CSC3002 Course Project Report
2019-2020 Term 2

Group Members

李卓凡 117010146

李松浩 116020123

熊翰峰 117010317

杨牧原 117010342

王子仁 117010279

韦美含 117020290

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Content

I Introduction

We implemented a simulated Operation System which can show to its user how a real OS works like. Basically we have simulated three modules in the projects: Process Control Module, Memory Management Module, and File System Module. We have successfully realized both the representation and basic management of crucial parts in an Operation System like process, threadpool, memory allocation, memory release, as well as directory and file system. We have also integrated a Graphical User Interface to control and visualize our implementation.

II Related Work

Basically, we are inspired by the materials in the project description, several open source projects, and advices from senior computer science students who already finished the course Operation System.

Here are several projects that inspired us to form our ideas in this proposal:
https://www.cnblogs.com/findumars/p/7123266.html;
https://blog.csdn.net/sinat_34341162/article/details/83351651;
https://www.pcjs.org/disks/pcx86/windows/1.01/;
https://download.csdn.net/download/chaokudeztt/11182519.

Under the recommendation of senior computer science students, we found the course content and the textbook of Operation System extremely valuable and we will continue to learn from that, including Operating System Concepts (Silberschatz, A., Galvin, P. B., & Gagne, G., 2013, ninth edition) and Operating Systems Design and Implementation (Andrew, S., Tanenbaum., 2006, third edition).

III Our Work

1. Task Scheduling Module

The major part of the task scheduling includes the *Process* Class and *threadpool* Class.

The task class inheritances the *sysProcess* class and *userProcess* class, and these two classes inheritance the *Process* class. In the *Process* Class, we use it to express the program and create it when a program is open. The process class have two states Ready, Running and Terminated, the task belonging to the process will be created only if the state is Running. It will create a structure called job which include the content of the task and its size. The process will ask the memory for a specific ID for the size of the job and record it in the field called *memoryForJob*. After the job is created, we put it into the *threadpool* and let the *threadpool* to do it.

To be brief, creating a program that is suitable for our system to run needs the following three steps:
1, The functions shall be divided into unit *Job* form.
2, The task program itself shall be written as a child class of user process.
3, Override the *run* method under the task class as the main function of the original task program.
4, For running the exe file, the path of the *exe* file must be specified, for example, the *task2* need to change the *strPath* field to the path of the file.

In the *threadpool* Class, we achieve the goal to using Mutex to lock the resources and using priority queue to let the task with higher priority to be handled before the task with lower priority. The *threadpool* will be created as our operating system is open and be destroy as the operating system is closed. The *threadpool* have a field called *queue_cur_num* to record how many threads has been created in the *threadpool*, it will be increase when a new thread is created and decrease if a thread completes its task. The *thread_num* field is used to record the maximum threads allowed in the *threadpool* and *queue_max_num* field is used to record the maximum tasks allowed in the whole queue our *threadpool* will consider the three fields and decide how to deal with the coming task.

Our task scheduling also achieve the goal of parallel operating programs. The way we do that is to split every program into small parts. Every part is separated and is put into the *threadpool* waiting for running. It allows the computer to run different program alternately rather than finishing the whole operating of one program and starting to do the next.

2. Memory Management Module

In the proposal we planned to implement a paging memory strategy. However, we realized it could be too demanding for us to design such a complex system, and it is beyond of the needs to simulate and represent how a system works. After searching for other open source projects and learn from OS design, we decided to implement a mid-way but also very interesting first-fit memory allocation strategy.

Basically we separate the "memory stick" into different frames, and allow process to access memory by occupying frames. The first-fit memory allocation means that, after process send memory request (*requestMemory* method) with size, OS will find the first memory frame that satisfies the requirement; the OS will cut the memory frame into one that just fit the requirement and another piece that remain idle to be used. For example, if the first satisfied memory frame is of a size of 40KB and requested size is 30KB, the 40KB frame will be cut into a 30KB one to be used, and another 10KB one to remain temporarily idle in the memory.

