MyFDViewer

Reading through the assignment description, I decided to split the task into 3 modules:

- `ProcessStruct`
  - defines the linked list structure to store the information of each processes and corresponding FD information. The utility such as insertion and created node are also defined here.
  - define the CDT to store all command-line arguments and the function to initialize it from `argc` and `argv`.
- `TableDisplay`
  - `assemble` function to be called from main function to assemble and display information as required by arguments
  - many sub-function to be called by `assemble` to deal with different scenario, e.g. `assembleCompositeTable` for the argument `--composite`
- `MyFDViewer`
  - the main module, has the functionality to retrieve process/fd info from `/proc`, store it in list and display in proper format.

One main task is to convert those information and fit them in my data structure, method used for each field is shown below:

field	method
PID	open `\proc\[pid]\status` and check for the attribute "Pid:"
FD	use `opendir("\proc\[pid]\fp")`, for each fd, use `fir->d_name` to get name
fileName	use `readlink("\proc\[pid]\fp\[FP]")` to get file name for symbolic link
inode	use `stat("\proc\[pid]\fp\[FP]", &statbuf);` and statbuf.st_ino

some unnecessary filed are created and retrieved for the sake of easier debugging, e.g. `filename` for a process

Usage

use `make` to compile the program `MyFDViewer`

usage: ./MyFDViewer [flags]

If multiple flag indicating table display are specified, all the tables indicated by the user will be displayed. The assignment says ONLY a table will be displayed but this does not make sense to be since user should not be expecting only one table if two table flag is used. If this behavior is really desired, we can easily achieve this by add a `disableAllTableFlag(args)` function after each call of table assemble function.

Also, I wanted to modify the flag name to make them follow the same convention (e.g. change them all to `--abc-def-gh`), but I did not since I believe keeping them this way will make TA's grading easier. Is this a convention that I don't know? Otherwise, I hope they can be in the same convention in the next assignment.

Flag	Description
`--per-process`	Display the process FD table
`--systemWide`	Display the system-wide FD table
`--Vnodes`	Display the Vnodes FD table
`--composite`	Display the composite table
`--threshold=X`	Flag processes with a number of FD assigned larger than X in the output. It lists the PID and number of assigned FDs, e.g. PID (FD)
`--output_TXT`	Save the "composite" table in text (ASCII) format into a file named `compositeTable.txt`
`--output_binary`	Save the "composite" table in binary format into a file named `compositeTable.bin`

Positional Argument:

- A single positional argument indicating a particular process id number (PID). If not specified, the program will attempt to process all the currently running processes for the user executing the program.

Default Behavior:

- If no argument is passed to the program, the program will display the composite table (same effect as using the `--composite` flag).

IO observation

statistics:

