**Pintos Project 2 : User Program (2)**

**(Project Report)**

Subject: :[CSE4070] Operating Systems

Faculty Member: Prof. Youngjae Kim

Group Number / Members: 53team, 20151562 신은우, 20151616 최승환

Development period: 2019/11/9 – 2019/11/16

**Project Title : Pintos Project 2 User Program (2)**

**Submission Date : 2019/11/16**

**Members : 신은우, 최승환**

1. **Goal of Development**

* **Systematical, Cooperative Workflow**
  + This project follows workflow of ‘Agile Development’ technique. According to the changing situation, one should elastically change the plans accordingly, without losing consistency of the initial plan.
* **Stable, Reversible version control**
  + This project uses Github through Git commands. Throughout the whole project, turning points or potential danger zones should be noticed to each other teammate. Fatal errors should be prevented by commits and merges between the branches at the right moments, retaining valuable information about previous versions.
* **Functional Operating System**
  + This Operating System should be able to function all the basic functionalities this course suggests.   
    (total of 15)
* **Robust Operating System**
  + This Operating System should prevent predictable errors from various inputs and thread (or process) creation and deletions  
    (total of 26)
* **Additional exception handlers**
  + This Operating System is able to prevent few additional errors, including fatal memory leaks and synchronization. With additional No-vm problem.  
    (total of 14)

1. **Range and Contents of Development**
   1. **Range**

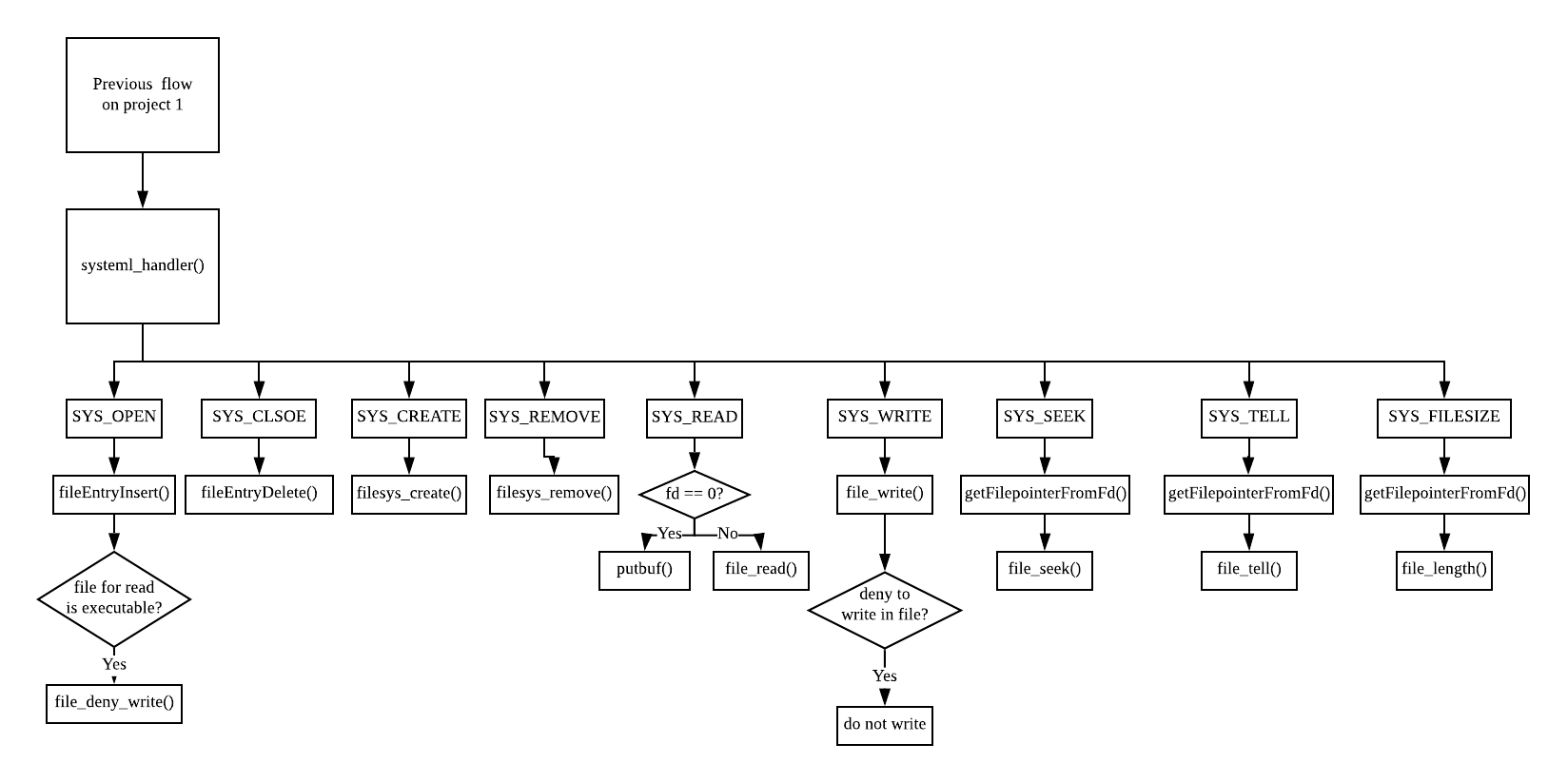
* **File Systemcalls :** Basic systemcalls for files. Write authority dependent operations included.  
  - open : open a file. return the file pointer through register.  
  - close : close a file. free the file pointer.  
  - create : create a file. return true if file is opened correctly.  
  - remove : delete a file. return true if file is removed correctly.  
  - filesize : return the size of a file through register.  
  - read : read a content of a file through file descriptor.  
  - write : write to a file through file descriptor.  
  - seek : change next byte to be written or read in file.  
  - tell : return next byte to be written or read in file.
* **File Descriptor management :** When a file was succesfully opened, return correct file descriptor. Find the according file pointer when accessed through such file descriptor.
* **Executing File management :** When a file is executed or exited, manage the list of family to search for currently executing files.
* **Write authority manipulation :** Executing files should not be written. Therefore, manipulate write authority when list of executing files changed.
* **Memory leak Re-examination :** Unfreed memory should be checked and freed.
* **Semaphore Family Re-structure :** Previous Busy wait technique was changed into semaphore wait-signal technique. New approach, better performance on speed and synchronization
  1. **Contents**
* **File descriptor / File pointer**
  + File can be opened with a file pointer. However, user program does not directly call file\_open using file pointer. Rather, user program will receive a file descriptor by opening a certain file. Using these file descriptor, kernel will open the file with file pointer according to that specific file descriptor.
* **Executing files**
  + Executing files are executables that are currently being executed. This means no other process or thread should write during the process of execution. Therefore, all execution and exit of a file should be notified to all threads existing on pintos. Then the threads’ file descriptors should modify their write authorities accordingly.
* **Semaphore**
  + Semaphore is a technique used for synchronization, using sema value to determine wheter a thread will enter the critical section.  
     For example, initial sema value(set to 1) will be decreased by 1, when first thread enters the critical section. Afterwards, when second thread tries to enter the critical section, sema value is 0. Therefore second thread will put itself into a wait queue and wait. When first thread completes execution of critical section, sema value is increased to 1 and wakes up second thread. When second thread is waken, it will continue its job. This technique will make sure only one thread will access critical section.