The implementation of memory is composed with basically three parts: data structure, request, and release. We designed frame and table structure to store necessary information of memory frames; *findSpace* and *requestMemory* are desgined for process to request memory, *releaseMemory* and *mergeWithNext* are used for release frame(s) that is/are previously occupied by a process.

I want to highlight the *mergeWithNext* method we designed creatively to support the release of a memory. By cutting a satisfied memory frame into a just-fit one and another piece, we are basically enlarging the number of memory frames and cutting frames into pieces. If we simply release and recycle the frames that are occupied without any further operation, after many requests and releases our memory stick will be very messy and filled with many small pieces. So we designed a way to recover the cutting operation from previous request. Initiating the memory will record initial positions of frames, and cutting a frame into two pieces will not change the record of *ini_pos*. When releasing a frame, the *releaseMemory* method will check whether the adjacent frames are coming from the same initial frame; if so, *mergeWithNext* will be called to merge pieces into one. By doing this, after many rounds of requesting and releasing, the memory frames are the same as initialized.

In the User Interface we successfully implemented the user interface which is a memory monitor to show the virtual memory usage in real time. Qt provides an efficient constructing system for us to handle this job. We studied what is so-called “signals and slots mechanism”, how we use several widgets in Qt, and other topics from online materials. In addition, based on Qt, we also implement a calculator programming as one of our OS test sessions. Thanks to solid contents about Object-Oriented-programming that we have studied in class as well as supportive teach videos on BiliBili channels, I gradually dived in the front-end programming, step by step then finished the work.

3. File System Module

Design & Implementation

Initialiser of the File System --
Meihan Wei & Hanfeng Xiong

Related files: secdialog.h, secdialog.cpp, Filesys.cpp

This part is designed to initialise the file system, moreover, to recover the tree structure that was serialised after last run. The initialiser would check if there is an already exist root folder (as being the root of out tree), and if the text file for deserialization exists; if both conditions are meted, then deserialization would be executed. If both are not meted, then it would start a brand-new file system. Otherwise, it would give the error of “Invalid Path”. In addition, the text file for deserialization would be stored (if exists) in the real directory of the project.

Underlying Mechanism---
Hanfeng Xiong

(Macros are used to adapt to different system, so this part should be able to compile and work on win, MacOS, and Linux.)[1]

[1] Test file: Filesys.cpp; to perform the test, some files and folders need to be created and write, since there is some test about external sources (from line 68).

FCB (file control block):
Related files: FCB.h, FCB.cpp, AccessControl.h

All the files and directories (folders) are abstracted into instances belong to the two derived classes of an abstract class[2] FCB (file control block): class File and class Folder; an instance of these two classes would have following attributes:

- name (std::string): the name of this object.
- path (std::string): the real path of this object.
- type (std::string): the type of this object, and if it is a folder, then "folder".
- isFolder (bool): whether this object is a folder or not; basically, this is used to determine whether we are justified to do the dynamic cast (FCB* => File*, FCB* => Folder*), till then could we use some exclusive attributes of File or Folder.
- sz (int): the size of this object, if it is a folder, then $sz = \sum_{c \in child} c.size()$. This is a protected attribute which would update whenever this->size() is called; the reason it is designed as being protected rather than private is that the two derived classes are expected to inherit this attribute.
- nodeId (int): the id of this object as a node in the directory tree (we would talk about the tree later).
- location (int): whether is a temporary object from a mount point, and if is, of which one; if not, then set it as 0.
- parentNode (Folder*): the parent node of this object in the directory tree, i.e., the folder that it is located at.
- access (Access): a struct to record several aspects about access.
- getptr, putptr (std::streamoff): this are two file-exclusive attributes, which store the positions of get-pointer and put-pointer.
- child (vector<FCB*>): this is a folder-exclusive attribute, a vector contains pointers to all the child node of this folder.

These two classes also have methods which were integrated into the methods of the directory tree, to support some common operations; we would discuss the details in the appendix.

[2] Hence no instance of it, what we would like to utilise is its pointer FCB*.

