Name: Lâm Lê Thanh Thế Trần Văn Duy Tuệ
ID: 1551034 1551044
APCS – HCMUS

CS333 Lab Project 02

File System with NachOS

REPORT

Contents

I. Preliminaries

1. Installing NachOS

Apply “Solution 2” in tutorial 3.1

Use given NachOS source code, including 2 folders: *nachos-3.4* and *gnu-decstation-ultrix*

Compiling steps:

- Navigate into *nachos-3.4/code*
- Open the Linux terminal and run: `gmake all`

2. Extra notes

- Submitted files include full source of NachOS as well as a copy of files that we only need to interact with in this exercise, in *Modified source only* folder.

II. Creating a system call

1. Declaring the new system call

Add into *code/userprog/syscall.h*:

- A constant definition to encode the identity value for the system call into an easier-to-remember name, e.g. SC_<SyscallName> (just use whatever integer number unused)

```
#define SC_CreateFile  18
```

- The function declaration of the system call

```
int CreateFile(char fileName[]);
```

2. Adding stub in Assembly

This helps make system call into NachOS kernel whenever the newly declared function is triggered.

Add into *code/test/start.c* and *code/test/start.s* the same code, whose pattern is similar to those of predefined system call (just copy the code for Halt or Yield and change the names):

```
        .globl CreateFile
        .ent CreateFile
CreateFile:
        addiu $2, $0, SC_CreateFile
        syscall
        j $31
        .end CreateFile
```

Must use the same name as the function declared in step 1.

3. Implementing the system call

We will create an exception to handle this call.

Modify the implementation of **ExceptionHandler** function in *code/userprog/exception.cc*:

- Create a switch case (or if case) for the value of variable **type** if it is our system call code, e.g. SC_CreateFile
- Use helper methods from class Machine to help read and write registers. The arguments are passed in starting from register r4, then r5, r6,... respectively, and the result must be returned in register r2. We can use #define to make those numbers easier to be used.

```
#define ret 2
#define arg1 4
#define arg2 5
```

```
#define arg3 6
#define arg4 7
```

- Use the array registers from class Machine to increase the Program Counter (PC) by 4 so that it points to the next instruction, else we will be stuck in a loop of the same system call forever
- Create additional cases and variables to help handle an invalid exception or system call
- We can split the code for the exception case into an independent function, for clarity of vision
- NOTE!
    - To read the file name passed in from the user, we can use some functions implemented during the previous exercises and projects (e.g. User2System in Machine class)
    - To create a file, we can use the function Create(buf, size) given by NachOS in the FileSystem class, where buf is the string pointer of file name and size is the initial size for the file
- Example code:

```
void SysCall_CreateFile() {
    int virBuf = machine->ReadRegister(arg1);
    int limit = machine->FindStrLim(virBuf), len = limit - 1;
    char *str = machine->User2System(virBuf, limit);

    if (str == NULL) {
        DEBUG('a', "Unsuccessful read from user memory at location %d\n", virBuf);
        machine->WriteRegister(ret, 0);
        return;
    }

    if (fileSystem->Create(str, 0) == false) {
        DEBUG('f', "Unsuccessful file creation of name %s\n", str);
        machine->WriteRegister(ret, -1);
    } else {
        DEBUG('f', "Successful file creation of name %s\n", str);
        machine->WriteRegister(ret, 0);
    }

    delete[] str;
}

void ExceptionHandler(ExceptionType which) {
    int type = machine->ReadRegister(ret);
    bool valid = true;

    switch(which) {
        case SyscallException:
        switch(type) {
            case SC_Halt:
            SysCall_Halt();
            break;

            case SC_CreateFile:
            SysCall_CreateFile();
            break;

            default:
            valid = false;
            break;
        }

        break;

        // case <Another type of exception>
```

```
            default:
            valid = false;
            break;
        }

        // Increment PC (even if the call is invalid)
        machine->registers[PrevPCReg] = machine->registers[PCReg];
        machine->registers[PCReg] = machine->registers[NextPCReg];
        machine->registers[NextPCReg] += 4;

        if (!valid) {
            DEBUG('a', "Unexpected user mode exception %d %d\n", which, type);
            ASSERT(FALSE);
        }
    }
```

