CS342301: Operating System

MP1: System Call

Team31

111060013 劉祐廷 111060019 黃子恩

	Contribution
劉祐廷	1. Trace code in Part I and write the report for it. 2. Implement code in part II 3. Discuss and complete the entire report.
黃子恩	1. Trace code in Part I and write the report for it. 2. Write the report for part II 3. Discuss and complete the entire report.

1. Trace Code

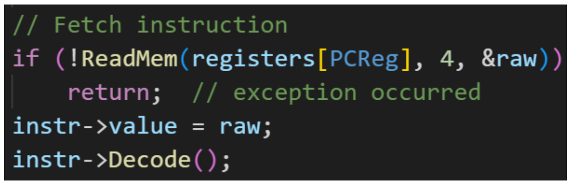
I. SC_Halt

To run a user-level program, turn into usermode. Execute instructions one by one calling Machine::OneInstrution().

A. Machine::Run()

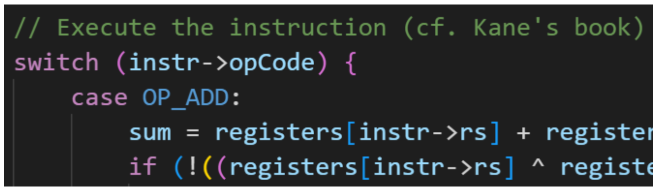
This function is responsible for simulating the process of the CPU executing a program in NachOS. It first sets the mode to user mode by calling kernel->interrupt->setStatus(UserMode);. Then, it enters an infinite loop, for(;;){}, which simulates the CPU's continuous fetching of instructions during execution. Within the loop, the OneInstruction() function is called to execute an instruction. After each execution of OneInstruction(), the OneTick() function is invoked to check for and handle any interrupts that may have occurred.

B. Machine::OneInstruction()



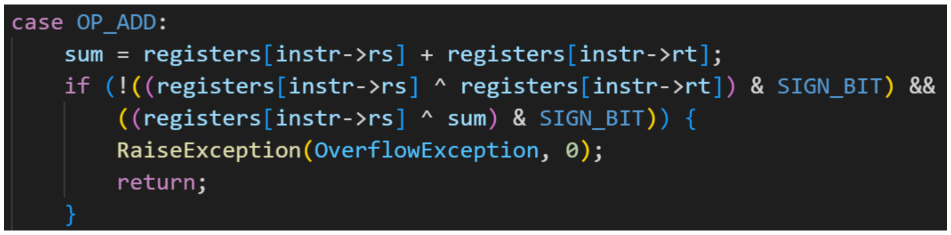
```
// Fetch instruction
if (!ReadMem(registers[PCReg], 4, &raw))
    return;  // exception occurred
instr->value = raw;
instr->Decode();
```

First, it fetches the instruction from registers[PCReg]. If it cannot fetch an instruction, it returns; if an instruction is fetched, it proceeds to decode it.



```
// Execute the instruction (cf. Kane's book)
switch (instr->opCode) {
    case OP_ADD:
        sum = registers[instr->rs] + register
        if (!((registers[instr->rs] ^ registe
```

Next, it uses the opCode to determine the action represented by the instruction.



```
case OP_ADD:
    sum = registers[instr->rs] + registers[instr->rt];
    if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
        ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
        RaiseException(OverflowException, 0);
        return;
    }
```

Using the action OP_ADD as an example, if an exception occurs, the error code is passed as a parameter to RaiseException() for handling. If no exception occurs, the respective operations are performed as usual.

C. Machine::RaiseException()

```
registers[BadVAddrReg] = badVAddr;
```

The problematic virtual address is stored in the register designated for holding the bad virtual address.

```
kernel->interrupt->setStatus(SystemMode);
ExceptionHandler(which);  // interrupts a
kernel->interrupt->setStatus(UserMode);
```

To handle the exception, turn into kernel mode. Call the ExceptionHandeler(), then turn into usermode again.