Directory tree:
Related files: Directory.h, Directory.cpp

This is the main body of the file system, using FCBs to construct a tree which characterises the structure of directories. Take figure 3.1 as an example:

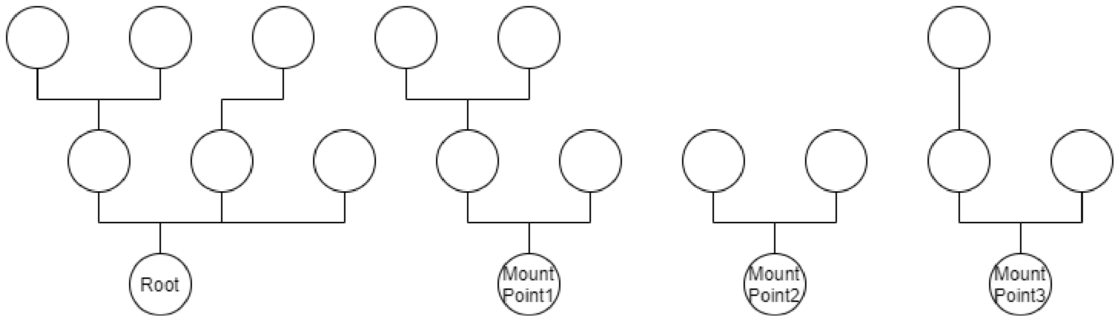


Figure 3.1 An example of Directory trees

Basically, there will be at least one tree, the root tree, and whenever a new directory is mounted successfully, a new tree should then be built, subsisting until umount is called. One can jump to a non-root tree using the method setMount, and jump back to the root tree just using the method cd, with input "/" or "/something/something", etc.

All the attributes of an instance Directory (tree) are private, since they are not expected to be changed without the use of corresponding methods. These private attributes include some crucial attributes such as a pointer to the root, a pointer to the working directory (as a Folder object), node-ID-pool, etc.

The methods of the Directory class are mostly designed as imitations of terminal commands; some important methods are introduced briefly as follows, for more details and other methods, please view the appendix.

Basic operations:
(All these methods are similar to corresponding Linux command.)

- **cd**: to change the working directory.
- **pwd**: to print working directory.
- **ls**: to list all child under the working directory.
- **Mkdir**: to create a directory under working directory.
- **Rmdir**: to remove a directory under working directory; unlike Linux, the target could be non-empty.
- **rm**: to remove a file; the target must not be a directory / folder.
- **mv**: to move a file or directory (and its child) in the current tree to the destination, the destination is allowed to be in a different tree (under one of the mounted points); also

support copying and renaming.

- **find**: to find matching files and folders.

Serialisation and deserialization:

- **serialise**: to serialise the directory tree and save it into a text file.
- **deserialise**: to recover the directory tree from the text file created by serialisation; by this two operation, we can shut down the system without loss of any information.

External-sources-related:

- **mount**: to mount an external source (directory) for temporary usage. A subdirectory of the root or of one of the mounted points will not be allowed to be mounted.
- **umount**: to remove a mounted source from reach.
- **lsMount**: to list all mounted sources.
- **setMount**: to move the working directory to a mounted point, i.e., the root of that tree (if you are confused about these concepts, please refer to figure 3.1).

File-related:

- **touch**: to create a empty file, open mode “std::ios::binary” is used here, so it does not have to be a text file.
- **cat**: to read and print the content of a text file (do not need to call open).
- **open**: to open a text file for potential operations, this operation would add its pointer to a vector, so there could be more than one open files, yet only one can be focused on at a time.
- **close**: to close the current-focused text file.
- **switchObj**: to switch the current focus onto another open text file.
- **lsOpen**: to list all open text files.
- **get**: to get a string of given size from current-focused file from the get-pointer.
- **put**: to write the given string into current-focused file from the put-pointer.
- **repos**: to reposition the get-pointer or the put-pointer; use this method to aid the previous two methods.
- **trunc**: to truncate the current-focused file.

Access-control-related:
(This module is semi-completed, since it is not so important and the limitation of time; all the material that would be needed are created, yet not linked to all the operations above)

- **newUser**: update the table of user-information.
- **renameUser**: update the table of user-information.
- **switchUser**: update the table of user-information.
- **deleteUser**: update the table of user-information.
- **setMode**: change the access mode of a file or a folder.

