminal mode allows us to take keyboard events and with them we could write what the user types directly on the grid. Ansi escape code instead

allows us to clear the screen, move the cursor and print some char in different colors.

All TUI is focused on the concept of moving inside the file system and toggling files. When a file is toggled/untoggled a request is sent to the analyzer through reporter in order to get the table that represents the analysis of all the toggled files.

User input

User input block is populated after a key is pressed. When line feed is pressed the text is parsed. If the processed text is an input it is passed to reporter that sends the input to the analyzer. Instead if the input is a command some changes are made. The commands are:

- **quit**: quits the application
- **help**: displays the help message
- **n**: changes the worker's amount
- **m**: changes the manager's amount
- **tree**: switches in tree mode and move cursor in tree input block
- **maiuscole**: highlights only upper case letters on the table
- **minuscole**: highlights only lower case letters on the table
- **punteg.**: highlights only punctuation letters on the table
- **cifre**: highlights only digits on the table
- **tutto**: highlights all ASCII letters on the table

The path written by the user can be relative or absolute and supports . (current directory reference) and .. (previous directory reference).

Tree input

Tree input block is populated after a key is pressed only if the tree command was typed on the user input. The tree mode area has four different types of input:

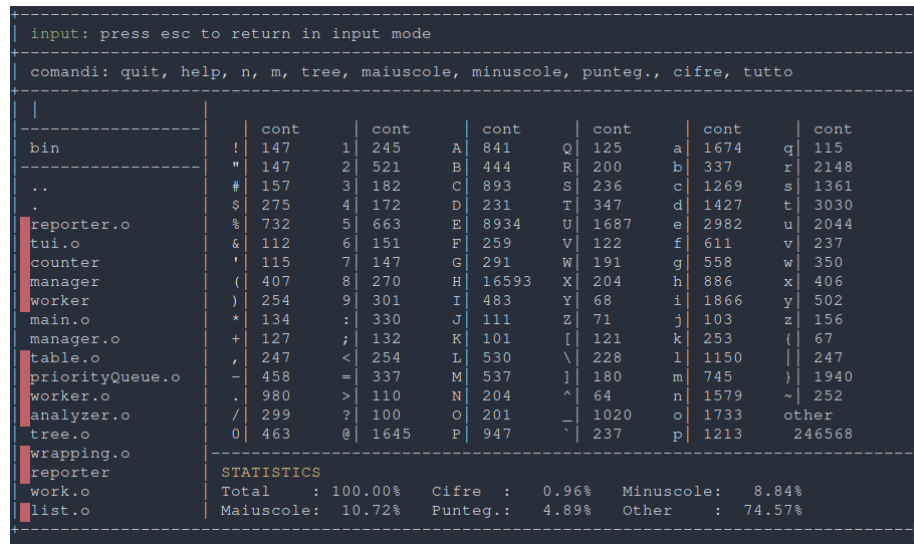
- file: when a file is written it becomes toggled if previously wasn't, otherwise becomes untoggled
- directory: when a directory is written the program change the working directory and file system block change according to them
- commands: there are two commands
- . dot toggles all untoggled files in the current directory (and vice versa)
- .. double dot moves backward in the file system
- down/up arrows: if the number of children of the current directory is greater than the lines of the file system block, with the down and up arrow the user can easily scroll between them

File system

The file system displayed in TUI's dedicated block shows only the files that the user passed as input to the system and are really stored on the disk. For this specific reason, the user can toggle only the analyzed files.

Considerations

Developing this component was very challenging and due the small amount of time given for this project and our inexperience with low level system call some parts of the code may be a little hard coded. Also our implementation may not be the most effective and the easiest to write but suits perfectly our needs.



The screenshot shows a TUI interface with a dark background and light-colored text. At the top, it says "input: press esc to return in input mode". Below that, it lists "comandi: quit, help, n, m, tree, maiuscole, minuscole, punteg., cifre, tutto". The main part of the interface is a table with columns for file names and their corresponding counts. The files are listed in a tree structure, with some files highlighted in red. At the bottom, there is a "STATISTICS" section showing the total count and the percentage of files in each category: Total (100.00%), Cifre (0.96%), Minuscole (8.84%), Maiuscole (10.72%), Punteg. (4.89%), and Other (74.57%).

	cont	cont	cont	cont	cont	cont	cont
bin	147	1	245	A	841	Q	125
..	147	2	521	B	444	R	200
.	157	3	182	C	893	S	236
.	275	4	172	D	231	T	347
reporter.o	732	5	663	E	8934	U	1687
tui.o	112	6	151	F	259	V	122
counter	115	7	147	G	291	W	191
manager	407	8	270	H	16593	X	204
worker	254	9	301	I	483	Y	68
main.o	134	:	330	J	111	Z	71
manager.o	127	;	132	K	101	[121
table.o	247	<	254	L	530	\	228
priorityQueue.o	458	=	337	M	537]	180
worker.o	980	>	110	N	204	^	64
analyzer.o	299	?	100	O	201	_	1020
tree.o	463	@	1645	P	947	`	237
wrapping.o							
reporter							
work.o							
list.o							

STATISTICS

Total : 100.00% Cifre : 0.96% Minuscole: 8.84%

Maiuscole: 10.72% Punteg.: 4.89% Other : 74.57%

Figure 6: Screenshot

Analyzer

The analyzer is the third element of the chain (starting from the bottom). The task of the analyzer is to keep track of the analyzed files with the help of a tree: if a node is a directory then it has a list of children. This component will get from standard input a file or a directory and, while inserting all the files in the tree, it will assign them to managers in a fair way. The analyzer will also handle any changes in the number of the managers by spawning or killing some of them. The same goes for the workers amount which will be changed by sending this particular information to the managers. If needed the analyzer will send the

list of children of a folder or, for the requested files, the sum of all the tables associated to the file, group by each character.

