MP1: System Call

Cover page

- 徐竣霆：
  - Trace SC_Halt, SC_Create
  - Implementation part
  - Write report
- 江承紘：
  - Trace SC_PrintInt, Makefile
  - Implementation part
  - Write report

Trace code

SC_Halt

MACHINE/MIPSSIM.CC

Machine::Run()

Details:

Run() will first allocate memory for the instruction(instr), set to UserMode.
Then will start an infinite loop, keep fetching and runing instructions(OneInstruction()). The debugging messages are checking the Ticks for running OneInstruction() and OneTick(). The OneTick() will update the simulated time and check if there are any pending interrupts to be called(at **interrupt.cc** (http://interrupt.cc)).
Debugger: lots of system calls, print out the contents of memory, the state of CPU…

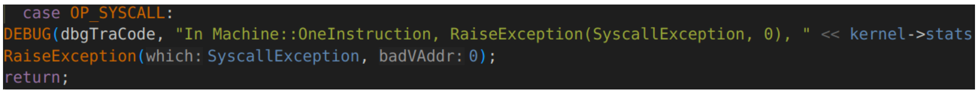
Purpose:

Simulate the execution of a user-level program on Nachos.
When the program starts, the kernel will call Run() and keep calling OneInstruction(); never return.

Machine::OneInstruction()

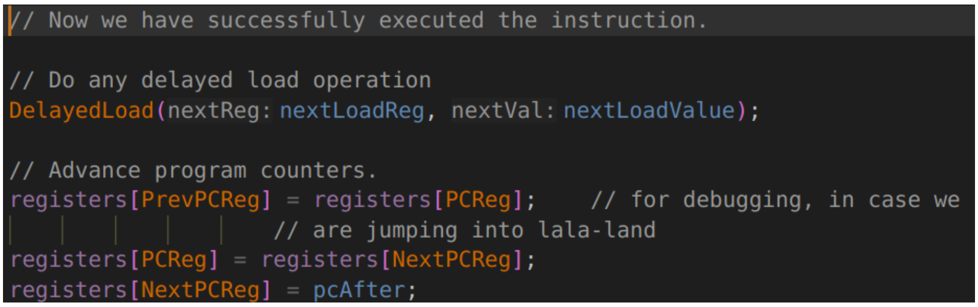
Details:

It will first create 3 int(raw(binary representation of the instruction), nextLoadReg, nextLoadValue). Then start fetching instruction.
Instruction Fetching:
Call ReadMem, read the content of registers[34](PC), 4 bytes, load the data to raw. If fail to translate from virtual to physical memory(return false), raise exception.(at **translate.cc** (http://translate.cc)) Otherwise, call Decode to decode the instruction into a MIPS instruction(->opCode, their are numerous #define in mipssim.h; ->rs 、rt 、st, slice the part; ->extra(if needed, immediate, offest…). Use OpString to format the instruction(for output). Use the pcAfter to represent the next pc. Then use the switch-case to handle the OPs(I think this part is more related to Computer Architecture, just believe it will handle all of the OPs correctly, raise exceptions when needed).



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

The important thing is this part, where it handles System Calls. It will call RaiseException() to transfer to kernel mode, deliver the task to ExceptionHandler to handle the syscall, pass down the virtual address(0, of no use) to RaiseException().
After that, determine whether the pipeline will be stalled or not, using nextLoadReg & nextLoadValue to record the needed data for delayed load.



```
// Now we have successfully executed the instruction.

// Do any delayed load operation
DelayedLoad(nextReg: nextLoadReg, nextVal: nextLoadValue);

// Advance program counters.
registers[PrevPCReg] = registers[PCReg];    // for debugging, in case we
                                            // are jumping into lala-land
registers[PCReg] = registers[NextPCReg];
registers[NextPCReg] = pcAfter;
```

Yipee!
If there is any kind of exception or interrupt, the exception handler will be invoked, when it returns, it return to Run(), continue to loop.

Purpose:

Fetch the instruction, decode it to MIPS, implement every type of instrction case by case, and run the instruction(also, determine the position of next PC). After the instruction is finished, return to Run(), so that it could continue.

