MP1_report_36

109062233蘇裕恆：

trace code: b, d

implementation: testIO_test3, testIO_test4, start.S, syscall.h, exception.cc(http://exception.cc), ksyscall.h

109062320朱季葳：

trace code: a, c

implementation: filesys.h

SC_Halt

machine/mipssim.cc Run()

AFTER THE PROGRAM STARTED, WE HAVE THE ROUTE BELOW

1. main function initialize kernel
2. kernel initialize(Kernel::Initialize) the objects below
    - ***currentThread***
    - stats
    - ***interrupt***
    - scheduler
    - alarm
    - ***machine***
3. main function run some tests
4. main function run an initial user program
    - ***kernel->ExecAll()***
5. ExecAll() calls ***Exec(execfile[i])*** for i = 1 to execfileNum
6. Kernel::Exec(name)
    - initialize new Threads for the thread array “t”
    - initialize space in the thread
    - call fork to ***invoke ForkExecute*** function
7. ForkExecute

- call the ***Execute*** function of the AddrSpace, space of the thread t
  - t->space->Execute(t->getName())

8. AddrSpace::Execute
   - call Run() to simulate the execution of a user-level program
     - kernel->machine->Run()

The function(功能) of Run()

1. Details:
   When the program starts up, the program will run through the above route, then it calls Run(), allocate memory to the Instruction, “instr”, and set the status to user mode. Then the program will run an infinite loop in which OneInstruction(instr), OneTick(), some debug message function, and debugger are called.
   In the function, OneInstruction(Instruction *instr), the program fetches instruction from ***ReadMem(implemented in translate.cc (http://translate.cc))*** in order to set the value of the instruction and decode it.After the instruction is decoded, the program executes it by a switch case, then the program returns to Run().
   As for the function, OneTick(), the program has it to update the simulated time and check any pending interrupts to be called next.
2. Purpose:
   The purpose of Run() is to simulate execution of the user-level program in NachOS, and it will call OneInstruction() to help it to implement what the MIPS instructions need to do.

machine/mipssim.cc OneInstruction()

The function of OneInstruction()

1. Details:
   As the details of Run() mentioned, OneInstruction() is called by Run(), then we will continue to introduce more

details in this part.
At the first, the program initialized three integers listed below:

- raw:
  The value of the instruction, and since the value is stored in the main memory at the first, the program called ReadMem to fetch the data.

In ReadMem, the function calls another function called Translate() to translate the desired address into the physical address that represent the address of the data we want in main memory.Besides, we can see that in OneInstruction(), it send “4” to be the size to ReadMem(), and size == 4 represents that the data is a “word”.

- nextLoadReg & nextLoadValue:
  both of the integers are used to record the data for delayed load operation.

Then the program “decodes” the value of the instruction, and sets the type of the instruction for the use of the instructions’ implementation in OneInstruction(). Afterwards, there are some codes for debugging, and after that is the implementation of the instruction. In this part, the program initialized several int and unsign int listed below:

- pcAfter:
  pcAfter is the next PC, and in some cases, the next PC will not be the current PC increment by 4, such as executing “Branch instructions”.
- sum:
  sum is used to record the addition value for the instructions such as “Add and Addi”.After checking out that there’s no Overflow condition in the instruction’s implementation, the program rewrite the sum to registers[instr->rd].
- diff:
  diff is used to record the substraction value for the instructions such as “Sub”, and the implementation of Sub and the usage of diff is similar to the ones we mentioned above.

- tmp:
  tmp is used to help the implementation in which the program needs temporary to store the data, and put the data back after some works are done.
  For example, in the implementation of “Divu”, tmp is used to store the value of registers[instr->rs] / registers[instr->rt] and registers[instr->rs] % registers[instr->rt].
- value:
  for the instructions that have to load data, value is used to store the data they load
- rs, rt, imm(immediate)

After the above variables are initialized, the program starts to implement the MIPS instructions by switch case.Then we can see the part of ***OP_SYSCALL*** since every system calls are implemented in this case.

```
  case OP_SYSCALL:
DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallException, 0), " << kernel->stats->totalTicks);
RaiseException(SyscallException, 0);
return;
```

By the above picture, we can see that the program calls “RaiseException()” to implement system call(the details of RaiseException() will be explained in the next part in this report).
After the program returning back to OneInstruction(), it returns back to Run().
As for the other instructions, if the program won’t return immediately after the case is end, the program will call DelayedLoad() to do dalayed load operations and advance the program counter.