Structure

The structure of the analyzer is basically composed of five threads:

- readDirectivesLoop that reads directives from standard input
- fileManageLoop that handle the files and directories obtained from the directives
- sendFileLoop that communicate with managers
- readFromFIFOLoop that reads the reporter's directives
- writeOnFIFOLoop that accomplishes reporter requests

Tree

All the infos about files and directories are stored in a tree; each node except the root node can have one parent and multiple children saved in a list. Root doesn't have a parent but has children, it also has the destructor that will be executed on each node when the function destroyTree is called. The basic TreeNode contains:

- data
- children
- parent

The data stored in the TreeNode is a FileInfo instance that consists of:

- name
- fileTable, table that contains all the occurrences of the characters found in a specific file
- path
- isDirectory, flag that tells if the current node is a file (value 0) or a directory (value 1)
- isRequested, flag that tells if the current node is requested (value 0) or if it is not (value -1)

File/Directory insertion

When the analyzer receives a file or a directory to process it stores all the information in order to maintain a sort of file system hierarchy. When a path is obtained the analyzer will firstly compact it by removing references to the same folder and then process the node where the insertion will begin, starting from root of the tree if the path is absolute or the current folder if the path is relative, and by going up the chain of fathers or the chain of children. Basically the insertion function will first of all check if the starting node has any children. If the List of children is empty it will attach the new node to the current one;

otherwise, each child of the current node will be scanned and, if a current portion of the path of the file to insert is contained in a node (which is a directory) the function will change the current node with the matching one and will continue to analyze the children of that one; this process will end when the correct location of the file is found (i.e. the least common ancestor or the file itself if was already inserted).

Read Directives Thread

In the Read Directives thread all inputs from the stdin are analyzed and, if they match a specific pattern, (path, number of managers, number of workers) a list of operations will be performed in this order:

- if the number of managers changed then a certain amount of process will be spawned or killed based on the value stored in a shared variable
- the given path is checked in order to verify if it's a directory or a file: if that's the case the path is enqueued in the candidateNode List, otherwise the thread will wait for the next input

The special path “///” is used if the user would like to only change the number of workers of each managers without adding new files to analyze.

Manage File Thread

If a file was inserted in the candidateNode List it is then extracted in this particular thread (but only if the readFlag is different from SUCCESS to avoid possible overwrites). The node extracted is then inserted directly into the tree and into the fileToAssign List if it's a file, otherwise it'll be spawned a find process, using a bash call, in order to check all the directories and their files nested in the one passed as input. This operation is executed recursively for each of the directories founded in this process. In order not to block the operations in the other thread only a file at time is read from the spawned process and, as said before, it won't be possible to extract another node while this operation is still running in order to avoid any kinds of overwrite. The same insertion process for a file is performed in the same way for each result returned from the find.

Send File Thread

In this thread two different operations are performed:

- In the first half each managers is extracted and its pipe is checked: if there's something to read then the path of a file is read from the pipe. The path obtained is checked with all of the files in the filesInExecution List of the manager and in case of a match a flag is setted. After this operation 129 numbers are read from the pipe and if the file was found in the previous

operation the file's table is updated. If the file is also requested, before updating the amount of a specific character the actual value is subtracted from the requestedFilesTable and then the same table is updated with the new amount. At last the control word sent from the manager is checked: if it's equal to "done" then the file is removed from the filesInExecution List of the manager (if the file is requested then the sendChanges flag will be set), if the control word is "undo" then no operation will be performed (if the file is requested the same operation will be performed as described above), otherwise the user will be informed of a communication error between manager and analyzer.

- In the second half if the filesToAssign List isn't empty then the manager with less files is extracted from the priorityQueue. After that the file is first sent and then it's inserted in the manager's fileInExecution List. If the manager's pipe is full (or if the informations' size is greater than the remaining space on the pipe) then the file won't be sent, it won't be extracted from the fileToAssign List and the thread won't be stucked on the write.

Read FIFO Thread

After checking if the "reporterToAnalyzer" file exists a FIFO is opened between the analyzer and the reporter in O_RDONLY mode in order to be able to read the operations sent from the reporter. There are three main operations that can be performed:

- with "dire" the path, the number of managers and the number of workers are popped from the List of read words and then an operation which is identical to the one described in the Read Directives Thread will be executed.
- with "/" the element in the requestedFiles List and the requestedFilesTable are reseted and, while there are still element in the read word List, the files with a matching path to the one popped are setted as requested and their table is summed with the requestedFilesTable. After this operation the flag sendChanges is setted.
- with "tree" the path to retrieve is popped from the read word List and it is then saved inside toRetrieve variable.