a. Creating a file

- After defining and creating stub for system call, we modify *exception.cc* and add function `SysCall_CreateFile()` to handle the process.
- We use already defined *fileSystem* variable to call function `fileSystem->Create(str, 0)`.
- This function will create an empty file with *filename* and *size*. In this case, new file will have size of 0 byte.
- Function will return 0 if file was created successfully and -1 otherwise.

b. Opening a file

- First, we read all necessary variables from registers.
- Check if the descriptor table is full by using variable *fcur* from above.
- Because each process will have 2 reserved file descriptors for console input and ouput, we will check if user wanted to access these modes.

```
      if (strcmp(str, NCIN) == 0) {
        printf("Accessing NachOS Console Input\n");
        machine->WriteRegister(ret, 0);
        return;
    }

    if (strcmp(str, NCOUT) == 0) {
        printf("Accessing NachOS Console Output\n");
        machine->WriteRegister(ret, 1);
        return;
    }
```

- If none of the two cases above happened, we proceed to open the file and return the OpenFileId if successful and -1 otherwise.

c. Closing a file

- We get file index from register and check if it is a valid index (in range and exists in the descriptor table)

- To close a file, we just have to remove the file's *OpenFileId* from the descriptor table and set it to NULL for later use.

  ```
  delete fileSystem->openFiles[idx];
  fileSystem->openFiles[idx] = NULL;
  ```

d. Reading from file

- We check validity of file id from user's call like above.
- In the next step, we check if user wanted to access NachOS console input or not. If it is the case, we will let user input from console and save it into the *nachcin* file by using

```
fileSystem->openFiles[fileid]->Write(buf, byteres);
```

- Otherwise, we just read the content of the file into buffer and return it to user:

```
fileSystem->openFiles[fileid]->Read(buf, charcount);
```

e. Writing to file

- Check validity of file id from user's call.
- If the file user wants to write is read-only, we abort the call.
- If user wants to write to NachOS console output file, we output it to console and also write to file *nachcout*.
- Otherwise, we write content from buffer to designated file and return number of bytes written to user.

f. Seeking in a file

- In the checking validity process, we also need to check if user wants to modify console I/O to prevent it from happening.
- Otherwise, we use function Seek to get to the requested position.

III. Add a new class for further use in NachOS

This part means to help implement system calls to read/print strings/ints, which later help in the test program to create a file, interacting verbosely.

As we are not allowed to use C/C++ libraries in our implementation, we need a new class to handle input/output (I/O) on the console – the so-called ***SynchConsole*** (*synchcons.h* and *synchcons.cc*)

1. Preparing the class code files
   - Copy the class files into *code/threads*
   - Modify *code/Makefile.common* to include compiling those files at every other Makefiles

     ```
     USERPROG_H = ...
         ../machine/translate.h\
         ../threads/synchcons.h

     USERPROG_C = ...
         ../machine/translate.cc\
         ../threads/synchcons.cc
     ```

```
USERPROG_O = ... translate.o synchcons.o
```

- Note that the '\' character just denotes an 'enter' (new line) as the string continues. We will **NOT** include that at the end of the string

2. Preparing a variable to use

Let call this variable *synchCons*. We need to modify 2 files in *code/threads*, just following the pattern of the existing variable *machine*.