2. Purpose:
   The purpose of OneInstruction() is to decode the MIPS instructions and implement them case by case.

machine/machine.cc RaiseException()

The function of RaiseException()

1. Details:
   As the details of OneInstruction() mentioned, RaiseException() is called by OneInstruction(), then we

will continue to introduce more details in this part.
At the first, we want to discuss the variables sent to this function.
Since we have already know that this function is called for handling system call, the Exception type, “which”, must be “SyscallException”.
As for the integer, badVAddr, is 0 because there isn’t any error happened, but it will not be 0 in other cases.
For example, in the picture below, we can see that there is an address error exception, and tmp (tmp = registers[instr->rs] + instr->extra) is sent to RaiseException().

```
    case OP_LH:
    case OP_LHU:
  tmp = registers[instr->rs] + instr->extra;
  if (tmp & 0x1) {
      RaiseException(AddressErrorException, tmp);
      return;
  }
```

By the above observation, we can see that BadVAddr is the virtual address that causes software interrupt.
As for the codes in RaiseException(), the program enrolls BadVAddr to the registers[BadVAddrReg], then the program calls DelayedLoad to finish all the Delayed load process before the program trapped to kernel mode.
After the processes in user mode are done, the program changes status to System mode (aka kernel mode), then calls ExceptionHandler(which) to handle a system call.
After the program returns back to RaiseException(), the changes status back to user mode.

2. Purpose
   The purpose of RaiseException() is to change the status from user mode to kernel mode when handling system call, and set it back to user mode after the execution of the system call is done.

userprog/exeception.cc ExceptionHandler()

THE FUNCTION OF EXCEPTIONHANDLER()

1. Details
   As the details of RaiseException() mentioned, ExceptionHandler() is called by RaiseException(), then we will continue to introduce more details in this part.
   At the first, the program initialized some variables, including val, type, status ,and so on.
   Among those variables, the most important one must be “type”, which means what type of system call is the program going to execute, and we get its value from registers[2].
   There is a nested switch case to implement the system call.
   The outer one has two cases including system call and default, and the default one means that the exception is unexcepted, which means that there must be some errors happened in user mode.
   As for the inner one, the program uses the “type” it declared before to distinguish different cases of system call, and implement them case by case.
   Then since the system call we want to implement is “SC_Halt”, the program calls “SysHalt()” to help us executing it.
2. Purpose
   The purpose of ExceptionHandler() is to execute different types of exception, especially for the system call exception, the program calls different functions that implement the system calls according to the type of them.

userprog/ksyscall.h SysHalt()

THE FUNCTION OF SYSHALT()

1. Details
   SysHalt() is called by ExceptionHandler(), and SysHalt() calls Halt() function.
2. Purpose
   Trigger an interrupt, Halt().

machine/interrupt.cc Halt()

THE FUNCTION OF HALT()

1. Details
   Halt() is called by SysHalt(), and the program prints out the performance statistics and delete the kernel to “halt” it.
2. Purpose
   Shut down the NachOS program and print the performance statistics.