MACHINE/MACHINE.CC

Machine::RaiseException()

Details:

Called by OneInstruction()(for this Path).
The parameter “which” is ExceptionType, such as Overflow, AddressError, SystemCall…
The debug message will show the type of the Exception.
Also, the virtual address that causes the trap will be saved in registers[39].
Call DelayedLoad to finish all the delayed instructions. And then trap to kernel(SystemMode). In kernel mode, handle the exception(ExceptionHandler()). After the ExceptionHandler() return back to RaiseException(), set the status back to user mode.

Purpose:

To raise Exceptions.
Change from UserMode to SystemMode.
Pass down the parameter “which” to ExceptionHandler().
After the Exception finished, set back to UserMode.

USERPROG/EXECEPTION.CC

ExceptionHandler()

Details:

It will handle the Exception RaiseException() raised.
Get the type of Exception from registers[2]. Using switch-case to determine which system call it is or other(nested switch-case).

Next part implement the System Calls case by case(For SC_Halt, call SysHalt() in ksyscall.h…). For exceptions other than System Call, it will only show error message(Unexpected user mode exception, like arithmetic exception). For SC_Halt, it will call SysHalt()(in ksyscall.h).

Purpose:

Entry point into the Nachos kernel. Do System Calls or addressing or arithmetic exception.

USERPROG/KSYSCALL.H

SysHalt()

Details:

Call Halt(), which is in **interrupt.cc** (http://interrupt.cc)(kernel->interrupt->Halt()).

Purpose:

Trigger the Halt(), which is an interrupt.

MACHINE/INTERRUPT.CC

Interrupt::Halt()

Details:

Call Print(), print out the performance statisics(in **stats.cc** (http://stats.cc)). Delete the kernel, fulfilling a "Halt".

Purpose:

Shut down the NachOS, after printing out the performance statisics, delete it.

- Discussion:
  For SC_Halt: we don't need arguments, the only thing we need to pass down are the "type"(by putting it into registers[2]) and the "which"(pass by reference), for ExceptionHandler()(work in kernel) to determine which what system call to do.

SC_Create

USERPROG/EXECEPTION.CC

ExceptionHandler()

Details:

As the comments said:
// system call code – r2
// arg1 – r4
// arg2 – r5
// arg3 – r6
// arg4 – r7
The result of the system call will be put back into r2.
For SC_Create, it need an argument(arg1, which is in r4).
Use the ReadRegister() to read in val, which represent the position in physical memory. Get the filename from the mainMemory, then call SysCreate(filename). After the SysCreate(), write the result to r2. Then move on, make PrevPCReg -> PCReg, (PCReg, NextPCReg) -> PCReg+4.

Purpose:

Entry point into the Nachos kernel.

USERPROG/KSYSCALL.H

SysCreate()

Details:

Call the FileSystem::Create(), return success or not(1 : 0).

Purpose:

Trigger the Create(), which is an interrupt.

FILESYS/FILESYS.H

FileSystem::Create()

Details:

Get the filename passed from SysCreate(). Since it's in the STUB mode, we don't need to trace the code in **filesys.cc** (http://filesys.cc). Create(char *name) will call OpenForWrite, return a flieDescriptor. If return == -1, return FALSE, else close the file (just create, won't do anything else), return TRUE.
The OpenForWrite() is implemented in **sysdep.cc** (http://sysdep.cc). It will call the open()(in fcntl.h). It's just like the thing we learned in UNIX programming(~~Call function in fcntl.h~~).
The flags mean:
O_RDWR: open for read, write
O_CREAT: create a file, if not existing.
O_TRUNC: set the file length to 0.

Purpose:

Since it's in the STUB mode, we don't need to implement NachOS's file system in this homework. We just need to make use of the existing function in UNIX.

- Discussion:
  For SC_Create: In addition to "type"(registers[2]), we also need to pass the filename(registers[4]). The filename is passed to open() to create the file named "$filename".

SC_PrintInt

USERPROG/EXECEPTION.CC

ExceptionHandler()

Details:

- Get the system call code from reading `reg2` .
- If the exception is a `SyscallException` , we determine the current exception type as `SC_Halt` , `SC_PrintInt` , `SC_MSG` , `SC_Create` , `SC_Add` , `SC_Exit` , or others. If it is other, an Unexpected system call error will be reported. If the exception is not a `SyscallException` , we throw the error: "Unexpected user mode exception."
- In this case, after finish `SC_ADD` , we receive a

`SyscallException` whose type is `SC_PrintInt` .