1. **Schedule and Methodology of Development**
   1. **Schedule**

* **file systemcalls (1) (11.9 ~ 11.12)  
  - open   
  - close**
* **file systemcalls (2) (11.9 ~ 11.11)  
  - create   
  - remove   
  - read   
  - write   
  - seek   
  - tell  
  - filesize**

* **FileEntry management (11.13 ~ 11.14)**
* **Manage executing file list (11.13 ~ 11.15)**
* **Code optimize  
  - family data structure (11.13 ~ 11.14)  
   :** change implementation of family relation and its functions **- write authority (11.13 ~ 11.15)  
   :** use foreach to deny and allow all existing threads’ authority. **- memory leak (11.15 ~ 11.15)  
   :** free memories. file close mistakes solve. entry clear function call
* **Github branch management (11.16 ~ 11.16)**
  1. **Methodology**
* **file systemcall (open, close) & File Entry management**  
  - **open** open will be called through filesys\_open() function in filesys.c. When opening a file, file descriptor should be managed together. File descriptor managing functions with, filesys\_open() should be in pair in a single function  
  - **close** close is much more simple compared to open function. However when closing a file, make sure multiple user programs does not use same file descriptor in the future.  
  - **struct FileEntry** FileEntry is the structure that contains all opened files in the current thread. Therefore, FileEntry is a member of a thread structure and be manipulated thread-dependently.
* **file systemcall (others)**  
   use basic functions in ‘file.h’ & ‘filesys.h’
* **Code optimize**  
  - **family data structure & manage executing file**  
   On the first project, parent & child thread relation was saved in family struct. However on second project, currently executing files should deny-writes. It wasn’t a efficient way to simply create another list to manage executing files independently. Therefore, we used a process table to manage each process that saves its executable file. Another change was on busy waiting technique changed to semaphore for more efficient use of time on CPU  
    