- For *system.cc:*

```
#ifdef USER_PROGRAM // requires either FILESYS or FILESYS_STUB
Machine *machine;   // user program memory and registers
SynchConsole *synchCons;    // for console I/O
#endif

#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg);   // this must come first
    synchCons = new SynchConsole();
#endif
```

- For *system.h:*

```
#ifdef USER_PROGRAM
#include "machine.h"
#include "synchcons.h"
extern Machine *machine;    // user program memory and registers
extern SynchConsole *synchCons;
#endif
```

- Run *gmake all* in *code* once to see if we add the class successfullly or not, then research into the class files for some functions to use later

IV. Creating a program to communicate with NachOS

1. Implementing a C program to test the new system call

Create a C (.c) source file in *code/test*

There are 2 important things:

- Include **syscall.h** to enable the program to make system calls
- Call **Halt()** to exit NachOS and come back to Linux terminal

```
#include "syscall.h"

#define FNAME_LEN   200

int main() {

    char buf[FNAME_LEN + 1];
    int fileFlag;

    PrintString("Please input a file name (max ");
    PrintInt(FNAME_LEN);
    PrintString(" characters): ");
    ReadString(buf, FNAME_LEN, '\n');

    PrintString("Creating file with name \"");
    PrintString(buf);
    PrintString("\"...\n");
```

```
        fileFlag = CreateFile(buf);
        if (fileFlag != 0)
            PrintString("FAILED!\n");
        else
            PrintString("SUCCESSFULLY!\n");

        Halt();
        return 0;
    }
```

2. Including the test program in Makefile

Modify *code/test/Makefile*:

- Add the executable file name (e.g. ascii) after **all**

```
all: ... fileManip
```

- Add the rule to compile the object and executable file for the program, following the same pattern as the others

```
fileManip.o: fileManip.c
    $(CC) $(CFLAGS) -c fileManip.c
fileManip: fileManip.o start.o
    $(LD) $(LDFLAGS) start.o fileManip.o -o fileManip.coff
    ../bin/coff2noff fileManip.coff fileManip
```

3. Running the test program

Compile NachOS again after modifying

Open the terminal in *code* and run:

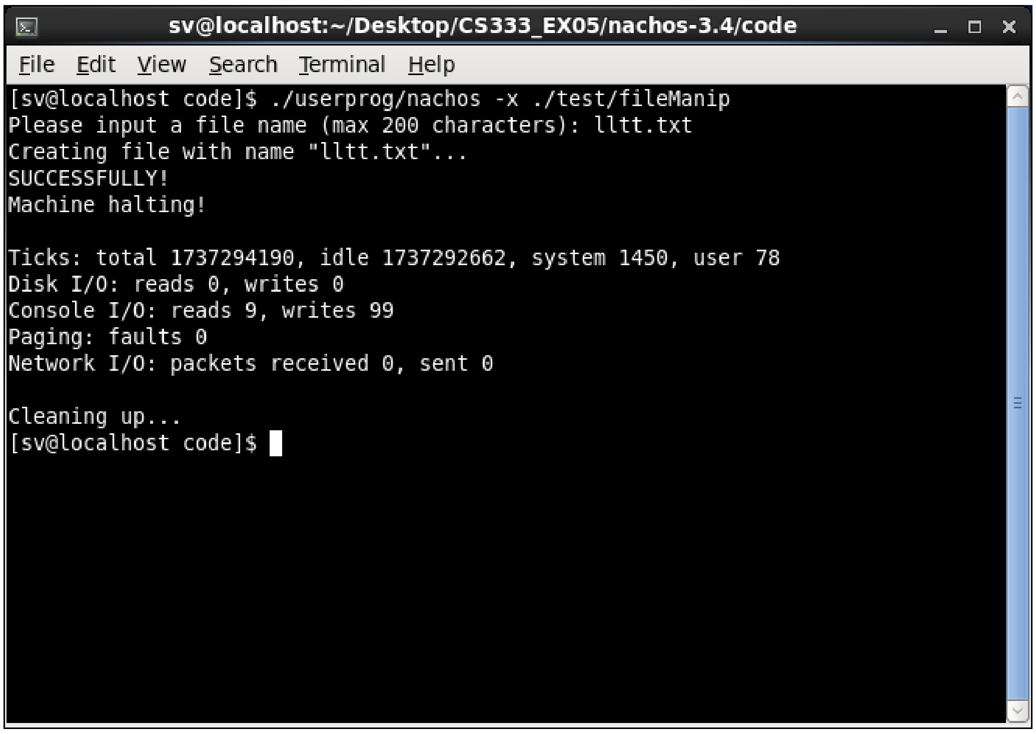
```
./userprog/nachos -x <path to executable>
```