D. ExceptionHandler()

```
    .globl Halt
    .ent    Halt
Halt:
    addiu $2,$0,SC_Halt
    syscall
    j   $31
    .end Halt
```

From start.S, it can be observed that the codes for these exceptions are all stored in the r2 register.

```
int type = kernel->machine->ReadRegister(2);
```

Therefore, the system call code can be obtained by reading the value of r2, as the system call code is stored in r2.

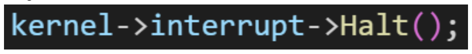
```
switch (which) {
    case SyscallException:
```

First, check whether the system call code exists.

```
switch (type) {
    case SC_Halt:
        DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
        SysHalt();
        cout << "in exception\n";
        ASSERTNOTREACHED();
        break;
```

Next, perform different actions based on the system call code. Since the system call code passed in is SC_Halt, the program will execute this section of code to call SysHalt().

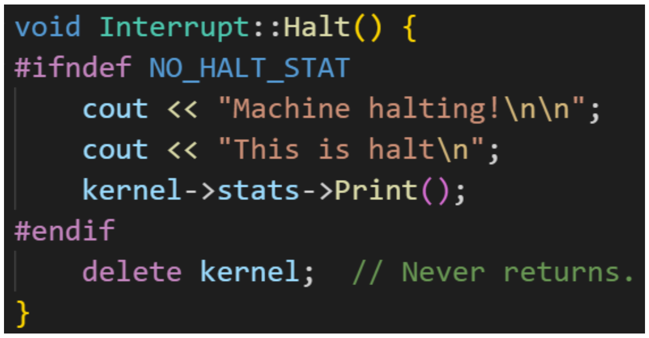
E. SysHalt()



```
kernel->interrupt->Halt();
```

Call the Halt() function defined in the kernel's interrupt operations.

F. Interrupt::Halt()



```
void Interrupt::Halt() {
#ifndef NO_HALT_STAT
    cout << "Machine halting!\n\n";
    cout << "This is halt\n";
    kernel->stats->Print();
#endif
    delete kernel;  // Never returns.
}
```

Deconstruct the entire kernel, releasing resources and stopping the entire system.

II. SC_Create

A. ExceptionHandler()

```
val = kernel->machine->ReadRegister(4);
```

Read the address of the file name string to be added from r4 (which is usually arg1).

```
{
    char *filename = &(kernel->machine->mainMemory[val]);
    // cout << filename << endl;
    status = SysCreate(filename);
    kernel->machine->WriteRegister(2, (int)status);
}
```

Extract the filename and pass it to SysCreate() to create the file, then write the return value to r2.

```
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);
```

Update the PC to prepare for executing the next instruction.

B. SysCreate()

```
int SysCreate(char *filename) {
    // return value
    // 1: success
    // 0: failed
    return kernel->fileSystem->Create(filename);
}
```

Call the Create() function in the fileSystem and pass the filename to it.

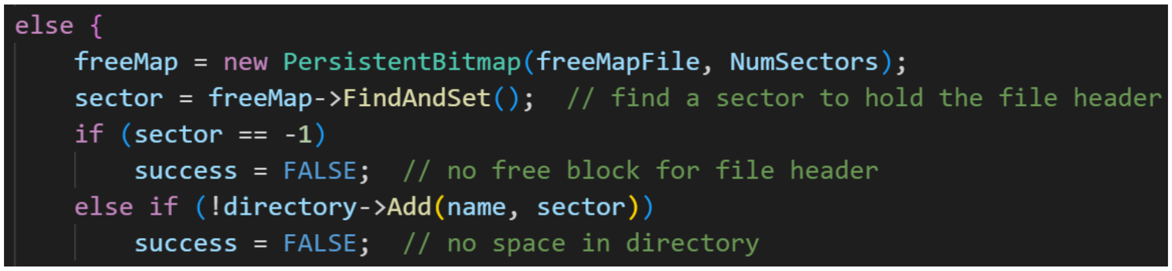
C. FileSystem::Create()

```
directory = new Directory(NumDirEntries);
directory->FetchFrom(directoryFile);
```

Obtain the corresponding directory.