- Read the first argument from `reg4` as `val` .
- Call `SysPrintInt()` to print the parameter `val` we have get.
- Update the program counter.

Purpose:

Entry point into the Nachos kernel. Called when a user program is executing, and either does a syscall, or generates an addressing or arithmetic exception.

USERPROG/KSYSCALL.H

SysPrintInt()

Details:

- This function is called by `ExceptionHandler()` in `userprog/exeception.cc` .
- Call `PutInt()` to print the parameter `val` .

Purpose:

Call `PutInt()` function in `synchconsole.cc` .

USERPROG/SYNCHCONSOLE.CC

PutInt()

Details:

- This function is called by `SysPrintInt()` in `userprog/ksyscall.h`
- `lock -> Acquire()`
  Since we do not want to be interrupted when executing the following operations, we call `lock -> Acquire()` so that all interrupts encountered by then will be temporarily locked into a queue. Note that when calling `lock -> Acquire()` , the kernel will lock after executing the current atomic instruction.
- do-while loop when `str[idx] != '\0'`

    - Call `PutChar()` to write char
    - Call `waitfor -> P()` to wait for a callback.
- `lock -> Release()`
  After finishing the operations above, we set the lock to be free, which means we can now handle interrupts.

Purpose:

Call `Putchar()` to print the string and lock the string we want to print in order to prevent from any changes of the string.

PutChar()

Details:

The structure is similar to `PutInt()` , while the only difference is that we don't need to call a do-while loop.

- lock -> Acquire()
- Call `PutChar()` to write char
- Call `waitfor -> P()` to wait for a call back.
- lock -> Release()

Purpose:

Write a character to the console display, waiting if necessary.

MACHINE/CONSOLE.CC

PutChar()

Details:

- This function is called by `PutInt()` in `userprog/synchconsole.cc`
- `ASSERT(putBusy == FALSE)`
  Make sure that there is no `PutChar()` operation in progress. Otherwise, call `Abort()` to abort the process.
- WriteFile
  Write a character to an open file with file descriptor

`writeFileNo` . Abort if the write fails.

- Set `putBusy` to be `TRUE`
  Prevent from being used by other threads.
- `kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt)`
  Schedule an interrupt to write integer.

Purpose:

Write a character to the simulated display, schedule an interrupt to occur in the future, and return.

MACHINE/INTERRUPT.CC

Interrupt::Schedule()

Details:

- This function is called by `PutChar()` in `machine/console.cc`
- `pending->Insert(toOccur)`
  Insert a new `PendingInterrupt` into the `pending` queue.

Purpose:

Arrange for the CPU to be interrupted when the simulated time reaches the specific time.

MACHINE/MIPSSIM.CC

Machine::Run()

Details:

- The details are the same as that in SC_Halt.
- Here, the `OneInstruction(instr)` has been done and return back to this function.

Purpose:

After finishing `PrintInt` , the process switches back to user mode.

MACHINE/INTERUPT.CC

Interrupt::OneTick()

Details:

- This function is call by `Machine::Run()` in `machine/mipssim.cc`
- Advance simulated time
  Update the `totalTicks` , then update `SystemTick` or `UserTick` depending on whether the status is `SystemMode` , respectively.
- Check any pending interrupts are now ready to fire
  Before enabling interrupts, turn off it first and check whether there are any interrupts. After finish checking, we re-enable interrupts.
- `if (yieldOnReturn){...}`
  Check whether the timer device handler asked for a context switch. If yes, then do so.

Purpose:

Advance simulated time and check if there are any pending interrupts to be called.

MACHINE/INTERUPT.CC

Interrupt::CheckIfDue()

Details:

- This function is called by `Interrupt::OneTick()` in `machine/interupt.cc`
- Return `FALSE` if either one of the following holds:
  - No pending interrupts are in the pending queue.
  - Not yet updated the Ticks.
- Otherwise return `TRUE` , which means we fire off some interrupts. The procedure includes:
  - Pick out the interrupt from the front of the pending list.
  - Call the interrupt handler and wait for the callback.

    - Repeat the same steps until there is no interrupt in pending.