e.g.

```
./userprog/nachos -x ./test/fileManip
```

The file created will be inside **code/**

Example results with the file “lltt.txt” created successfully

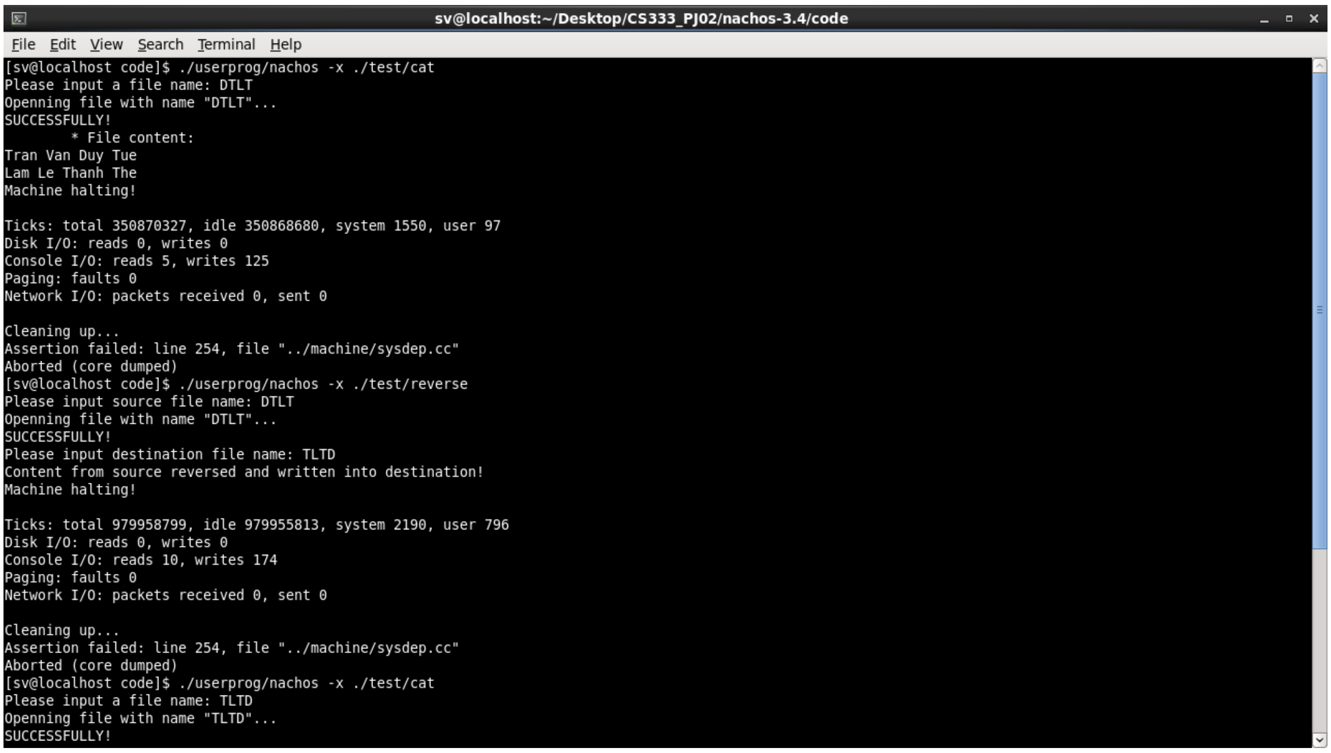

sv@localhost:~/Desktop/CS333_EX05/nachos-3.4/code
File Edit View Search Terminal Help

```
[sv@localhost code]$ ./userprog/nachos -x ./test/fileManip
Please input a file name (max 200 characters): lltt.txt
Creating file with name "lltt.txt"...
SUCCESSFULLY!
Machine halting!

Ticks: total 1737294190, idle 1737292662, system 1450, user 78
Disk I/O: reads 0, writes 0
Console I/O: reads 9, writes 99
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
[sv@localhost code]$
```

and reversing the content of "DTLT"



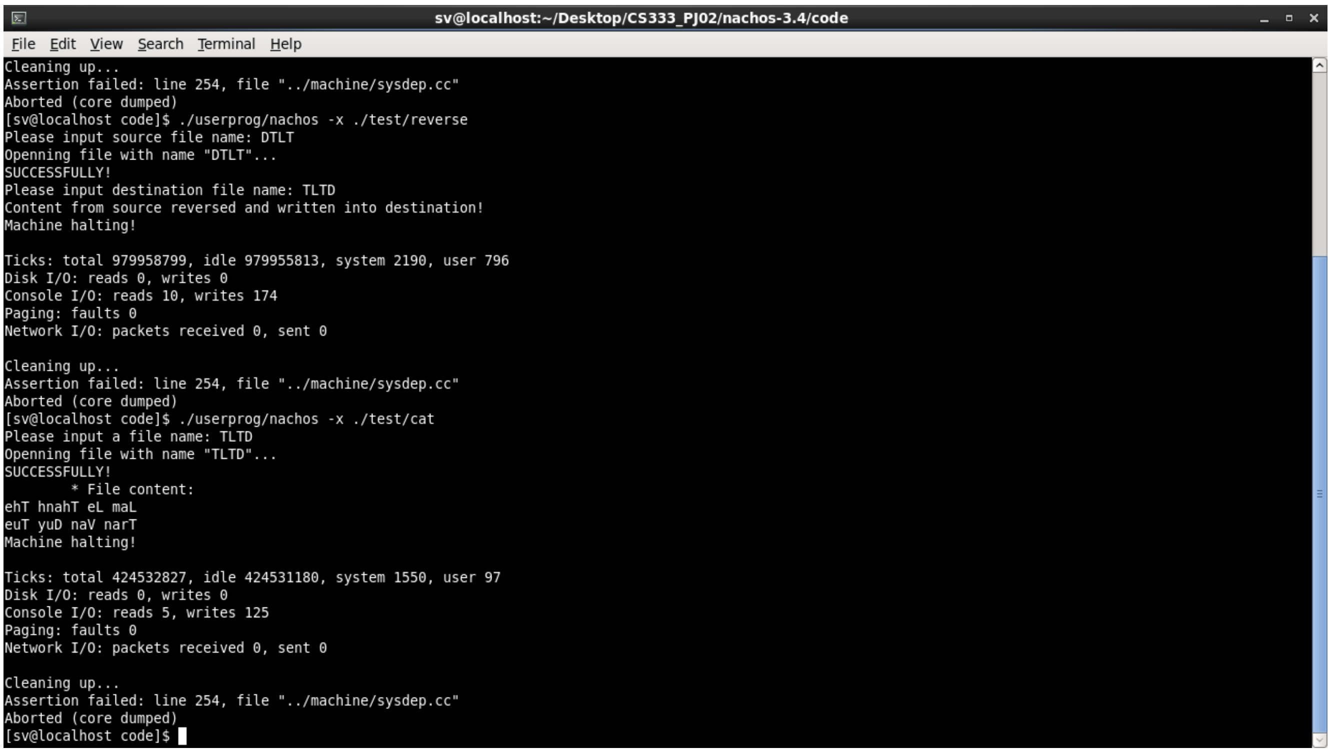
```
[sv@localhost code]$ ./userprog/nachos -x ./test/cat
Please input a file name: DTLT
Openning file with name "DTLT"...
SUCCESSFULLY!
        * File content:
Tran Van Duy Tue
Lam Le Thanh The
Machine halting!