  - **memory management** kernel side malloc space was freed after it was no more of a use
* **Github branch management** Github branch management was done in a rather cautious manner since this project was a big, sensitive one. All sub-projects were implemented and tested on an individual branch and merged to each other afterwards.  
   Push on master branch was done only when a module was fully capable of its job. Less merge on branches were needed, complicated division of issues were efficiently managed on two local work directories, and the final push to master branch was also merged without any conflicts.  
  (Though conflicts doesn’t necessarily mean bad project flow, it still is a laboring job, therfore was minimized through all parts of project)
  1. **Role Sharing**
* **file systemcalls : 신은우, 최승환**
* **executing file management : 최승환, 신은우**
* **file descriptor management : 최승환**
* **Code optimize : 신은우  
  - write authority  
  - memory leak  
  - family data structure**
* **Github branch management : 최승환, 신은우**

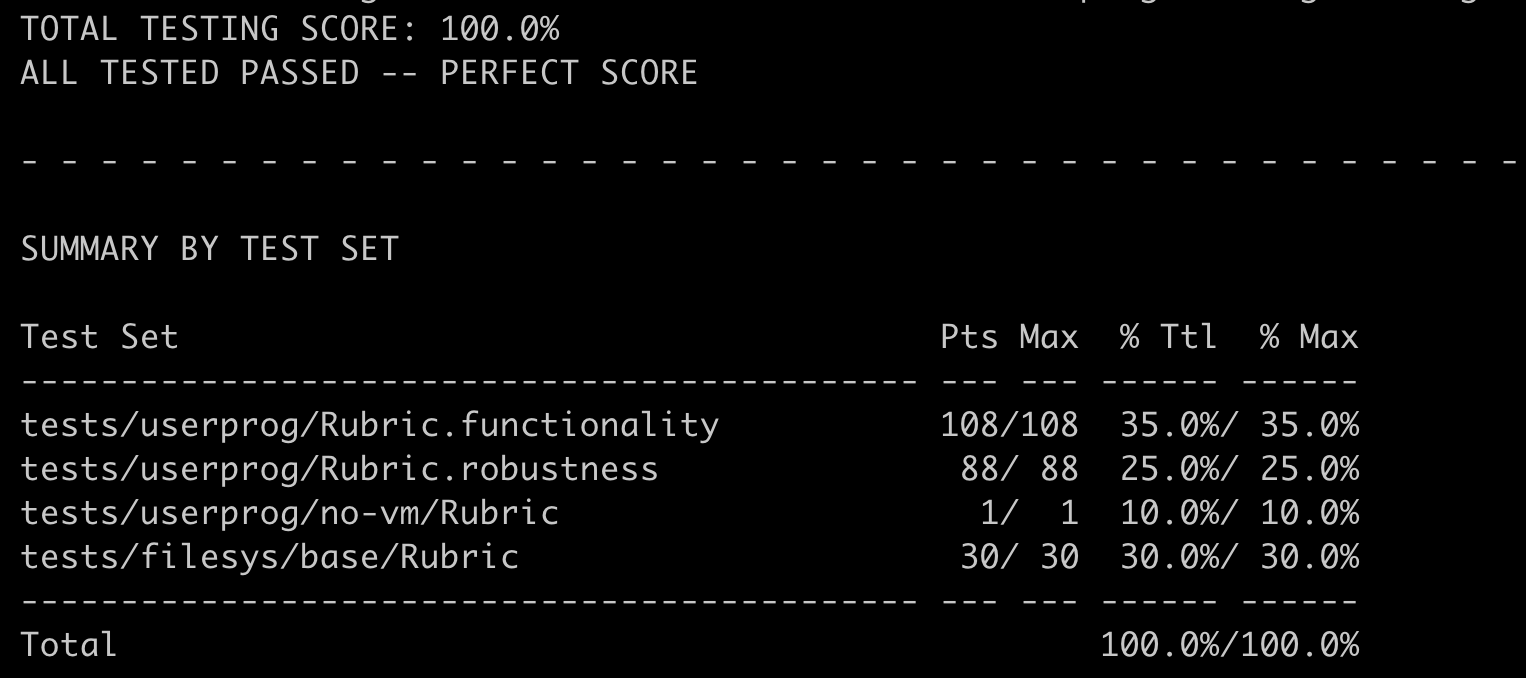
1. **Result of Research**
   1. **Composition of Contents**