Purpose:

Check if any interrupts are scheduled to occur, and if so, fire them off.

MACHINE/CONSOLE.CC

ConsoleOutput::CallBack()

Details:

- This function is called by `Interrupt::CheckIfDue()` in `machine/interupt.cc`
  After we initialize the interrupt in Schedule(), we pass the `ConsoleOutput` callback object. So the `CallBack()` links to `ConsoleOutput::CallBack()`
- `putBusy = FALSE`
  A PutChar operation is not in progress, so we set it to FALSE
- `callWhenDone -> CallBack()`
  Since we set `consoleOutput = new ConsoleOutput(outputFile, this)` when initializing a `SynchConsoleOutput` , this callback function links to `SynchConsoleOutput::CallBack()`

Purpose:

Simulator calls this when the next character can be output to the display.

USERPROG/SYNCHCONSOLE.CC

SynchConsoleOutput::CallBack()

Details:

- This function is called by `ConsoleOutput::CallBack()` in `machine/console.cc`
- `waitFor -> V()`

Same as lock -> Release() in PutInt(), V() is to run thread in queue if the queue is not empty.

Purpose:

Make the interrupt handler to handle interrupts and call back when the procedure is done.

Makefile

- `include Makefile.dep`
  Make sure the kernel on which NachOS is installed can execute our executable file.
- `CC` stands for C compiler.
- `LD` stands for linker.
- `AS` stands for assembly language compiler.
- `INCDIR` stands for the directory for the compiler.
- `CFLAGS` stands for the command to compile the C code.
- `PROGRAMS` contains some of the main instructions that can be compiled below.
- Main instructions include: `halt` , `add` , `LotOfAdd` , `shell` , `sort` , `segments` , `matmult` , `consoleIO_test1` , `consoleIO_test2` , `consoleIO_test3` , `fileIO_test1` , `fileIO_test2` , and `createFile` .
  - Since all the main instructions are similar, we take `halt` as an example to explain the details.
  - When we execute `make halt` in the terminal, the program starts to find whether there are object files `halt.o` and `start.o` for it to link them all, and then it makes an executable file `halt.coff` .
  - Finally, call `$(COFF2NOFF)` to translate `halt.coff` into `halt` so that the executable file can be executed on our based kernel (Here we use Linux).
  - If `halt.o` does not exist, it will use `halt.c` and the C compiler to generate the object code.
  - If `start.o` does not exist, it will use `start.S` , `syscall.h` , and the C compiler with `CFLAG` and `ASFLAG` to generate the object code.

- `make clean` removes all the `.o` , `.ii` , and `.coff` files.
- `make distclean` removes all executable files.
- Use `unknownhost` for error handling.

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

The main idea is to pass arguments via registers. Take `SC_PrintInt` from `ExceptionHandler()` in `userprog/exception.cc` as an example:

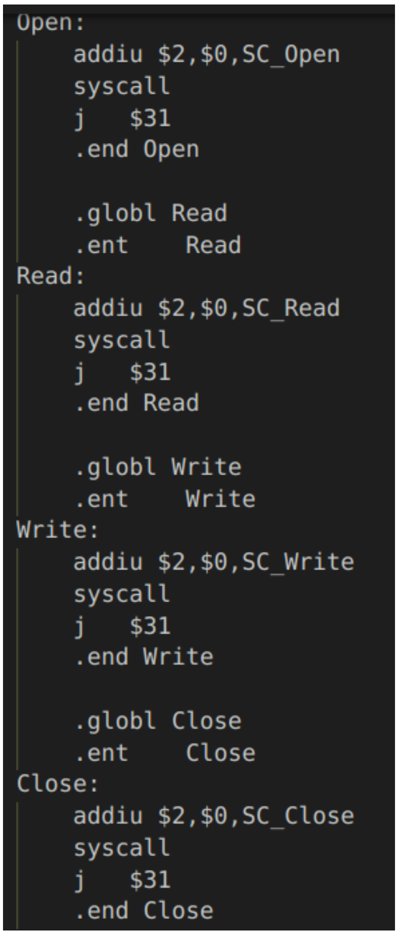
```
1   case SC_PrintInt:
2       DEBUG(dbgSys, "Print Int\n");
3       val=kernel->machine->ReadRegister(4);
4       DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kern
5       SysPrintInt(val);
6       DEBUG(dbgTraCode, "In ExceptionHandler(), return from SysPrintInt, " 
7       // Set Program Counter
8       kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegist
9       kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(P
10      kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegist
11      return;
12      ASSERTNOTREACHED();
13      break;
```

In Line 3 adn Line 5, after `val` gets the argument by `ReadRegister()` , we pass `val` to the system call.