SC_Create

TYPE

one of the exceptions that the exceptionhandler has to deal with.

userprog/exeception.cc ExceptionHandler()

As we’ve trace in the SC_HALT, the SC_CREATE is something that an ExceptionType like SC_HALT

```
case SC_Create:
val = kernel->machine->ReadRegister(4);
{
char *filename = &(kernel->machine->mainMemory[val]);
//cout << filename << endl;
status = SysCreate(filename);
kernel->machine->WriteRegister(2, (int) status);
}
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
return;
ASSERTNOTREACHED();
break;
```

STEPS

1. store the value of the fourth register (first arguement) into the val
2. extract the content of &(kernel->machine->mainMemory[val]) and make the *filename pointer point to the content
3. call the SysCreate(filename) to create the desired space in Filesystem
   we first go through the things next section and do the following steps below
4. whether SysCreate(filename) succeed or not. We write the result to the second registor.
5. Make all PrevPCReg / PCReg / NextPCReg + 4

userprog/ksycall.h SysCreate()

1. call the FileSystem::Create() and return 1 if success otherwise 0

We intoduce the how the original filesystem do when implementing the create

filesys/filesys.h FileSystem::Create()

1. It will first create a directory with size of 10 and Fetch the content of directoryFile, which is a private variable of FileSystem that represents “Root” directory whom has list of file names, represented as a file (But it will be delete after all events)

TRUE OR FALSE

If any thing happened below, the Create() will return false otherwise it will be true

1. the file is already in directory (directory->Find(name) != -1)
    1. Find() if a funtion that just scan through all inUse table and check if the file exists

```
Directory::FindIndex(char *name)
{
    for (int i = 0; i < tableSize; i++)
        if (table[i].inUse && !strncmp(table[i].name, name, FileNameMaxLen))
        return i;
    return -1;      // name not in directory
}
```

    Find() is differ from FindIndex() but the concept is nearly the same while Find return the sector of table[FindIndex()]

If we pass this , we will generate a freemap and then we can move on to the other requirement

2. no free block for file header (sector == -1) // sector = freeMap->FindAndSet()
    The implementation of FindAndSet() is below :
    1. find and set a clear bit , or return -1 if there’s no such a bit amd named that as sector

```
Bitmap::FindAndSet()
{
    for (int i = 0; i < numBits; i++) {
    if (!Test(i)) {
        Mark(i);
        return i;
    }
    }
    return -1;
}
```

3. no space in directory (!directory->Add(name, sector))

1. Add's implementation is like below :

```
Directory::Add(char *name, int newSector)
{
    if (FindIndex(name) != -1)
    return FALSE;

    for (int i = 0; i < tableSize; i++)
        if (!table[i].inUse) {
            table[i].inUse = TRUE;
            strncpy(table[i].name, name, FileNameMaxLen);
            table[i].sector = newSector;
        return TRUE;
    }
    return FALSE;   // no space.  Fix when we have extensible files.
}
```

2. It will return false as long as the content already exist or there's no space in the table

If none of above hapened , the Create() will writeBack those parameter (sector, directoryFile , freeMapFile) and return true

The sequence in priority: directory -> BitMap(freeMaoFile) -> sector

But we use the FILESYS_STUB in MP1

I will briefly introduce it , it will call OpenforWrite

```
int
OpenForWrite(char *name)
{
    int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);

    ASSERT(fd >= 0);
    return fd;
}
```

In the **sysdep.cc** (http://sysdep.cc) and will return the file descriptor of the file.

```
bool Create(char *name) {
int fileDescriptor = OpenForWrite(name);

if (fileDescriptor == -1) return FALSE;
Close(fileDescriptor);
return TRUE;
}
```

And then after checking the filedescriptor is valid , it will close the file and return true

SC_PrintInt

userprog/exception.cc ExceptionHandler()

THE FUNCTION OF EXCEPTIONHANDLER()

1. Details
   After the program starts, the route is similar to the one of SC_Halt since both of SC_PrintInt and SC_Halt are system call, and the first difference between them appears in ExceptionHandler().
   We can seperate the code that implements case SC_PrintInt into three parts to explain it.
   - read value that is going to be print from registers[4]
   - call SysPrintInt() and send the value read just now to it.
   - update the program counters below:
     PrevPCReg, PCReg, NextPCReg
2. Purpose
   The purpose of ExceptionHandler() is just the same as what we explained in the SC_Halt part.

userprog/ksyscall.h SysPrintInt()

THE FUNCTION OF SYSPRINTINT()

1. Details
   SysPrintInt() is called by ExceptionHandler(), and the program just calls PutInt() function and debugs in this part.
2. Purpose
   The purpose of SysPrintInt() is to call PutInt() function in synchconsole.

userprog/synchconsole.cc PutInt()

THE FUNCTION OF PUTINT()

1. Details
   PutInt() is called by SysPrintInt(), and we will seperate the implementation in this part into three parts to explain it.