Ticks: total 350870327, idle 350868680, system 1550, user 97
Disk I/O: reads 0, writes 0
Console I/O: reads 5, writes 125
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Assertion failed: line 254, file "../machine/sysdep.cc"
Aborted (core dumped)
[sv@localhost code]$ ./userprog/nachos -x ./test/reverse
Please input source file name: DTLT
Openning file with name "DTLT"...
SUCCESSFULLY!
Please input destination file name: TLTD
Content from source reversed and written into destination!
Machine halting!

Ticks: total 979958799, idle 979955813, system 2190, user 796
Disk I/O: reads 0, writes 0
Console I/O: reads 10, writes 174
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Assertion failed: line 254, file "../machine/sysdep.cc"
Aborted (core dumped)
[sv@localhost code]$ ./userprog/nachos -x ./test/cat
Please input a file name: TLTD
Openning file with name "TLTD"...
SUCCESSFULLY!
```



```
Cleaning up...
Assertion failed: line 254, file "../machine/sysdep.cc"
Aborted (core dumped)
[sv@localhost code]$ ./userprog/nachos -x ./test/reverse
Please input source file name: DTLT
Openning file with name "DTLT"...
SUCCESSFULLY!
Please input destination file name: TLTD
Content from source reversed and written into destination!
Machine halting!

Ticks: total 979958799, idle 979955813, system 2190, user 796
Disk I/O: reads 0, writes 0
Console I/O: reads 10, writes 174
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Assertion failed: line 254, file "../machine/sysdep.cc"
Aborted (core dumped)
[sv@localhost code]$ ./userprog/nachos -x ./test/cat
Please input a file name: TLTD
Openning file with name "TLTD"...
SUCCESSFULLY!
        * File content:
ehT hnahT eL maL
euT yuD naV narT
Machine halting!

Ticks: total 424532827, idle 424531180, system 1550, user 97
Disk I/O: reads 0, writes 0
Console I/O: reads 5, writes 125
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Assertion failed: line 254, file "../machine/sysdep.cc"
Aborted (core dumped)
[sv@localhost code]$ 
```