If the word read is different from the previous ones and also different from "" (empty string) then the word is enqueued into the read words List ontherwise the FIFO is closed.

Write FIFO Thread

In this thread the informations requested from the reporter are sent. First thing first if the file "analyzerToReporter" doesn't exists then it's created. After this

operation a FIFO in O_WRONLY mode is opened. In the first half of the loop the toRetrieve variable is checked and if it's different from NULL then a file with a matching path to the one saved in the variable is searched inside the tree and the children of the retrieved node are sent to the reporter using the format "tree" "number of children" "fileName1" "fileName2" ... In the second half of the loop the sendChanges variable is checked and if it's setted then the word "tabl" followed by the requestedFilesTable (which means 129 numbers) are sent to the reporter.

Considerations

The analyzer and his working chain (managers and workers) performs wells, the threads that compose it allows it to both answer at user input and process the given files at the same time.

Manager

The manager is the second element of the chain (starting from the bottom). The task of a manager is to get from standard input a file, split it into multiple works and assigns them to workers. The directives through user or other component can be communicated with a manager in this specific form:

- a string that indicates the path file name
- a number that indicates the number of worker that the manager must handle

Structure

The structure of a manger is basically composed by two threads. One reads from standard input for new directives, the other communicates with workers.

Directives thread

The task of this thread is very simple:

1. read from standard input the path
2. read from standard input the number of workers
3. post processing the input
4. enqueue new directives in a pending list