- lock->Acquire():
  the goal of Acquire() is to wait until lock is free
  - call semaphore->P():
    the goal of P() is to make the thread sleep until value

> 0(the lock isn't buzy), and we will seperate the codes in P() into 2 parts to explain it.
    - Change the status to "IntOff" and save the old status(IntOn) by calling SetLevel()
        - SetLevel():
          the program change level to the new one(the value sent to this function) and return the old one
    - Run a loop while value == 0, and calls Append() and Sleep() to disable the interrupts
        - queue->Append(currentThread):
          add currentThread to the queue in which the threads waiting in P() for value > 0 are all in this queue
        - currentThread->Sleep(FALSE):
          set the status to "Blocked" and stay idle until there's an interrupt, then call Scheduler->Run() and not to destroy the old thread(finishing == FALSE) but to run the old thread.
    - decrement the value(the lock is buzy)
  - set the lockHolder to be the current thread
- do while loop when str[idx] != '\0', and calls ConsoleOutput::PutChar() to write a char(this function will be explained later in this report) then call waitFor->P() (waitFor is a semaphor) to wait for a call back
- lock->Release():
  the goal of Release() is to free the lock
  - let lockHolder to be NULL
  - semaphore->V():
    the goal of V() is opposite to P(), the program here Run the threads in queue if it isn't empty, then increment the value(the lock isn't buzy)

3. Purpose
   The purpose of PutInt() is to call PutChar() to print integer, and lock the string we want to print in order to prevent from any changes in the string.

userprog/synchconsole.cc PutChar()

THE FUNCTION OF PUTCHAR()

1. Details
   In the program, SynchConsoleOutput::PutChar() is only called by Kernel::ConsoleTest() to test the console device.
   The implementation in this part is similar to the one in SynchConsoleOutput::PutInt().
   Lock ch and call ConsoleOutput->PutChar(ch) to print the char, then wait for a call back.
   Finally, release the lock.
2. Purpose
   The purpose of SynchConsoleOutput::PutChar() is to call ConsoleOutput::PutChar() to print char, and lock the char we want to print in order to prevent from any changes in the char.

machine/console.cc PutChar()

THE FUNCTION OF PUTCHAR()

1. Details
   ConsoleOutput::PutChar() is called by PutInt(), and we will seperate the implementation in this part into four parts to explain it.

- ASSERT(putBuzy == FALSE)
  In this part, the program will check the condition(putBuzy == FALSE).
  If the condition is false, then the program will print a message and call Abort() to abort the process.
  As a result, we can ensure that we do the PutChar operation when it is available.
- WriteFile(writeFileNo, &ch, sizeof(char))
  In this part, the program will write “ch” into a file.
- putBusy = TRUE
  Set putBuzy to be TRUE to prevent it from being used by the other threads.
- kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt)

To schedule an interrupt to write int.
(the details will be discussed in the next part)

2. Purpose
   The purpose of ConsoleOutput::PutChar() is to write the character sent to this function to the simulated display, and schedule an interrupt generated by the hardware device that is responsible for writing int.

machine/interrupt.cc Schedule()

THE FUNCTION OF SCHEDULE()

1. Details
   Schedule() is called by ConsoleOutput::PutChar(), and we will seperate the implementation here into three parts to explain it.

- set the integer, “when”, to tell CPU when will the interrupt happen
- initialize a new Pending interrupt with the arguments sent to this function
- insert the one initialized just now to the sorted list, “pending”.