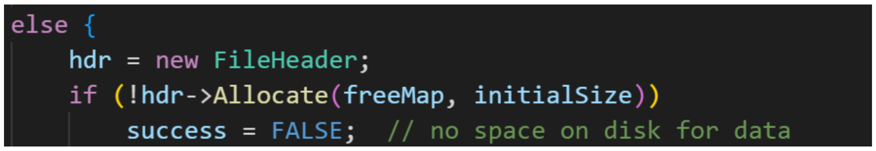
```
if (directory->Find(name) != -1)
    success = FALSE;  // file is already in directory
```

Check if the file exists. If it does, then return false.



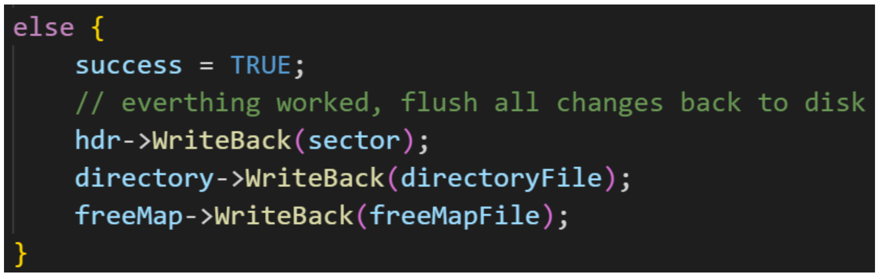
```
else {
    freeMap = new PersistentBitmap(freeMapFile, NumSectors);
    sector = freeMap->FindAndSet();  // find a sector to hold the file header
    if (sector == -1)
        success = FALSE;  // no free block for file header
    else if (!directory->Add(name, sector))
        success = FALSE;  // no space in directory
```

Check if there is enough space to create the file. If not, then return false.



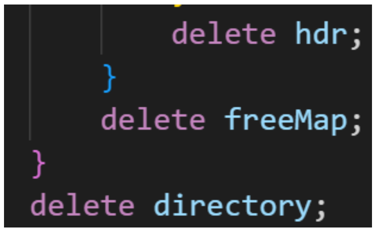
```
else {
    hdr = new FileHeader;
    if (!hdr->Allocate(freeMap, initialSize))
        success = FALSE;  // no space on disk for data
```

Try to allocate space for the file. If there is not enough space, then return false.



```
else {
    success = TRUE;
    // everthing worked, flush all changes back to disk
    hdr->WriteBack(sector);
    directory->WriteBack(directoryFile);
    freeMap->WriteBack(freeMapFile);
}
```

Successfully create the file and write the modified data back to the disk.

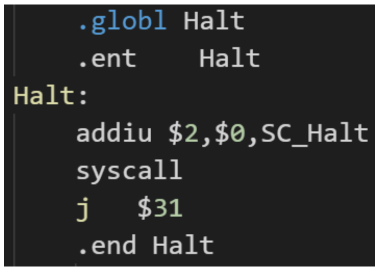


```
            delete hdr;
        }
        delete freeMap;
    }
    delete directory;
```

Deconstruct these pointers to release memory resources.

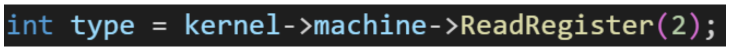
III.SC_PrintInt

A. ExceptionHandler()



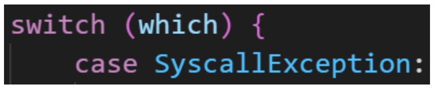
```
    .globl Halt
    .ent    Halt
Halt:
    addiu $2,$0,SC_Halt
    syscall
    j   $31
    .end Halt
```

From start.S, it can be observed that the codes for these exceptions are all stored in the memory location referenced by r2.