V. File manipulating in NachOS

1. Building the file system

The NachOS file system related code is located under *code/filesys*: *filesys.h* and *filesys.cc*

NachOS provides 2 versions for the file system: with and without *stub*

- *filesys_stub* version: this requires a manual implementation of the file system totally from scratch. This virtual file system may not interact with the one of the mother Unix operating system NachOS is running on

- *filesys* (without *stub*) version: this map each operation of the file system with the corresponding actions of the actual file system managed by the mother Unix operating system NachOS is running on, hence it can easily interact with the real hardware via the Unix (e.g. CentOS in this project). Everything we do in this file system, we can check and verify clearly in the Unix file system, e.g. if we create a file in NachOS, that file will appear under the file explorer of the Unix.

NachOS uses the without-*stub* version by default. To change this, define FILESYS_STUB in the *Makefile* inside *code/filesys*.

To limit the number of file descriptors, we can create a list of pointers of file descriptors and keep track of the most recently opened file, as well as the openning type (read only or read/write). We can check for these constraints directly in the exception managing if we make the attributes public.

Some basic methods of the file system can be:

- Open the file, with or without type specification, also add its descriptor into the list
- Close the file, removing its descriptor link
- Create a new file
- Format the file system, as an initialization
- Destructor in order to prevent memory leaking when maintaining pointer lists

In *filesys* class:

```
OpenFile** openFiles; // descriptor table, max fDescMax entries
 int fcur; // keep track of the currently selected file
```

- Every time we open a file, we need to:
    - Get the sector of file with input <name> from the *directory*.
    - If the file exists, we increase the *fcur* by 1 and add a new entry to descriptor table *openFiles*
    - We return this entry to caller if we opened successfully, otherwise we return NULL.
    - These steps are in 2 functions of `FileSystem::Open`

2. Building a class to interact with a file

We need to modify *openfile.h* and *openfile.cc* inside *code/filesys*. Remember that this class must also provide 2 versions (with and without *stub*) corresponding with those of the file system.

We can reuse most operations NachOS has to interact with a file via Unix. Just add a variable to maintain file type: console I/O, read only, read/write.

Some basic methods of the file class can be:

- Constructors with or without file type specification
- Read and write at the current or another location
- Get the current location of the file cursor
- Seek a location, also jumping to that place
- Get the length of the file content in bytes

In *openfile* class:

- Add supporting for open a file with a specific type (0 – standard file, 1 – read-only file, -1 – console input, -2 – console output)

  ```
  OpenFile(int sector, int openFileType);
  ```

VI. Further discussion on implementing system calls (in both modules)

1. Discussion on ReadString

The original system call structure does not allow a user-defined delimiter for the string to be inputted. To be more general, a refined version of the system call is implemented, with a new argument to be the delimiter, not necessarily an enter (new line).

2. Discussion on PrintString

The original system call structure does not require the user the pass in the length of the string. In practice, this is natural, but in implemental fact, this causes difficulties. A helper function to find the length of the string terminated by '\0' is implemented inside the Machine class, same scope as User2System and System2User discussed below.

3. Discussion on ReadInt

Some compilers allow inputs of long string representations of integers, and accept calculation overflow, which means if the user input a very long string for an integer, the got integer may look differently.

Here we can have two alternative approaches. First, read the whole string and reject if the string is not a valid integer pattern. Second, read as many digits as possible until an invalid character (this is chosen) and avoid overflow by stopping before the being formed integer exceeds the 4-byte limit.

Terminator for an integer string can vary. Some can read the whole line, then form the integer. Others may read up to an empty space – blank space, tabs, space feed, etc., then stop.

We may not start to read the integer string after all trailing spaces and enters are eliminated, like when the user press many enters before he/she types in a number.

4. Discussion on communication memory spaces

ReadString may require writing the read string currently stored in kernel memory into user memory, while PrintString may require fetching the passed-in string from the user memory into the kernel memory to print on the console. Therefore, two helper functions, User2System and System2User are implemented.

These two functions use methods from Machine class to directly read or write from the memory, hence they interfere directly with the system. This makes it better to implement these functions as member methods of Machine class, as what this class means, to cover internal system interference in one class.

These functions may be use in more general cases, not only for reading and printing string, so their implementations should not focus on any string characteristics. This means they should read or write memory as stated by the passed-in size in byte, and they should not care anything about the string terminator '\0'.

5. Discussion on Synch Console class

Sometimes the read function of Synch Console class gets stuck with unknown result. However, this class is not of our business.

We can test this easily by creating a simple system call that just repeatedly call the read function of Synch Console for 1 byte (1 char only). If we input (type in) too fast, the bug will occur.

6. Discussion on OpenFile

In general, we can have 2 approaches to manage file descriptors:

- Allow only one file descriptor per actual file. This can help prevent duplicated deletion or closing on one file, as well as preserve the memory while sacrificing the speed and is harder to implement.
- Allow multiple file descriptors per actual file. This will make implementation easier and the process can run faster. However, we may run out of file descriptor slots more quickly.

7. Discussion on Read and Write console I/O files

On agreement, reading console input file will require one more console input from the user (via Synch Console class), while writing will print the data on the console screen.

For consistency, we can save all console I/O in those files as if they are logs. This is the nature of having files for console I/O, isn't it?