****

* 1. **Development of Contents**
* **file systemcall (open, close) & File Entry management**  
  - **open** fileEntryInsert() called to insert a FileEntry to a list. Initialize struct FileEntry elements to current file name, file descriptor, file pointer. Since file descriptor should not be a value of 0, 1, 2, the file descriptor should start from 3 increasing by one on every open.  
  - **close** fileEntryDelete() calls file\_close() and deletes the current FileEntry from the list of the thread.  
  - **struct FileEntry** FileEntry is manipulated per thread. All threads (implemented as a struct thread) have their own list of FileEntry. When deleting or inserting to the list, always call thread\_current() to get the pointer of current thread.
* **file systemcall (others)**  
  - SYS\_OPEN : fileEntryInsert()을   
  - SYS\_CLOSE : fileEntryDelete()  
  - SYS\_CREATE : create a file with filesys\_create()  
  - SYS\_REMOVE : remove a file with filesys\_remove()  
  - SYS\_READ : if parameter fd is 0, call putbuf() to print to stdout, otherwise read with file\_read()  
  - SYS\_WRITE : if fd is 1, take input from stdin and copy it into the parameter buffer. If buffer is file descriptor of a file, write with file\_write()  
  - SYS\_SEEK : return file pointer with getFilepointerFromFd() and return the pointing value with file\_seek()  
  - SYS\_TELL : return file pointer with getFilepointerFromFd() and change the pointing value with file\_tell()  
  - SYS\_FILESIZE : return file pointer with getFilepointerFromFd() and return the size of file with file\_length()
* **Code optimize** struct family is managed in linked list, working as a process table. When a thread is created, it will be pushed to the list. When a thread is exited, it will not be deleted from list and will remain until parent thread calls wait to collect its exit status. Previous busy wait was also re-implemented with semaphore to reduce unnecessary CPU usage.  
   First, tid is set to ‘ready’ in **thread\_create()**. Note that, initial thread is created in **thread\_start()** not thread\_create. **init.c/run\_action()** has to push this initial thread to list with **makeFamily()**.After **process\_execute()** creates a thread and puts it into a ready queue, **familyWaitChild()** will wait until child is executed normally and return its tid. **familyCheckChildState()** will call sema\_down to collect -1 on fail, tid on success when child thread is executed normally. **start\_process()** will change the thread’s status to ‘alive’ in **familyIamAlive()** when thread is executed. Then the thread will sema\_up() to wake up the parent waiting for this thread to be executed. In case of thread “main”, **init.c/run\_action()** will call **familyIamAlive()**.

In **syscall\_handler()**, At **SYS\_EXIT** case (normally exit case) change exit value and status value to die in **struct family** by **familyIamDie()**. And exit by thread\_exit function. In thread\_exit function, by using familiyKillMe function chagne status value to kill, if present status value isn’t ‘die’ (normally exit). After that, Allow to write to file descriptor opening executable file of exiting program and increase semaphore value in it’s family struct to wake parent who implement wait function.  
 In **process\_wait()**, by **familiyCheckChildState()** decrease semaphore value of it’s family struct and delete family struct of child in family list. And then return exit value of child thread.  
 In **init.c/run\_acitionI()** when all execution is done, free memory used for family struct list.  
  
- **memory management** unfreed memory space found in load(). before exiting load(), freed malloced space.  
 unfreed memory space found in execFileEntryInsert(), freed malloced space  
 unfreed memory space found on exit. execFileEntryClear() called on thread\_exit().

* 1. **Test and Evaluation of Contents**

****

1. **Etc.**
   1. **Level of Contribution**

* **신은우 (50%)**
* **최승환 (50%)**
  1. **Thoughts for Project**
* **신은우**Before starting a project, I think it was not so hard, because I read so many code on project1. I think, implementing file system is not hard, but modifying a code for test case is challenging. So I solved each problem using much of my time. Finally when I saw all pass result, I felt sense of accomplishment. By this project I gain importance of writing how my code is run and before writing a code plan how i structurized my code.
* **최승환** Upon changing the overall logic of parent-child relation, I was a bit anxious to say we should. So we agreed to divide the job into developing the previous code and developing a new method(semaphore). The previous code had its limitation on speed, so synchronize problems caused TIMEOUT even though it executed normally given more time. Even though it leaved only 2 problems left unsolved, it seemed to have no proceeding solution further.  
   The other teammates job was being done parallely, so after he successfully implemented some logic, we decided it as a dominant algorithm. I think such work flow brought us advantage on changing situation.  
   Unlike the first project, we had big change on the workflow of the project but I think it became a good experience for real-time change in the project