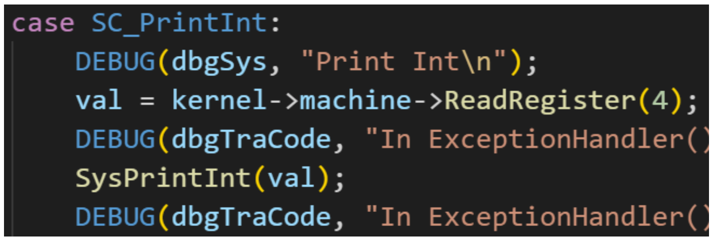
```
int type = kernel->machine->ReadRegister(2);
```

Therefore, the system call code can be obtained by reading the value of r2, as the system call code is stored in r2.



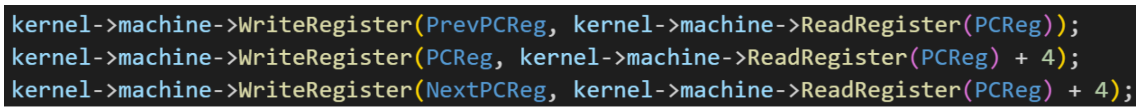
```
switch (which) {
    case SyscallException:
```

First, check whether this system call code exists.



```
case SC_PrintInt:
    DEBUG(dbgSys, "Print Int\n");
    val = kernel->machine->ReadRegister(4);
    DEBUG(dbgTraCode, "In ExceptionHandler()
    SysPrintInt(val);
    DEBUG(dbgTraCode, "In ExceptionHandler()
```

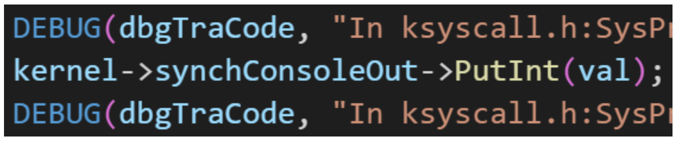
Next, perform different actions based on the system call code. Since the passed system call code is SC_PrintInt, the program will execute this section of code. First, it will read the value to be output from r4 (which is arg1) and store it in val. Then, it will pass val as a parameter to SysPrintInt().



```
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);
```

Update the PC to prepare for executing the next instruction.

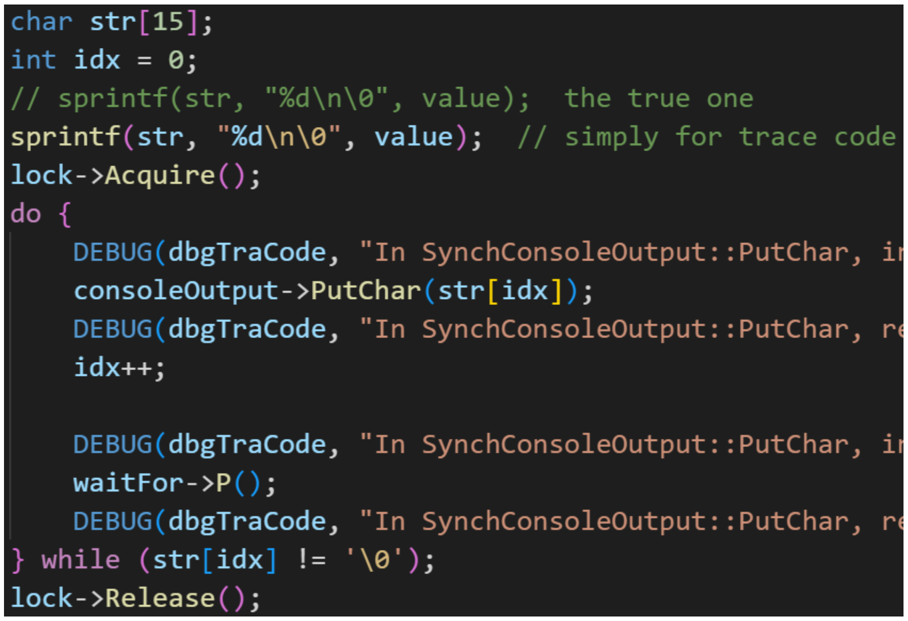
B. SysPrintInt()



```
DEBUG(dbgTraCode, "In ksyscall.h:SysP
kernel->synchConsoleOut->PutInt(val);
DEBUG(dbgTraCode, "In ksyscall.h:SysP
```

Pass the value to be output (val) to the PutInt() function, which is already defined in the kernel's synchConsoleOut.