Implementation

Modification to make it able to run

TEST/START.S

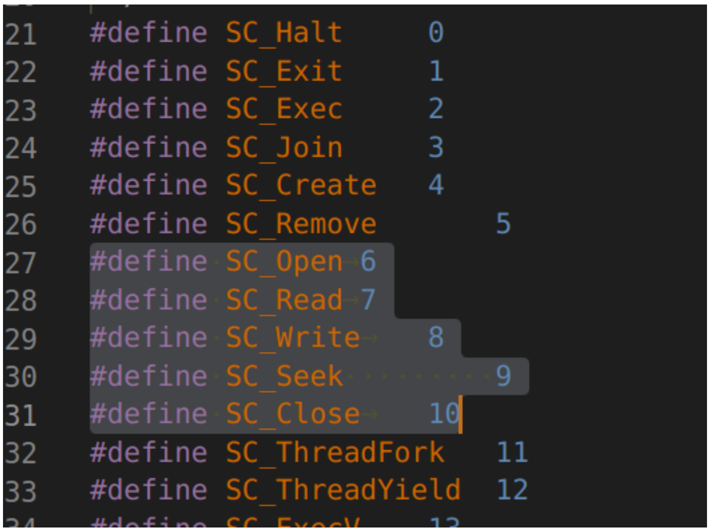


```
Open:
    addiu $2,$0,SC_Open
    syscall
    j   $31
    .end Open

    .globl Read
    .ent    Read
Read:
    addiu $2,$0,SC_Read
    syscall
    j   $31
    .end Read

    .globl Write
    .ent    Write
Write:
    addiu $2,$0,SC_Write
    syscall
    j   $31
    .end Write

    .globl Close
    .ent    Close
Close:
    addiu $2,$0,SC_Close
    syscall
    j   $31
    .end Close
```

Add this four part.

Refer to other System call to implement this part. Save the type of system call in Register[2](it's an int acutually), then do syscall. As the mechanism explained above, the ExceptionHandle handle the correct syscall. Afer the syscall finished, jump to the return address, end the instruction.

USERPROG/SYSCALL.H



```
#define SC_Halt      0
#define SC_Exit      1
#define SC_Exec      2
#define SC_Join      3
#define SC_Create    4
#define SC_Remove        5
#define SC_Open 6
#define SC_Read 7
#define SC_Write    8
#define SC_Seek         9
#define SC_Close    10
#define SC_ThreadFork    11
#define SC_ThreadYield   12
```

Uncomment these four line.

USERPROG/EXCEPTION.CC

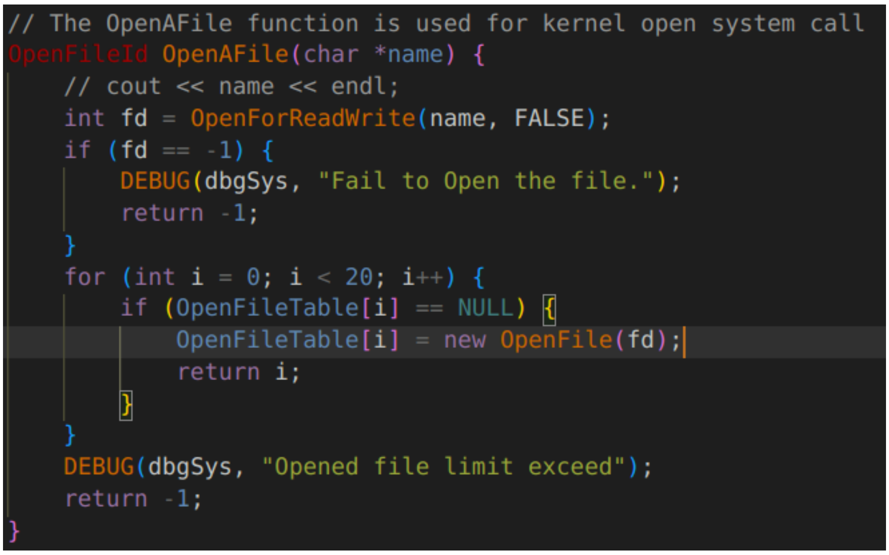
Add four case to handle: SC_Open, SC_Read, SC_Write, SC_Close. Read the arguments(saved in Register), pass them to the Sys*** in ksyscall.h. After the system call is done, get the status(as the return value), write back to Register[2]. Also, update the PCs.