Among all these operations, we have several of them need to traverse the tree (or a subtree), such as Rmdir, mv, find, serialise, etc., and some other similar operations like deserialise and mount, of which different methods are employed to perform the jobs. For example, Rmdir, mv, serialise and mount use recursion, while the others use loop; Most of these are performed the same pattern, first reach to the nethermost of a branch, then another, etc., although difference exists: e.g., Rmdir must go straight to the end of one branch, then goes up and execute the unit operation, until a fork is encountered; while mv must go down one branch, execute the unit operation before moving to the next node, and when the bottom is reached, it then returns to the fork that was most recently encountered directly, and go down another sub-branch

Deserialisation is a different one: since path, equivalent to the node ID, is a super-key to the tree as a dataset, and we know that during the process of serialisation, a tree node of higher level must have been wrote to the file before its child, therefore, we can simply reconstruct the tree by reading the data file a line by another, and re-create the corresponding node; when the reading is finished, the tree would then be rebuilt.

4. Graphic User Interface

Graphic User Interface & General interaction---
Meihan Wei
This part implements the interface between user and program, which helps to make the complex bottom mechanism more friendly and directly to user. First, through user interface, user do not need to learn about the difficult command in terminal. For example, instead of inputting command like "cd..", "cd-", user can directly click button "back" and "history". Besides, instead of remembering all input parameters of method for directory, some button helps to give parameters automatically. For example, "dest" and "dest_loc" parameter in "mv" method will be automatically generated for user when they want to download and upload files. This relieves user from pain of remembering and learning parameters. What's more, some constrains I set in the buttons makes it very robust and less likely to make error from command.

Login Window
Meihan Wei
Related files: mainwindow.h, mainwindow.cpp, main.cpp
This part simulates login in process of real operating system. This interface is pretty robust. First, users are required to input username, password to prevent bad people from accessing our operating system. The username is "test" and the password is also "test" by default. If the user input incorrect password or username, there will be information showed at the bottom of login window. Then, in the empty "LineEdit", user should input a valid path to initialize file system. This directory is automatically linked to root directory of our simulated file system. If the path is not valid, or the path exists without "detail.txt", or the path does not exist but with "detal.txt", warning window will pop out to make user enter again. To learn more detailed algorism that this part is based, please refer to "initializer" part written by Hanfeng Xiong.

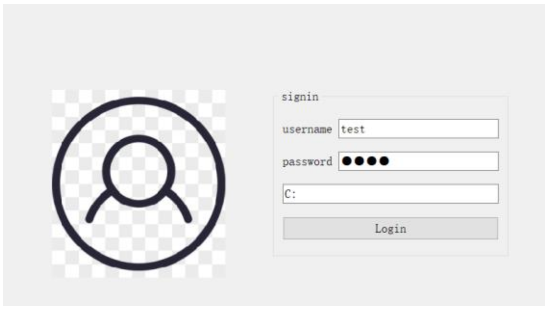
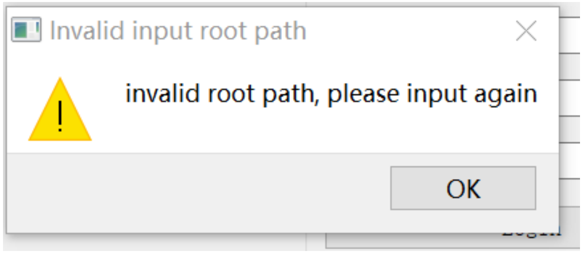