2. Purpose
   Schedule an interrupt to occur at “when”.

machine/mipssim.cc Run()

The route, details, and purpose here is identical to the explanation in part a, including setting status, calling OneInstruction() and OneTick() in a infinite loop to achieve the function()'s goal.

machine/interrupt.cc OneTick()

THE FUNCTION OF ONETICK()

1. Details
   OneTick() is called by Run(), and we will discuss the function of OneTick() in several points.

- advance the simulated time according to the status
  - e.g. update systemticks in system mode and update userticks in user mode.

- before enabling interrupts, turn off them first and call CheckIfDue() to check there's no pending interrupts.
  After there's no pending interrupts, re-enable the interrupts.
  Finally, if it is allowed to conduct context switch(yieldOnReturn == TRUE), set to status to system mode to do the process.
  When the process is done, set the status back to user mode.

2. Purpose

- advance the simulated time
- check if there is any pending interrupts and wait for them until there's no pending interrupts, then re-enable the interrupts.

machine/interrupt.cc CheckIfDue()

1. Details
   CheckIfDue() is called by OneTick(), and we will discuss the function of it in several points.

- Check the interrupts are disabled to invoke an interrupt handler
- return false if
  - there's no pending interrupts
  - advance clock is false(we can't advance clock to the time when next pending interrupt occur)
- return true if the program fire off any interrupt handler, and the procedure are shown below
  - take the first pending interrupt in the queue
  - if the time doesn't reach the time when the pending interrupt occur, then advance the clock
  - enable the interrupt handler
  - remove the pending interrupt in the front from queue and make "next" point to the removed one.
  - call back to the interrupt handler
    - next->callOnInterrupt->CallBack()
  - delete the interrupt handled

    - repeat the above procedure if the queue storing pending interrupt isn't empty, and reaches the time when the interrupt should be execute
    - disable the interrupt handler
2. Purpose
   To check the pending interrupts are scheduled to occur, and fire off the executed one.

machine/console.cc CallBack()

1. Details
   In Schedule(), we initialize the PendingInterrupt object with toCall, a ConsoleOutput.
   As a result, when we called CallBack() in CheckIfDue(), it means ConsoleOutput::CallBack().
   The procedure in this function will be shown below.

- set putBuzy to false
- call back interrupt handler
    - callWhenDone->CallBack()
        - callWhenDone is a SynchConsoleOutput object since a new ConsoleOutput object is initialized the former one.
        - CallBack() here is SynchConsoleOutput::CallBack()

2. Purpose
   When the next character can be display, this function will be called in order to call back the interrupt handler to do the thing we want.

userprog/synchconsole.cc CallBack()

1. Details
   As what we mentioned in the details in ConsoleOutput::CallBack(), the CallBack() called in that function is SynchConsoleOutput::CallBack().
   Then the procedures of this function will be shown below.

- waitFor->V()
    - As what we mentioned in the details in PutInt(), the usage of V() is to Run the threads in queue if it isn't empty, then increment the value

- The usage of waitFor is to wait for a call back
- After the interrupt is handled, return.

2. Purpose

   Make Interrupt handler to handle the desired interrupt, and call back when the procedure is done.

How the arguments of system calls are passed from user program to kernel?

```
case SC_Create:
DEBUG(dbgSys, "Call create.\n");
val = kernel->machine->ReadRegister(4);
{
filename = &(kernel->machine->mainMemory[val]);
//cout << filename << endl;
status = SysCreate(filename);
kernel->machine->WriteRegister(2, (int) status);
}
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
return;
ASSERTNOTREACHED();
break;
```

We can take the implementation in SC_Create case in **exception.cc** (http://exception.cc) for example, we can see that the program get the argument from ReadRegister function.
As a result, we can say the arguments of system calls are passed using registers.

code/test/Makefile

Componant

```
CC = $(GCCDIR)gcc
AS = $(GCCDIR)as
LD = $(GCCDIR)ld
```

The CC is the abbriation for **c compiler** , and the meaning for $(GCCDIR)GCC is that we redirect the c compiler to the gcc (abbrivation for GNU compiler collection)
See the following link for more precise explanation

```
https://www.cnblogs.com/zhouyinhui/archive/2010/02/01/1661078.html
```

舉一反三

By the previous arguement , we can found out that the $(...) [thing],
the thing is the command we execute in the linux environment.