USERPROG/KSYSCALL.H

Uncomment the SysOpen, add SysRead, SysWrite, SysClose.
Call the functions in filesys/filesys.h.

filesys/filesys.h

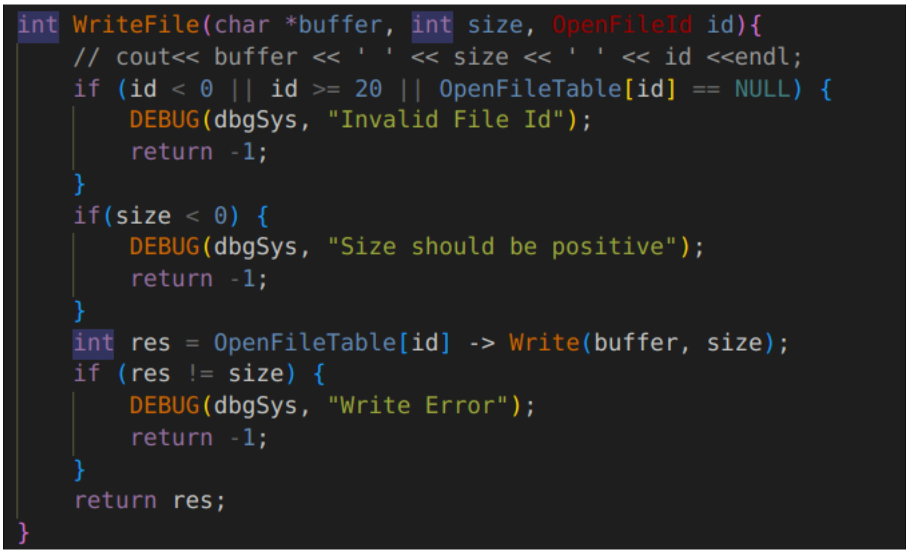
OPENFILEID OPEN(CHAR NAME)



```
// The OpenAFile function is used for kernel open system call
OpenFileId OpenAFile(char *name) {
    // cout << name << endl;
    int fd = OpenForReadWrite(name, FALSE);
    if (fd == -1) {
        DEBUG(dbgSys, "Fail to Open the file.");
        return -1;
    }
    for (int i = 0; i < 20; i++) {
        if (OpenFileTable[i] == NULL) {
            OpenFileTable[i] = new OpenFile(fd);
            return i;
        }
    }
    DEBUG(dbgSys, "Opened file limit exceed");
    return -1;
}
```

Firstly, we call OpenForReadWrite() in **sysdep.cc** (http://sysdep.cc) (open() in fcttl.h again). But this time, only the flag O_RDWR, which means that we don't create non-existing file nor adjust the file size.(Just Open). Get the filedescriptor returned, check if it succeed(not -1). If succeed, we will look up the OpenFileTable, try to find space to put the new-opened file in(it's actually a pointer, points to the filedescriptor), return the index. If we cannot find the space, means that we have already opened 20 files, return -1.

INT WRITE(CHAR BUFFER, INT SIZE, OPENFILEID ID)