Figure 3.2 a possible input for login in process under windows system



invalid root path, please input again

Figure 3.3 warning about invalid input

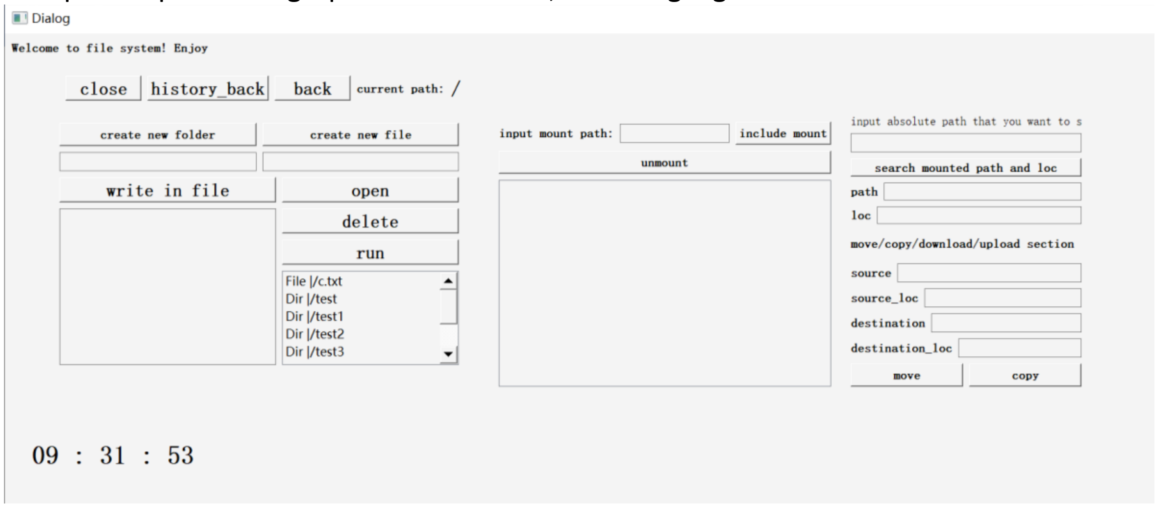
If the user run this program for the first time, a new directory will be created inside the path he input. If the user run this program again and **enter the same path as the first time**, what have changed before in this filesystem will be updated. Remember, do not delete the directory created before, otherwise the directory will be initialized from the beginning.

Window for Filesystem

Meihan Wei

Related files: secdialog.h, secdialog.cpp, secdialog.ui

This part implements graph user interface, including algorism behind.



Welcome to file system! Enjoy

File |/c.txt
Dir |/test
Dir |/test1
Dir |/test2
Dir |/test3

09 : 31 : 53

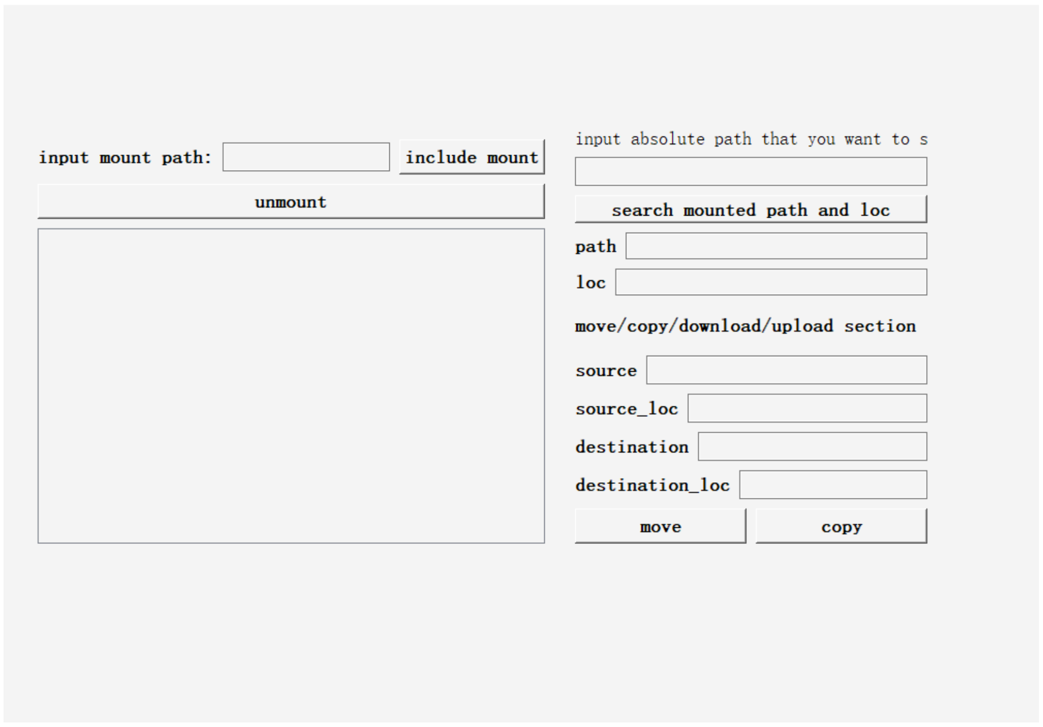
Figure 3.4 window for file system

Functions inside root directory:

Once the user login in successfully into operating system, this window will pop up and the current directory, which is presented on top of window, is root directory. The files and folders in current path are presented with help of “QDir” class in Qt. To link this interface with method in “Directory” class, each time user clicked an item in list box, and click certain action button, the slot function behind the button, will read the text on item selected, deal with the text string, find out name of folder and file as input of method in “Directory”.

- Open: If user select a file, the text inside file will be shown in text box on the left. If user select a folder and click open, filesystem will go inside that directory, “ListWidget” will update and show the files and folders inside that folder and current directory will change to path from root for that folder.
- Delete: This button linked to method “rm” and “rmdir” in “directory” class. This button is able to deal with the situation when user does not select anything in file/folders list and click this button. This is relied on “bool” type design of “rmdir” and “rm” in “Directory” class. When the method return false, a warning window will pop up to remind user about the error.
- Run: This button is used to execute executable files, related to integration of three main part of our simulated system, memory, thread, and filesystem. The executable files are prewritten, and this button is not able to execute any file that the user upload. This button can not deal with the situation when user did not select any file in file list.
- Write in file: This button is designed for editing a file. User can open a file, type in box on the left, and click this button. The string stream will flow into that file and save the file permanently.
- Create new folder: type in folder name, and click this button, a new folder will be created under current working directory. This button is robust to invalid folder name. Make sure that the name should not contain special symbol such as ‘/’ , ‘”’, ‘\’ , ‘|’ , ‘?’ , ‘*’ , ‘.’ .Otherwise, you will be asked to input again.
- Create new file: similar to create new folder without constraint of including “.” in filename.
- Back: This button is used to back to upper directory. If the directory is already root, warning will pop up.
- History back: This button is used to return to directory that the user has visited.

Functions related to mount
(Mount is directory that outside our simulated filesystem, which can be regarded as USB flash disk.)
This part is one of the highlight parts of our simulated filesystem. The advantages of this part include specific warning about input, which make it easier for user to correct their input. Second, more than one mount can be loaded. Validation of mount path will be checked. What’s more, this graph user interface is designed in the most concise and efficient way, implementing three complex functions into only two buttons. Button “move” and “copy” can perform function of downloading, moving file/folder from root directory to mount, copying from root to mount, at the same time perform function of uploading(move/copy) file/folder from mounts to root. Also, copy and move inside root directory can be performed with previous mentioned two buttons.



input mount path:
include mount
unmount
input absolute path that you want to s
search mounted path and loc
path
loc
move/copy/download/upload section
source
source_loc
destination
destination_loc
move
copy

Figure 3.5 window for mount part of filesystem

- Button “include mount”: Enter valid absolute path independent to root path into box after “input mount path” and push this button will include new mounts (insert new USB flash disk) to simulated operating system. Mounted paths will be present in “ListWidget” box.
- Button “umount”: If the user does not want to access resources under certain path, user can select that item in “ListWidget” box and click “unmount”. Particular mounted path will be removed.
- Input parameters for movement of files/folders
- Source : related path which follow writing style after linux terminal language after “cd”. If the source is inside root directory, input path from root. If the source is inside mounts, use “search mounted” button to generate this parameter, shown behind “path”.
- source_loc: integer input parameter specifies source location. If the source is inside root directory, this parameter should be 0. If the source is inside mounts, use “search mounted” button to generate this parameter, shown behind “loc”. Just copy it.
- Dest: Specify path for destination. Which must be a folder. Similar to source.
- Dest_loc: Specify location for destination. Which must be a folder. Similar to source.
- Search mounted path and location button: this is designed to make it more user friendly to input the complex parameter and move/copy files in different conditions. User just need to enter absolute path of resources in certain mounts, then, what the user should input into “move/copy” section will be presented to user directly. If the input path is not valid or is unmounted path, no result warning will pop up.