C. SynchConsoleOutput::PutInt()



```
char str[15];
int idx = 0;
// sprintf(str, "%d\n\0", value);  the true one
sprintf(str, "%d\n\0", value);  // simply for trace code
lock->Acquire();
do {
    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, i
    consoleOutput->PutChar(str[idx]);
    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, r
    idx++;

    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, i
    waitFor->P();
    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, r
} while (str[idx] != '\0');
lock->Release();
```

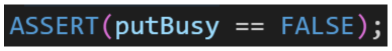
Convert the number to a string and use a loop to pass each character one by one to PutChar().

- lock->Acquire();
  This line of code acquires a lock before performing operations on shared resources, ensuring that the current process has exclusive access to the resource before entering the critical section. This prevents other processes from simultaneously accessing consoleOutput, which could lead to mixed output data. When the lock is held by one process, any other processes attempting to acquire the lock will be blocked until the lock is released.
- lock->Release();
  Once this operation is complete, the lock is released, allowing other processes to enter the critical section. This ensures that after a complete output operation has finished, other processes can safely execute their own output operations.

D. SynchConsoleOutput::PutChar()

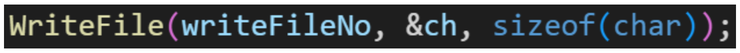
Compared to SynchConsoleOutput::PutInt(), it simply omits the process of converting the number to a string; the remaining parts are largely the same.

E. ConsoleOutput::PutChar()



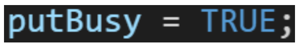
```
ASSERT(putBusy == FALSE);
```

putBusy indicates whether the console output is busy. When it is FALSE, it means the console is idle, and a new output can begin. If it is TRUE, it indicates that an output operation is in progress, and this call should be blocked to prevent multiple characters from being output simultaneously.



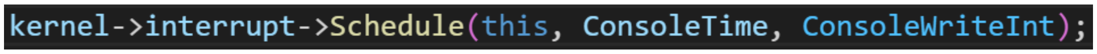
```
WriteFile(writeFileNo, &ch, sizeof(char));
```

Simulate outputting ch to the console output.



```
putBusy = TRUE;
```

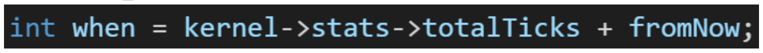
Set putBusy to TRUE to indicate that the console output is busy, preventing multiple characters from being output simultaneously.



```
kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
```

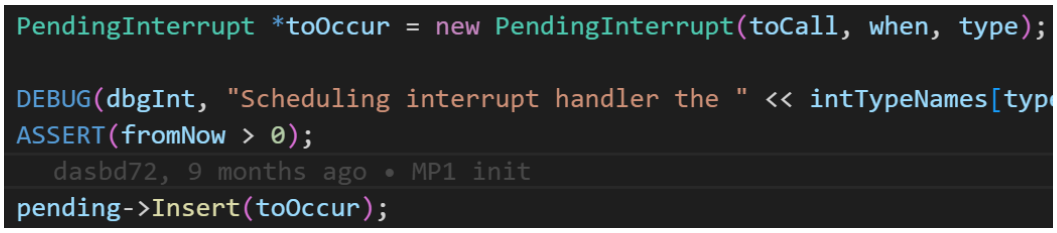
Call Schedule() to raise an interrupt after the output is completed, indicating that the output operation is finished.

F. Interrupt::Schedule()



```
int when = kernel->stats->totalTicks + fromNow;
```

Calculate the time at which this interrupt should be triggered.



```
PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);

DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[typ
ASSERT(fromNow > 0);
   dasbd72, 9 months ago • MP1 init
pending->Insert(toOccur);
```

Construct a PendingInterrupt and check if fromNow is greater than 0 (indicating an event that will occur in the future). If it is, add this PendingInterrupt to the sorted list, waiting for it to be triggered.