- using `time ./MyFDViewer --output_binary`
    - 1 |real 0m0.081s| user 0m0.010s | sys 0m0.046s
    - 2 |real 0m0.042s| user 0m0.018s | sys 0m0.022s
    - 3 |real 0m0.078s| user 0m0.012s | sys 0m0.036s
    - 4 |real 0m0.035s| user 0m0.011s | sys 0m0.023s
    - 5 |real 0m0.066s| user 0m0.015s | sys 0m0.040s
    - 6 |real 0m0.034s| user 0m0.016s | sys 0m0.016s
    - Mean |real 0m0.056s| user 0m0.015s| sys 0m0.031s
    - Std Dev |real 0m0.019s| user 0m0.003s| sys 0m0.010s
    - file size| 173.2kb
- using `time ./MyFDViewer --output_TXT`
    - 1 |real 0m0.081s| user 0m0.011s | sys 0m0.048s
    - 2 |real 0m0.078s| user 0m0.010s | sys 0m0.032s
    - 3 |real 0m0.081s| user 0m0.012s | sys 0m0.035s
    - 4 |real 0m0.034s| user 0m0.009s | sys 0m0.024s
    - 5 |real 0m0.040s| user 0m0.009s | sys 0m0.032s
    - 6 |real 0m0.034s| user 0m0.008s | sys 0m0.026s
    - Mean |real 0m0.058s| user 0m0.010s| sys 0m0.033s
    - Std Dev |real 0m0.021s| user 0m0.001s| sys 0m0.008s
    - file size| 207.8kb
- using `time ./MyFDViewer --output_binary 1444`
    - 1 |real 0m0.037s| user 0m0.012s | sys 0m0.026s
    - 2 |real 0m0.034s| user 0m0.008s | sys 0m0.027s
    - 3 |real 0m0.025s| user 0m0.011s | sys 0m0.015s
    - 4 |real 0m0.025s| user 0m0.010s | sys 0m0.016s
    - 5 |real 0m0.025s| user 0m0.008s | sys 0m0.017s
    - 6 |real 0m0.024s| user 0m0.016s | sys 0m0.009s
    - Mean |real 0m0.028s| user 0m0.011s| sys 0m0.018s
    - Std Dev |real 0m0.006s| user 0m0.002s| sys 0m0.006s
    - file size 4.3kb
- using `time ./MyFDViewer --output_TXT 1444`
    - 1 |real 0m0.036s| user 0m0.008s | sys 0m0.029s
    - 2 |real 0m0.033s| user 0m0.013s | sys 0m0.021s
    - 3 |real 0m0.029s| user 0m0.007s | sys 0m0.023s
    - 4 |real 0m0.036s| user 0m0.014s | sys 0m0.024s
    - 5 |real 0m0.027s| user 0m0.004s | sys 0m0.024s
    - 6 |real 0m0.027s| user 0m0.004s | sys 0m0.024s
    - Mean |real 0m0.031s| user 0m0.008s| sys 0m0.024s
    - Std Dev |real 0m0.003s| user 0m0.004s| sys 0m0.002s
    - file size 5.0kb

From the stats above, we can see that the binary file size is much larger than the TXT output file, which make sense since we know a integer usually take less size compare to store it as a string. For example, an integer like 12345 would take up five bytes (one for each digit). In a binary file, the same integer would only take up 4 bytes (assuming a 32-bit integer).

The execution time when using the --output_binary option is generally slightly less than when using the --output_TXT option, both with and without the additional parameter 1444 (i.e. both in term of big and small sample). This suggests that the binary output may be more efficient in terms of execution time. This follows what we said in class and make sense since writing binary skip the process of converting a integer to a character.

The system time is larger then user time across all test cases, this might suggest my program is IO-bounded since it spend a lot of time waiting for system-level I/O operations to complete.

One peculiar thing I found is the first execution of specific command always seems to take the longest time. For example, the first execution of `./MyFDViewer --output_TXT --composite --systemWide --Vnodes --per-process` takes 0.120 seconds, the rest of the call are around 0.050 and 0.080 seconds. Perhaps system is optimizing the cache , therefore, if I execute the same command a few time in a row, the running time would be reduced. Suppose this assumption is true, we can see that `./MyFDViewer --output_binary --composite --systemWide --Vnodes --per-process` takes 0.080 seconds for the first time, which is a lot faster tha txt output. This support my first observation. Also, executing the same command a few moment later will get close result, i.e. 0.080 second.

Function Overview

`ProcessStruct`

Function Name	Description	Input Parameters	Output
`_is_all_num`	Checks if all characters in a string are digits	`char* str` : A pointer to the string to be checked	`bool` : true if all characters are digits, false otherwise
`createFDNode`	Creates a new FDNode	`int FD` : The file descriptor number `int inode` : The inode number `char* filename` : The name of the file	`fdNode*` : A pointer to the new FDNode
`insertFDNode`	Inserts a new FDNode into a linked list of FDNodes	`fdNode* root` : The root of the FDNode linked list `fdNode* node` : The FDNode to be inserted	`fdNode*` : The root of the FDNode linked list