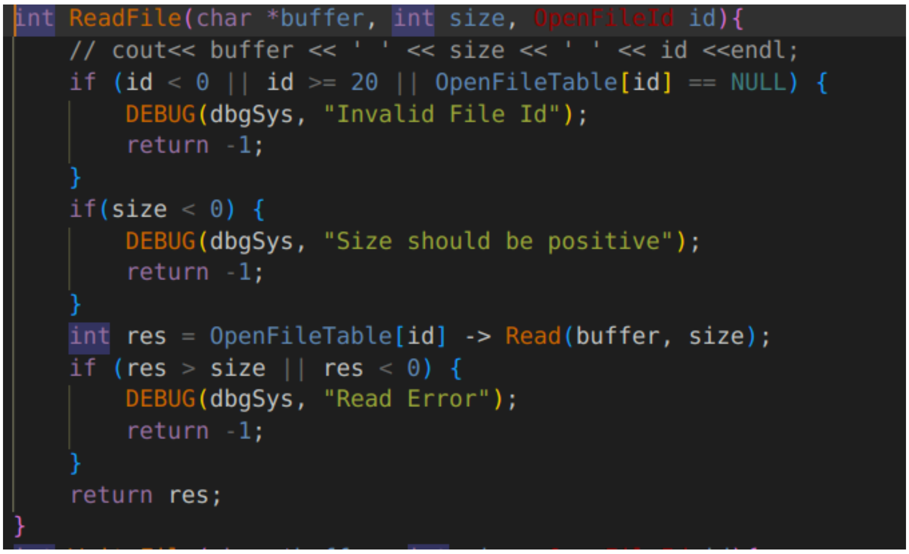
```
int WriteFile(char *buffer, int size, OpenFileId id){
    // cout<< buffer << ' ' << size << ' ' << id <<endl;
    if (id < 0 || id >= 20 || OpenFileTable[id] == NULL) {
        DEBUG(dbgSys, "Invalid File Id");
        return -1;
    }
    if(size < 0) {
        DEBUG(dbgSys, "Size should be positive");
        return -1;
    }
    int res = OpenFileTable[id] -> Write(buffer, size);
    if (res != size) {
        DEBUG(dbgSys, "Write Error");
        return -1;
    }
    return res;
}
```

Firstly, check whether the id is valid.(Out of bound, Not existing…).

Secondly, check whether the size to write is valid(size < 0 is strange).

Lastly, call the OpenFile::Write in openfile.h. It also call the Lseek() and WriteFile() directly(in sysdep.h, call the lseek(), write() in unistd.h, again). It ought to return the number of bytes writed, if not equal to size, return -1.

int Read(char buffer, int size, OpenFileId id)



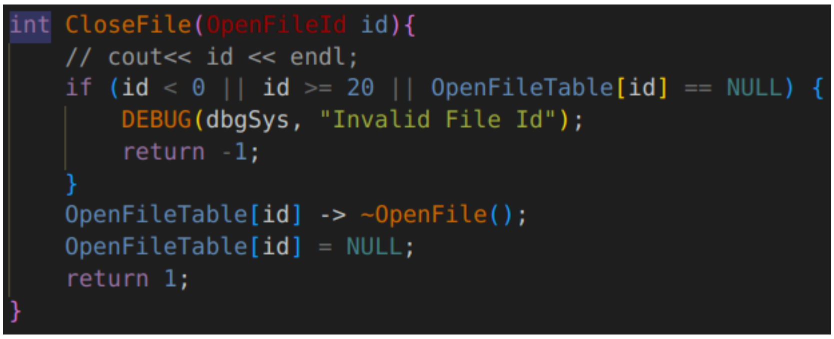
```
int ReadFile(char *buffer, int size, OpenFileId id){
    // cout<< buffer << ' ' << size << ' ' << id <<endl;
    if (id < 0 || id >= 20 || OpenFileTable[id] == NULL) {
        DEBUG(dbgSys, "Invalid File Id");
        return -1;
    }
    if(size < 0) {
        DEBUG(dbgSys, "Size should be positive");
        return -1;
    }
    int res = OpenFileTable[id] -> Read(buffer, size);
    if (res > size || res < 0) {
        DEBUG(dbgSys, "Read Error");
        return -1;
    }
    return res;
}
```

Firstly, check whether the id is valid.(Out of bound, Not existing…).

Secondly, check whether the size to read is valid(size < 0 is strange).

Lastly, call the OpenFile::Read in openfile.h. It also call the Lseek() and ReadPartial() directly(in sysdep.h, call the lseek(), write() in unistd.h, again). It ought to return the number of bytes writed. We want to do something different here. Sometimes, we might encouter EOF(or other things) here, so, we want that even the read bytes < size, it still can be handled.