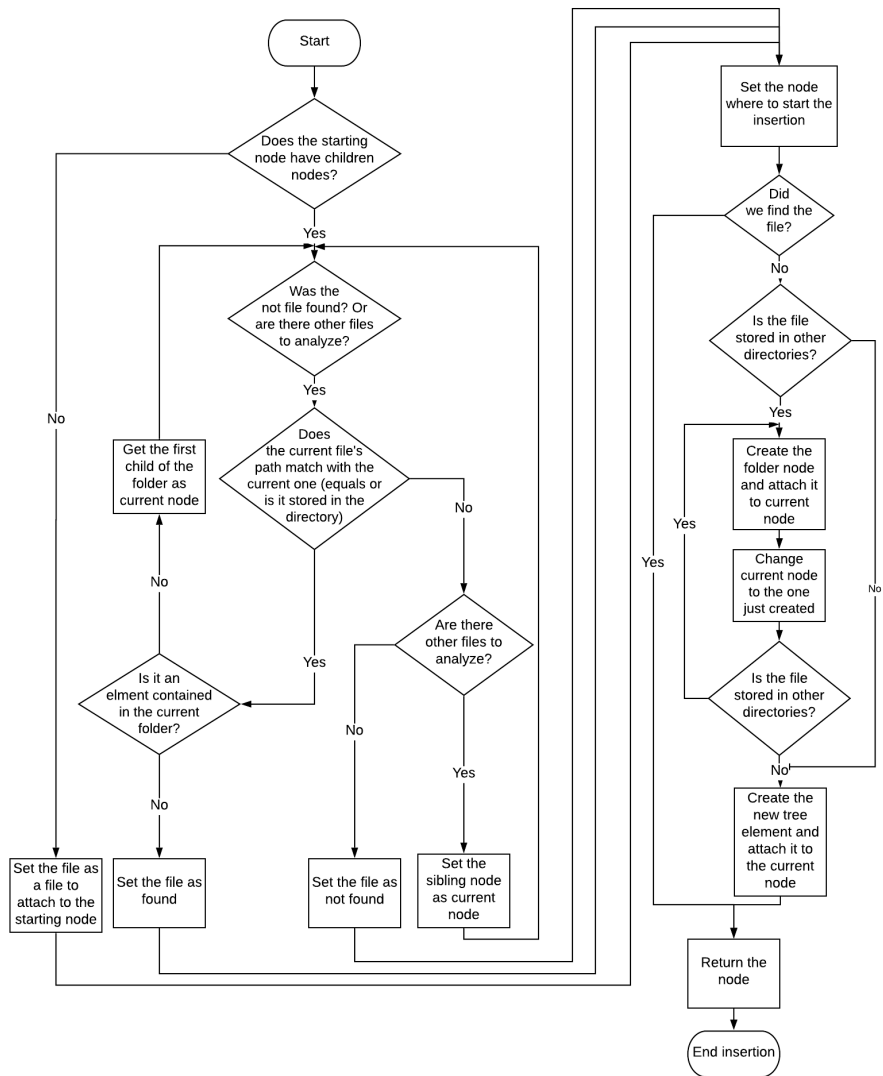


Figure 7: Node insertion

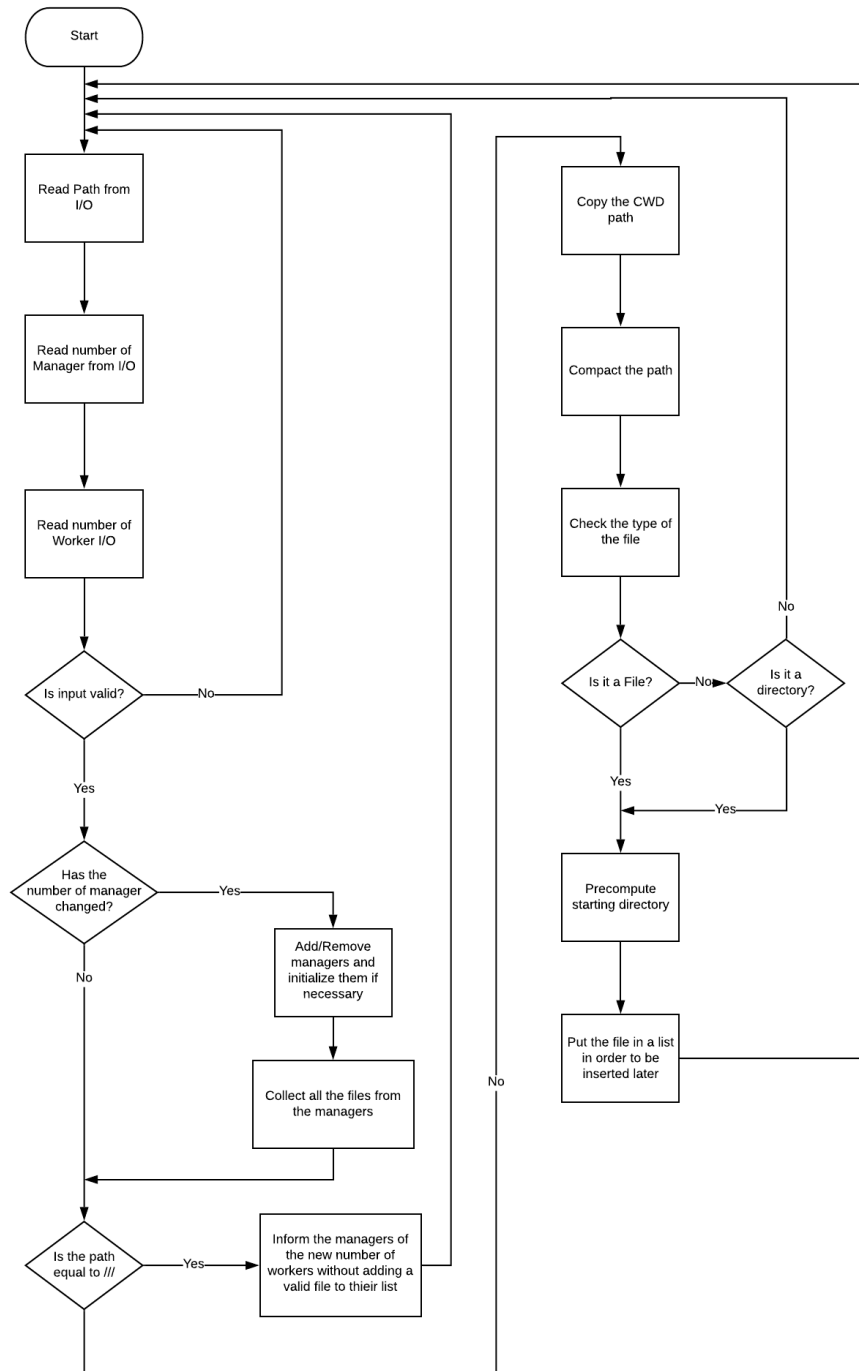


Figure 8: Read directives thread

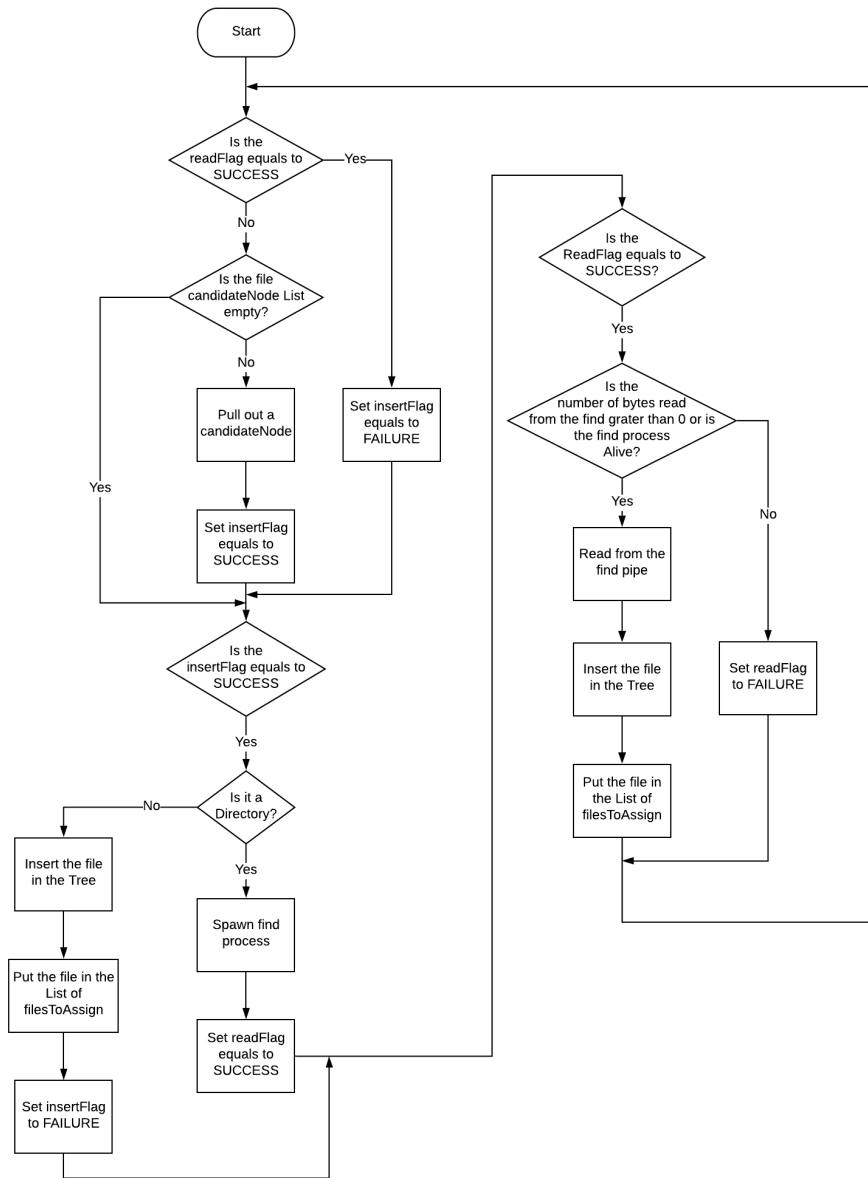


Figure 9: Manage file thread

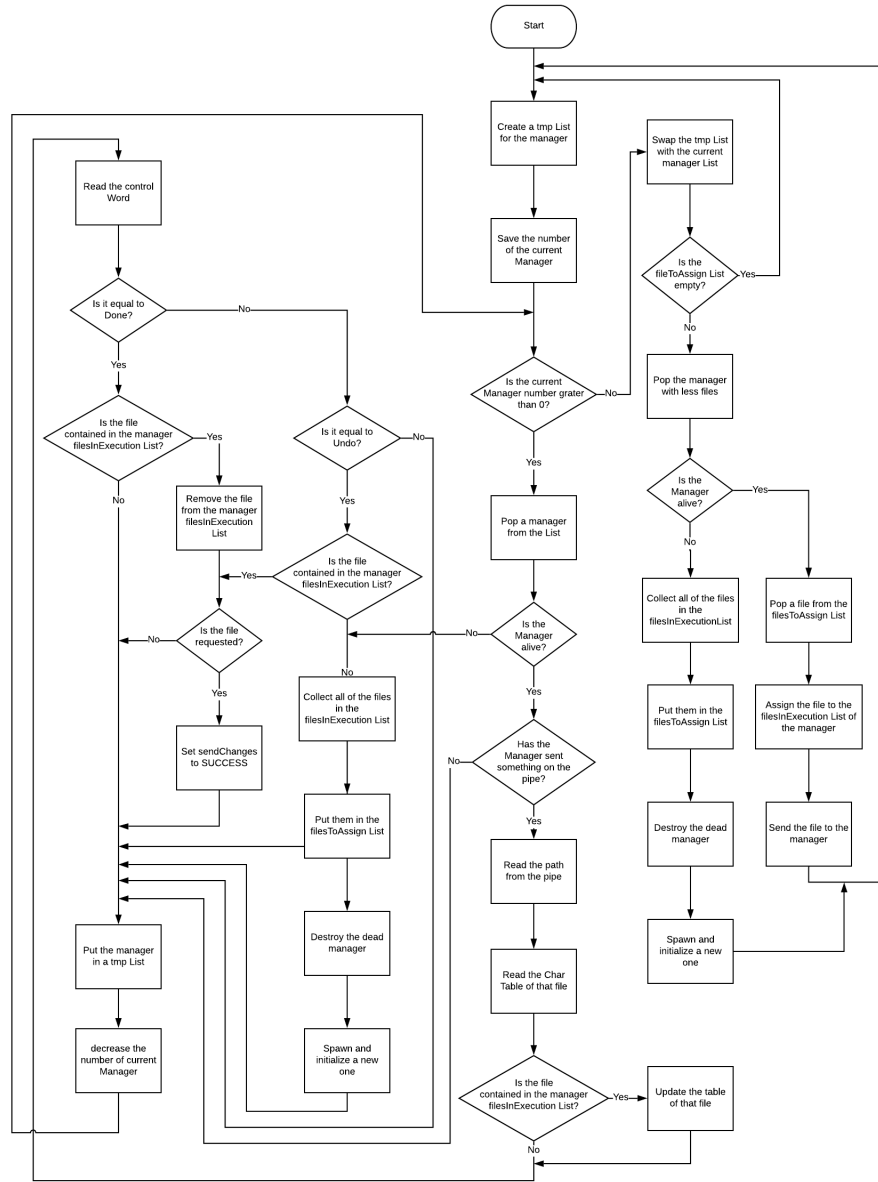


Figure 10: Send file thread

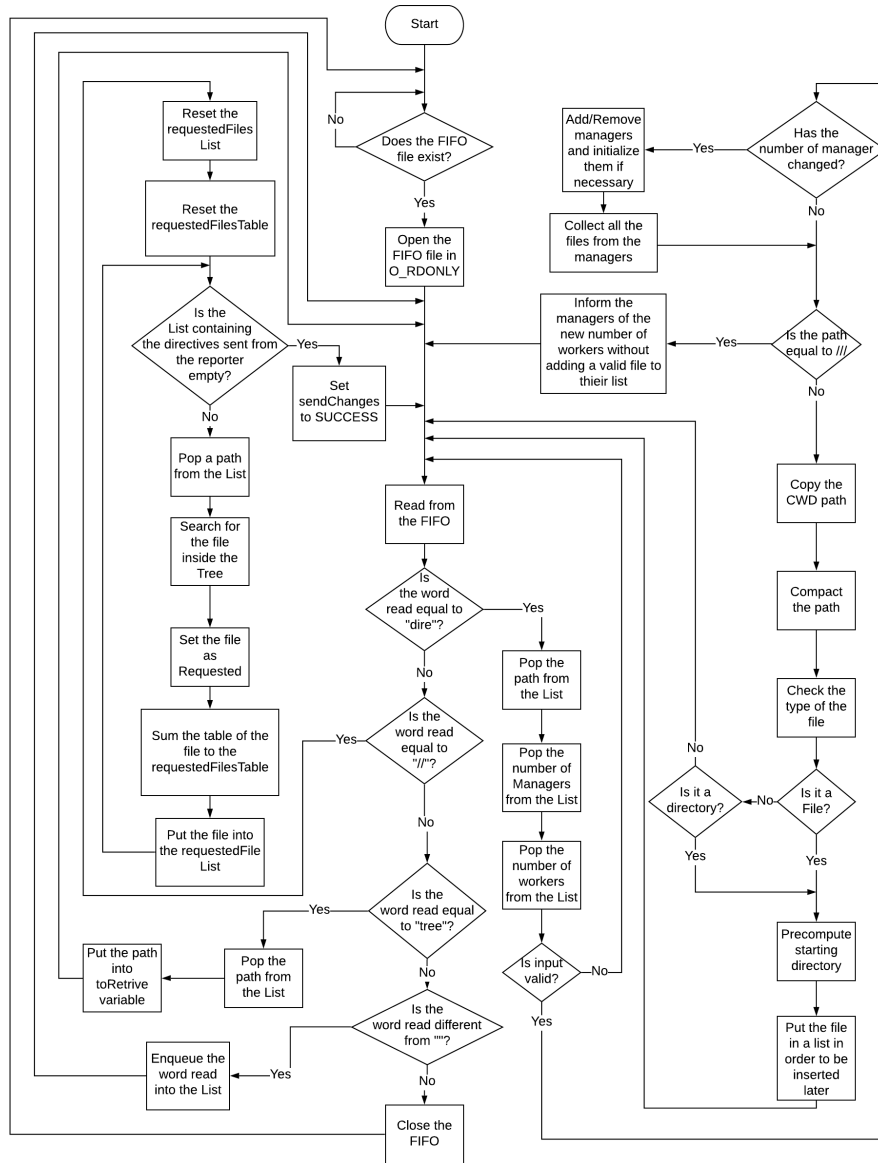


Figure 11: Read FIFO thread

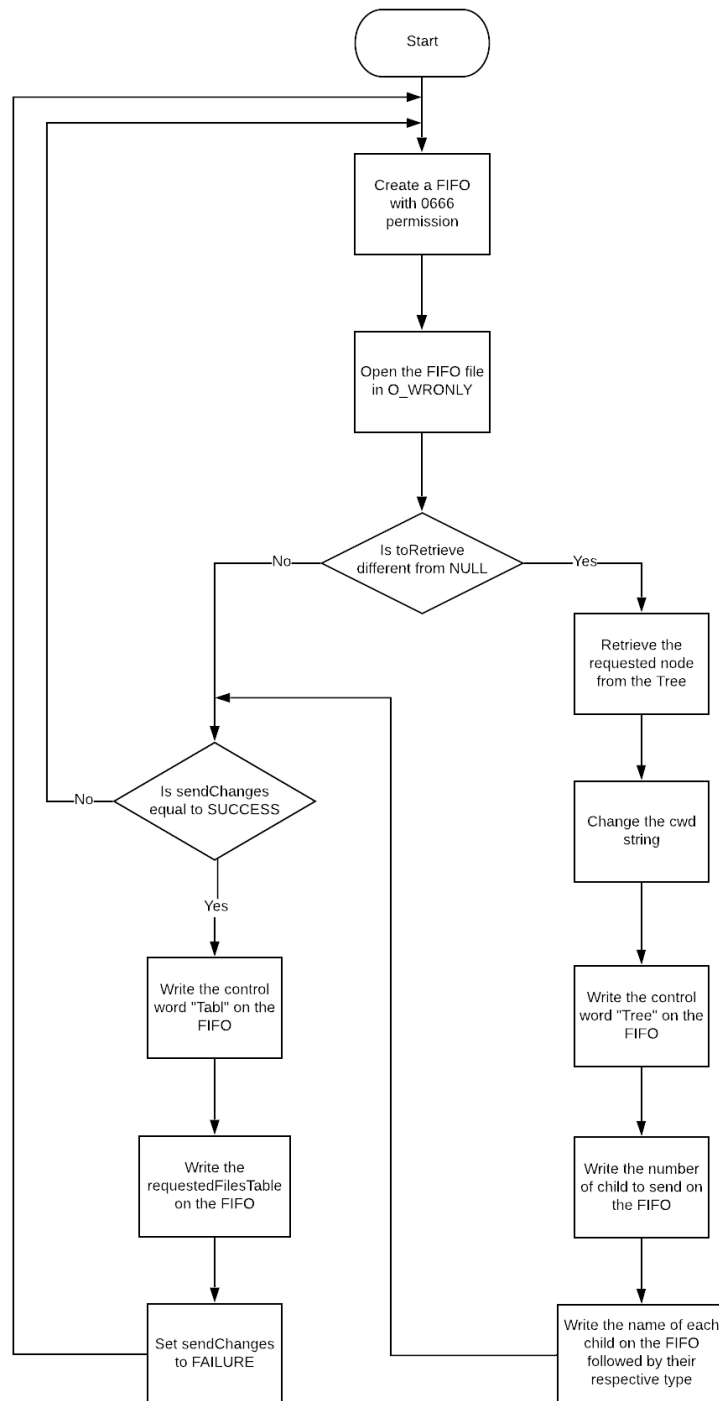


Figure 12: Write FIFO thread

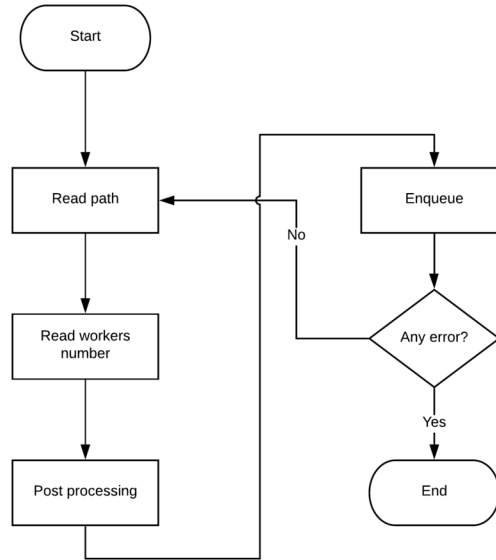


Figure 13: Directives thread

Work thread

The task of this thread is composed by some sequentially steps:

1. check if new directives were added
2. assign work to a pending worker
3. read worker's work
4. send to standard output a summary

New directives

If a new directive was added the size of the file is analyzed, then works equal to current number of workers are created splitting the total size of the file.

If the worker amount changes during the execution, all doing works are invalidated and are splitted again in multiples works. This operation is made for efficiency and parallelism reasons.

Read worker's work

Workers' works are read once for cycle. The manger try to read all worker's work amount for efficiency reason. After all work is sent the worker send a control

word in order to notify if everything is ok. If done is received, manager marks as ended the worker's work, otherwise worker's work is moved in to do list.

Spawn process

By default a manger spawns four worker. The spawn process consists in:

1. create two pipes (one for read, another for write)
2. set pipes as non blocking
3. fork a child
4. override child's standard input and output
5. change the child's code calling worker binary

Is alive process

After spawn process, manager saves the process ID of the spawned one. This PID is used to check if the worker, for any reason, is dead. If so, the worker's work is moved in to do list and new worker is spawned.

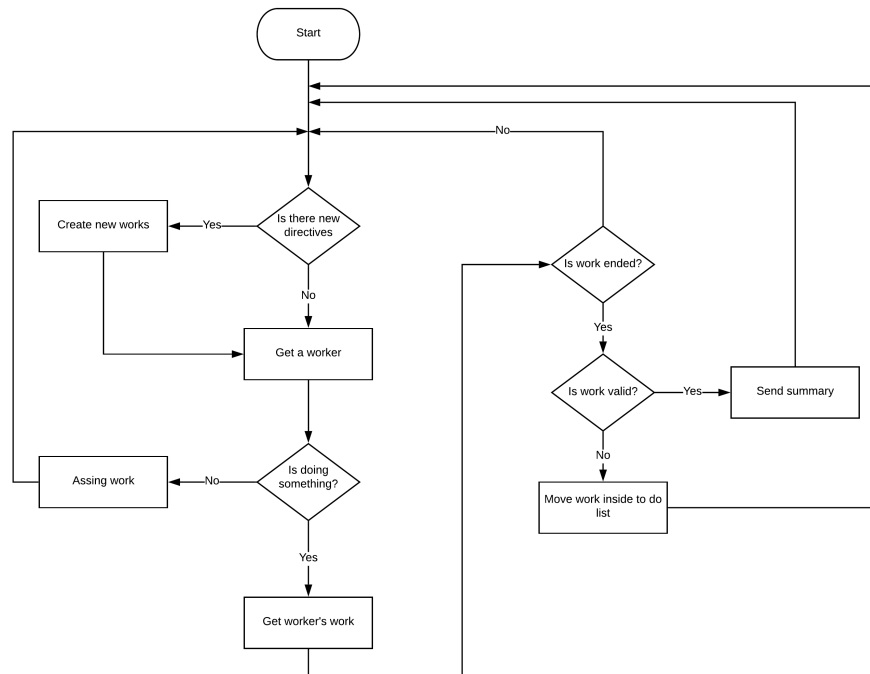


Figure 14: Directives thread

Considerations

In very stressful situations manager seems very slow and greedy of resources. The problem is caused by the limit size of the pipe. As a matter of fact the code is very good and fast. If you want to unlock the true power edit the file that contains the limit size of the pipe.