And thus the AS is link to command as in linux (Using as command, we can read and assemble a source file. - google)
LD is link to ld (It produces an executable object file that can be run. - google)

INCDIR , CFLAGS

INCDIR -> include thedirector for compile
CFLAG -> compiler command for C

https://blog.csdn.net/qq_40309341/article/details/113541112

ifeq

```
ifeq ($(hosttype),unknown)
PROGRAMS = unknownhost
else
# change this if you create a new test program!
#PROGRAMS = add halt shell matmult sort segments test1 test2 a
PROGRAMS = add halt createFile fileIO_test1 fileIO_test2 LotOfAdd
endif
```

Similar to if in c and in this example , we add (add, halt ...etc) to the PROGRAM variable that we can use next time

https://blog.csdn.net/u010312436/article/details/52459609

explaination

```
121 start.o: start.S ../userprog/syscall.h
122     $(CC) $(CFLAGS) $(ASFLAGS) -c start.S
123
124 halt.o: halt.c
125     $(CC) $(CFLAGS) -c halt.c
126 halt: halt.o start.o
127     $(LD) $(LDFLAGS) start.o halt.o -o halt.coff
128     $(COFF2NOFF) halt.coff halt
```

Since all are the implementations are similar , we use the three top case as our example.
Some thing I forgot to mention -> we include Makefile.dep and it contain some flags (e.g. LDFLAG...)

```
ifeq ($(osname),Linux)
# full path name of your cpp program i.e.:
CPP = ../../usr/local/nachos/lib/gcc-lib/decstation-ultrix/2.95.2/cpp
# directory in which your gcc cross-compiler lives i.e.:
GCCDIR = ../../usr/local/nachos/bin/decstation-ultrix-
LDFLAGS = -T script -N
ASFLAGS = -mips2
CPPFLAGS = $(INCDIR)
COFF2NOFF = ../../coff2noff/coff2noff.x86Linux
hosttype = x86Linux
endif
```

Similar concept doesn't worth mentioning

```
Target ... : Prereq...
Command ...
```

The above is the core of the process
Target: The file that is meant to be build
Prereq: The file that is used to build the target , nust exist or it will cause error
Command: The command that shell will excute
For the first (line 121):
start.o : start.S .../userprog/syscall.h , we can find the Start.S in same folder , and can also the second file exist.
We make the start.o by cflags and asflags , $(CC) here can be represent by gcc and so does other statement. (Since the .S is the assembly file and we need asflag to compile)
halt.o : halt.c -> make the halt.o file from halt.c
halt : halt.o start.o
$(LD) $(LDFLAGS) start.o halt.o -o halt.coff
$(COFF2NOFF) halt.coff halt
link the .o file together and make an executable file halt , we make the .o file to Common Object File Format and finally make it to executable file

```
clean:
    $(RM) -f *.o *.ii
    $(RM) -f *.coff

distclean: clean
    $(RM) -f $(PROGRAMS)
```

same as before , as we understand how the make work , we know that clean is to remove all the .0 .ii .coff file.
And distclean is to also remove the excutable file
Worth reading
https://www.cnblogs.com/balaamwe/archive/2012/03/15/2397998.html (https://www.cnblogs.com/balaamwe/archive/2012/03/15/2397998.html)
https://hackmd.io/@sysprog/gnu-linux-dev/https%3A%2F%2Fhackmd.io%2Fs%2FSySTMXPvl?type=book (https://hackmd.io/@sysprog/gnu-linux-dev/https%3A%2F%2Fhackmd.io%2Fs%2FSySTMXPvl?type=book)

Second Part

In this part, we implement four I/O system calls according to the hint mentioned in the slide.
Then we will explain our implementation in every file.

test

start.S

In this part, the program place the code for the system call to register2, and go to handle the system call.
Then return.
As for our implementation, we take consideration of the format of the other system calls to implement the assembly code of Open, read, write, close.