INT CLOSE(OPENFILEID ID)



```
int CloseFile(OpenFileId id){
    // cout<< id << endl;
    if (id < 0 || id >= 20 || OpenFileTable[id] == NULL) {
        DEBUG(dbgSys, "Invalid File Id");
        return -1;
    }
    OpenFileTable[id] -> ~OpenFile();
    OpenFileTable[id] = NULL;
    return 1;
}
```

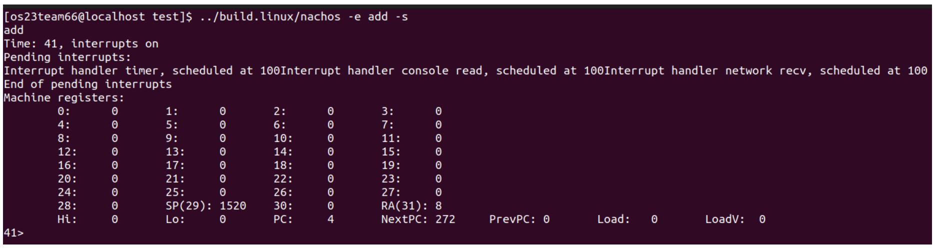
Firstly, check whether the id is valid.(Out of bound, Not existing…).
If it's valid, call ~OpenFile() to destruct it(Call close() in unistd.h. It will actually close the filedescriptor, not killing the file itself). Set the OpenFileTable[id] to NULL(so that it can accommodate future open file). Return 1, means succeed.
To be honest, most of the Debug message are useless, since there are lots of ASSERT in ***sysdep.cc*** *(http://sysdep.cc).*
However, let's say it's a good habit:D

Difficulties

- 徐竣霆：
  一開始以為全部的code都要trace過，要把所有的流程、細節全部搞懂，所以壓力山大。加上清大一直停電、Server一直掛掉(Resolved)，又沒辦法本地寫作業(環境build不起來)，而且我的VPN每隔5分鐘就會自己斷掉(可能是我的宿網問題，還沒解決QAQ)，一開始真的有點燥。後來問過助教後，得知只需要把流程圖的部份搞懂，之後的部份交給之後的作業，就比較有方向一點了，把流程圖的部份搞懂，加上額外trace一點延伸出去的東西，就可以把implementation的部份完成了。隨著清大不再停電之後，一切都慢慢地迎刃而解了。

- 江承紘：
  一開始不知道該從何切入，習慣性去找 main function 在哪裡。可是其實 trace code 的過程不用從頭開始，只要從 Spec 指定的地方開始 trace 就好了。上手之後後面的部分都比較清楚自己應該做些什麼。

Feedback

- 徐竣霆：
  我發現了如果指令不小心把-d下成-s，他會出現這個畫面：
  

```
[os23team66@localhost test]$ ../build.linux/nachos -e add -s
add
Time: 41, interrupts on
Pending interrupts:
Interrupt handler timer, scheduled at 100Interrupt handler console read, scheduled at 100Interrupt handler network recv, scheduled at 100
End of pending interrupts
Machine registers:
        0:      0       1:      0       2:      0       3:      0
        4:      0       5:      0       6:      0       7:      0
        8:      0       9:      0       10:     0       11:     0
        12:     0       13:     0       14:     0       15:     0
        16:     0       17:     0       18:     0       19:     0
        20:     0       21:     0       22:     0       23:     0
        24:     0       25:     0       26:     0       27:     0
        28:     0       SP(29): 1520    30:     0       RA(31): 8
        Hi:     0       Lo:     0       PC:     4       NextPC: 272     PrevPC: 0       Load:   0       LoadV:  0
41>
```

  好像是顯示當前各個Register的狀態。但我也不知道為什麼就是了。

不太知道報告要寫得多細，不太清楚哪些是助教想看的重點，所以剛開始trace的時候把report寫得很詳細。所以如果有哪邊覺得是廢話的地方，我很抱歉。後來想說應該有蠻多部份是後面的作業，就想說大概寫一下就好(~~減輕助教負擔~~)。所以如果助教覺得有哪些寫得不清楚(或者太detailed)，希望能反應給我們知道一下，讓我們以後能交出讀起來更加舒服的report。

- 江承紘：
  有些 code 的排版不太舒服。想像上要對齊的 code 沒有對齊，必須要自己按 tab 排版才能增加可讀性。不過自己排版過後就都看得懂了，內容本質上沒有問題