Integration ---
I also incharge of integrating three parts, file system, memory, and threads. The processes are initialized as directory initalised in secDialog. Executing function of executable files are also integrated inside with file system.

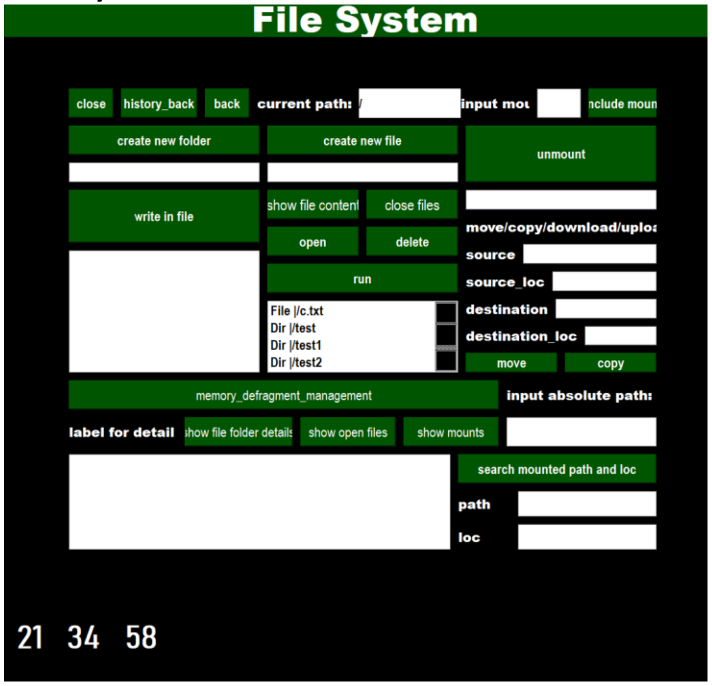


Figure 3.6 new graph user interface for filesystem

After adding memory control and defragment of memory in filesystem:
Maximum memory limitation is set to 1MB
Defragment is used to manage space
Major changes of code is shown in dev_log_6_2.doc
Changes in new GUI:

- Button: memory defragment management: after clicking this button, you will see message in detail box and more detailed information about fragment in Qt terminal. Method behind this button is "defragment()" in Directory class.
- To read content in file: open and read content function is split into two buttons. Button "open" is only used to open file button "show file content" is used to read the file to "ui->plainTextbox."
- Button close: close all files that is open. If no file is open, a warning will pop out.
- Button move and copy: When uploading files and folders into simulated OS, the size of source folder and directory will be checked in method "mv" of directory.
- Button "file/folder details": show detail information about all files and folders in text edit box. User can check size of certain file.
- Button "show open files": list all files that user has open in text edit box.
- Button "show mounts": show outside resources' absolute path that user has add to OS in text edit box.

Reflection from Meihan
There are still some drawbacks such as some input part is not so robust. For example, for buttons related to “listWidget” are not robust enough. Before clicking “open”, “run”, “write in file”, user must select particular items in “listWidget”, otherwise the program will give an unnormal stop error. Also, the coding style can be more concise.
What I am pride of myself most is that this is the first time I learn something brand new (QT GUI) by myself without guide of teachers.
Thanks a lot to my partner Hanfeng, who help me learn a lot outside text book, such as the concept of mount, the input style in terminal (“cd”). Especially, he helped me do debugging, which cultivate my logic thinking ability.

IV Contribution

Our group has six members and are split into three couples to handle three modules. Zhuofan Li and Muyuan Yang are responsible for Process Control Module; Songhao Li and Ziren Wang are responsible for Memory Management Module; Hanfeng Xiong and Meihan Wei are responsible for File System Module. We collectively think everyone has done a great job in-time; every teammate should be evaluated equally based on our collective result.