Worker

The worker is the lowest component of all the hierarchy. The task of a worker is to read a portion of the file very quickly. The following directives are read from standard input:

- a string that indicates the path of the file
- a number that indicates the offset where the worker must start its work
- a number that indicates the offset where the worker must end its work

Start and end points are included in the read operation. After the reading operation from the standard input, the worker casts the two numbers and returns an error in case of failure.

Before the whole process starts, the worker checks if there is enough memory for allocation. If so he allocates all work amount. If the available memory is smaller than the work amount, the worker tries to allocate the 50% of the available memory.

When the work is finished the worker tries to read new directives from the standard input.

Considerations

The worker can read huge file (we test it with a single file of 15 GB). This is the main reason for using unsigned long long type. The read operation is split if the work amount assigned to the worker is greater than 1.0 GB. We made this choice because the kernel doesn't allow big read operation in one time.

The atomicity of the write operation is 4096 Bytes but in this case is not a problem because other components wait for it.

The worker performance are very impressive. The 15 GB file was read in about 30 seconds (intel i7 7th gen, 40 Gb ram and ssd), the main problem is the usage of the pipe for the communication because their limit size is about 65000 bytes (depends on the OS).

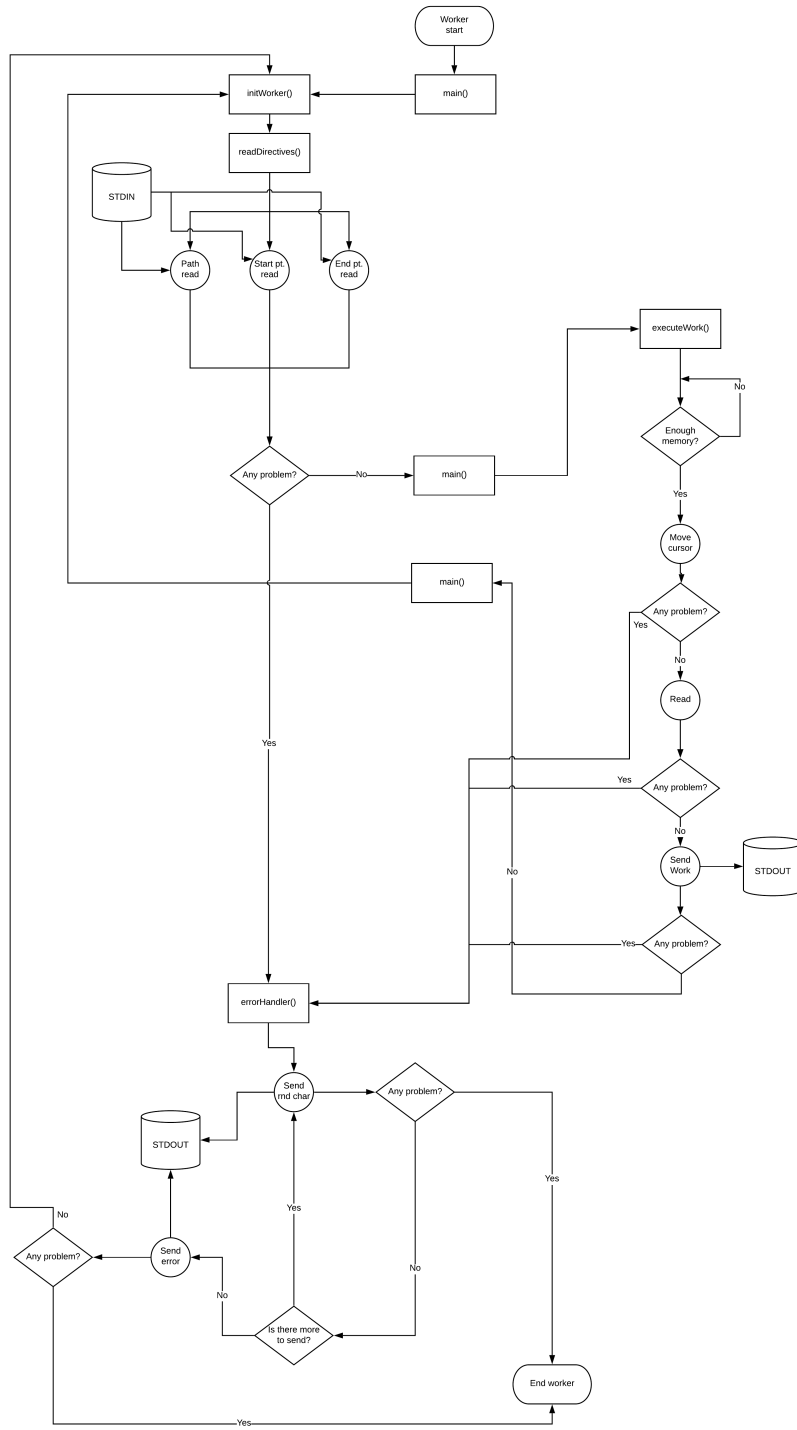


Figure 15: Worker schema
26