Function Name	Description	Input Parameters	Output
`creatProcessNode`	Creates a new ProcessInfoNode	`int PID` : The process ID `int Inode` : The inode number `fdNode* FD` : A pointer to the FDNode `char* filename` : The name of the file	`processInfoNode*` : A pointer to the new ProcessInfoNode
`insertProcessNode`	Inserts a new ProcessInfoNode into a linked list of ProcessInfoNodes	`processInfoNode* root` : The root of the ProcessInfoNode linked list `processInfoNode* node` : The ProcessInfoNode to be inserted	`processInfoNode*` : The root of the ProcessInfoNode linked list
`printProcessList`	Prints the information of each ProcessInfoNode in the linked list	`processInfoNode* root` : The root of the ProcessInfoNode linked list	N/A
`readArguments`	Reads and processes the arguments passed to the program	`int argc` : The count of arguments `char* argv[]` : The array of arguments `arguments* args` : A pointer to the arguments structure to be filled	`int` : Returns 0 if successful, -1 if an error occurs
`deleteProcessList`	Deletes a linked list of ProcessInfoNodes	`processInfoNode* head` : The head of the ProcessInfoNode linked list	N/A
`deleteFDList`	Deletes a linked list of FDNodes	`fdNode* head` : The head of the FDNode linked list	N/A

`TableDisplay`

Function Name	Description	Parameters	Return Value
`getFDNumber`	Counts the total number of file descriptor nodes in the linked list starting from the node pointed to by `fd`.	`fd` : Pointer to the first node of the file descriptor linked list.	Total number of nodes in the linked list.
`assembleHead`	Assembles and prints the header of the table based on the arguments provided.	`args` : Pointer to an `arguments` structure that contains the flags for the different types of tables. `binaryFP` : Pointer to a binary file where the header will be written. `txtFP` : Pointer to a text file where the header will be written.	None.
`assemblePerProcessTable`	Assembles a per-process table and prints it to the console and the provided files.	`head` : Pointer to the head of the process information linked list. `binaryFP` : Pointer to a binary file where the table will be written. `txtFP` : Pointer to a text file where the table will be written.	None.
`assembleSystemWideTable`	Assembles a system-wide table and prints it to the console and the provided files.	`head` : Pointer to the head of the process information linked list. `binaryFP` : Pointer to a binary file where the table will be written. `txtFP` : Pointer to a text file where the table will be written.	None.
`assembleVnodesTable`	Assembles a vnode table and prints it to the console and the provided files.	`head` : Pointer to the head of the process information linked list. `binaryFP` : Pointer to a binary file where the table will be written. `txtFP` : Pointer to a text file where the table will be written.	None.
`assembleCompositeTable`	Assembles a composite table and prints it to the console and the provided files.	`head` : Pointer to the head of the process information linked list. `binaryFP` : Pointer to a binary file where the table will be written. `txtFP` : Pointer to a text file where the table will be written.	None.
`printThreshold`	Prints the processes that have a number of file descriptors greater than the provided threshold.	`head` : Pointer to the head of the process information linked list. `threshold` : Integer threshold for the number of file descriptors. `binaryFP` : Pointer to a binary file where the offending processes will be written. `txtFP` : Pointer to a text file where the offending processes will be written.	None.
	Assembles and prints the desired table based	`args` : Pointer to an `arguments` structure that contains the flags for	0 upon

Function Name	Description	Parameters	Return Value
`assemble`	on the flags set in the `arguments` structure.	the different types of tables. `root`: Pointer to the root of the process information linked list.	successful execution.

`MyFDViewer`

Function Name	Description	Parameters	Return Value
`isProcessOwner`	Determines if the current user owns the process given by `path`.	`path`: A string representing the path of the process.	0 if the current user owns the process, -1 if not, -2 if there was an error getting the process info.
`retrieveProcessInfo`	Retrieves information of a process given by the file descriptor at `dir`, and stores it in a linked list rooted at `root`.	`root`: A pointer to the root of the process information linked list. `dir`: A pointer to a dirent structure representing the directory entry of the process.	A pointer to the root of the updated process information linked list.
`initFDList`	Opens the `/proc` directory and initializes the ProcessInfoNode linked list.	`args`: A pointer to an `arguments` structure that contains the flags for the different types of tables. `root`: A pointer to a pointer to the root of the process information linked list.	The length of the process information linked list.
`main`	The main function of the program. It initializes the process information linked list, reads the command line arguments, assembles the desired table, and then deletes the process list.	`argc`: The number of arguments passed to the program. `argv`: An array of strings representing the arguments passed to the program.	0 upon successful execution, -1 if there was an error.