V Reflections

During designing the UI for memory management system, one useful lesson we have learned is: The connector between frond-end and back-end is crucial in programming design. Programmers may address their own works separately. But in the end, they make up all the components as the whole by connectors. One’s output becomes one’s input. We should firstly figure out how we link to each other before we do our own works in isolation.

In the Task Scheduling Module, we have met much more difficulties. The first difficulty for us is how to deliver the parameter of the function to the *threadpool* since parameters has various forms and the function can only be called along. To deal with this problem, we deliver the callback function and its parameter separately. For the parameters, we use a pointer to handle its various forms and change its form when we need to use it. The second difficulty for us is to put the function which belongs to a class into our *threadpool*. It is known that the callback function needs to be static function and the *function* belongs to a class is not easy to become static. To overcome this problem, when we need to put a function that belongs to a class into our *threadpool*, we construct a helper function that follow the required form and call the helper function in the given function and deliver a pointer to the class instance as the parameter for the helper function. Another problem is that the UI and some methods will cause some errors when its belonging stack is destroyed. To solve this problem, we take some of the instance as static instance and deliver its pointer to the method that need to use it.

We also meet the problem that when we combine process control with file system, the program is hard to operate. To solve this problem, we design the *systemProcess* and *userProcess* using inheritance of *Process* class. They have the form as the following figure. Then all of the program that we run in our operating system can be divided into the two child classes depend on whether they need to go through the *threadpool* to be processed.

Another hard problem is that when use the UI provided by Qt, it is hard to combine our work of *threadpool* with the UI. The reason we supposed is that Qt will automatically block other threads when doing its UI. Since we cannot find a way to solve this problem, our *threadpool* is not allowed for the exe file with UI. For those exe file with UI, we just give let them to be *systemProcess* and record the memory that it needs to use directly rather than going through our *threadpool*.

VI. Compile & Test Manual

In our project, we prepared three testing programs to demonstrate the usage and power of our operating system. They are named as Task1, Task2 and Task3. All three programs are established by our own.

Task1 is a multi-thread program, it consists of two subprocess and each thread does a simple job: print "step 1", sleep for a while, then print "step 2".
Task2 is a program that would run an execution file. It basically showed how does the project pack a real execution file into our task class and then run it. With it the user may try pack different exe file and let out operating system run it.
Task3 is a C++ program with an independent UI. It shows how does our operating system handle the program that has interaction with the user.

1. Compiling
    a) Environment:

    i. Operation System: Mac / Windows / Linux
    ii. Qt Creator Version: 4.11.0 (For lower version, Windows user may need to download the pthread package)
  b) Compiling
    i. Directly use Qt to build and run the project, or
    ii. Using minGW as compiler

2. Preparation
   a) The task2 will run an execution file which is sensible to the environment. In our program we prepared an assignment.exe that could run on Mac. (This execution file is a Mac version release of the first assignment.) If you are using Windows, you may release the same project.
   b) In task2.cpp line 14, change the file path to the absolute path of this assignment.exe.
   c) You may also use other execution file. However, this may run into problem and crash our Operating System project.

3. Start running
   a) In the file page. You will see three files named as Task1.cpp, Task2.exe and Task3. You can first select them and click run.
   b) The memory usage page will pop out and show the usage occupied by the process.
   c) For Task1.cpp, the multi-thread program will directly print the words in Qt's terminal. Since it consists of two subprocess. It will have two occupied memory.
   d) For Task2.exe, it will also print the words in Qt's terminal. However, it cannot interact with you, since our project does not manage the IO.
   e) For Task3, it will pop out a calculator UI. This program can interact with you and you may play with it.

4. Modification
   a) For Task1, you may change the number of subprocesses to change the output and the memory occupied.
   b) For Task2, you may use your own program. This may risk crash our operating system.

VI. Reference

Andrew, S., Tanenbaum. (2006). *Operating Systems Design and Implementation, Third Edition.*

Philipp Oppermann(2018), CPU Exceptions, Writing an OS in Rust. Retrived at:

https://os.phil-opp.com/cpu-exceptions/

Silberschatz, A., Galvin, P. B., & Gagne, G. (2013). *Operating System Concepts (ninth edition).*