fileIO_test3.c & fileIO_test4.c

In this part, we take reference on fileIO_test1 and fileIO_test2 to write new testbenches in order to testify the other cases that aren't included in the original testbenches.

userprog

syscall.h

In this part, we cancel the comment out of the #define for the four system calls.

ksyscall.h

In this part, we make the function including SysOpen, SysRead, SysWrite, SysClose to call the corresponding function in filesys/filesys.h in which the system calls are implemented here.

exception.cc (http://exception.cc)

In this part, we add four system calls to the switch case, then read the arguments from registers, and send them to the corresponding function implemented in ksyscall.h.
After the execution of the system call is done, we write back its status to register2, and update the program counter.

filesys

filesys.h

In this part, we will explain the functions seperately. Besides, in order to record the opened file, we set a new variable called file_opened, and we initialize it in the constructor of FileSystem.

OpenFileId OpenAFile(char *name)

- take the fileDescriptor value from OpenForReadWrite(), and check the value of fileDescriptor.If it is -1, then the file with the name sent to this function isn't opened, so we return -1.
- add file_opened by 1 and check whether the value is in the legal interval(file_opened >= 0 && file_opened <= 20), if it's illegal, then substract file_open by 1 and return -1.
- find a empty space in OpenFileTable to put a newly opened file(use fileDescriptor to initialize it) to it.
- return the index of the OpenFileTable that store the newly opened file.

int WriteFile(char *buffer, int size, OpenFileId id)

- check if the id is valid(id >= 0 && id <= 19) or return -1.
- check if the file stored in OpenFileTable[id] is exist, or return -1.
- check the size of character we want to write is positive, or return -1.
- call OpenFile::int Write(char *from, int numBytes) to implement the system call, and record the return value, retVal.
- if retVal < 0, then return -1.
  else return retVal.

int ReadFile(char *buffer, int size, OpenFileId id)

- check if the id is valid(id >= 0 && id <= 19) or return -1.
- check if the file stored in OpenFileTable[id] is exist, or return -1.
- check the size of character we want to read is positive, or return -1.

- call OpenFile::int Read(char *into, int numBytes) to implement the system call, and record the return value, retVal.
- if retVal < 0, then return -1.
  else return retVal.

INT CLOSEFILE(OPENFILEID ID)

- check if the id is valid(id >= 0 && id <= 19) or return -1.
- check if the file stored in OpenFileTable[id] is exist, or return -1.
- call destructor of the opened file stored in OpenFileTable[id]
- set OpenFileTable[id] to NULL
- substruct opened_file by 1
- return 1

Conclusion

- 蘇裕恆: 我寫了兩個新的testbench，其中約有12個斷點，如果助教需要的話可以給明年，令人最頭痛的應該是在寫testbench時他老舊的c使用方法 for(int i = 0 ; i < k ; i++) 都會報錯。我也是寫了頗久才發現如何正確的declare function。同時，希望助教可以將討論區問的問題寫加入明年的spec中，因為今年的spec在實作上沒有太多的舉例。而助教給予的testbench測試點太少，我與助教確認了蠻多實作上的細節 (對於問蠻多問題我感到比較抱歉。我當然有將code全部trace過一次，問助教的問題我基本上都自己有想法，但我認為很多會遇到的問題其實都是一樣的，將spec寫清楚也可以減少助教的loading)
- 朱季葳：這是我們這學期的第一次作業，然後一開始我們花了蠻多的時間在trace code的，因為每一個函式中所呼叫到的所有函式我們都有去trace一遍，然後有些部分真的理解蠻久的，也上網找了很多資料。不過最後成功理解那些code在幹嘛之後真的蠻有成就感的。
  至於implement的部分我們一開始沒有花很多時間然後就通過了模板給的兩個testbench，但是後來看到討論區上面有同學問到一些我們沒有考慮的的情況，所以就再寫了兩

個testbench來測試其他測資，也花了一些時間debug才通過新的tb。