G. Machine::Run()

```
OneInstruction(instr);
DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  "
                    << "== Tick " << kernel->stats->totalTicks << " ==");

DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
                    << "== Tick " << kernel->stats->totalTicks << " ==");
kernel->interrupt->OneTick();
DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
                    << "== Tick " << kernel->stats->totalTicks << " ==");
```

Run() will call OneTick() after each invocation of OneInstruction(). The purpose is to check for any interrupts after executing each instruction and to update the time ticks.

H. Machine::OneTick()

```
MachineStatus oldStatus = status;
```

Save the current state for data recovery purposes.

```
Statistics *stats = kernel->stats;
```

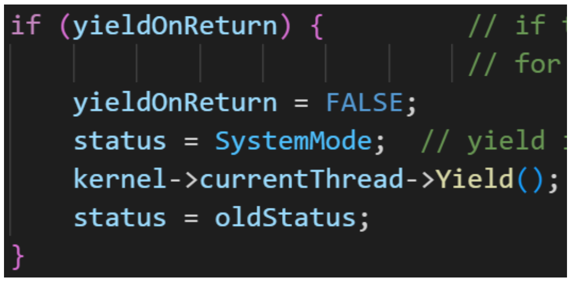
Obtain the statistics metrics to update the ticks.

```
// advance simulated time
if (status == SystemMode) {
    stats->totalTicks += SystemTick;
    stats->systemTicks += SystemTick;
} else {
    stats->totalTicks += UserTick;
    stats->userTicks += UserTick;
}
```

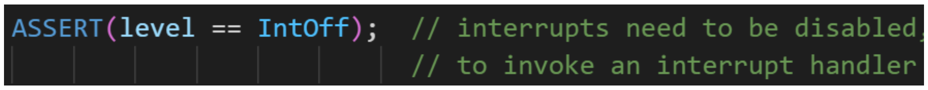
Update the ticks according to the current mode.

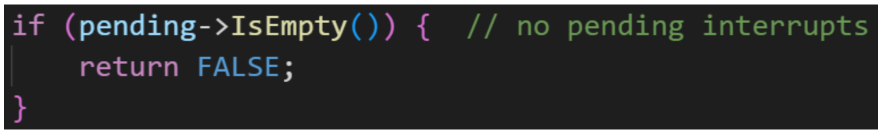
```
// check any pending interrupts are now ready to fire
ChangeLevel(IntOn, IntOff);  // first, turn off interrupts
                             // (interrupt handlers run with
                             // interrupts disabled)
CheckIfDue(FALSE);           // check for pending interrupts
ChangeLevel(IntOff, IntOn);  // re-enable interrupts
```

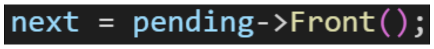
Call ChangeLevel() to disable interrupts, allowing the interrupt handler to execute. Then, call CheckIfDue() to check if any PendingInterrupts are due and need to be raised. Finally, call ChangeLevel() again to re-enable interrupts.

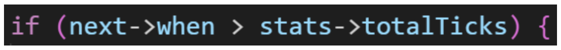

```
if (yieldOnReturn) {          // if
                              // for
    yieldOnReturn = FALSE;
    status = SystemMode;  // yield
    kernel->currentThread->Yield();
    status = oldStatus;
}
```
Check if the timer has requested a context switch. If so, switch directly to the new context. Since yield is managed by the kernel, it is necessary to first switch to system mode, allowing the CPU to change tasks. After the switch is complete, restore the status to continue execution.

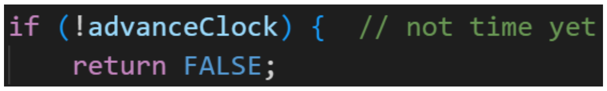
I. Interrupt::CheckIfDue()

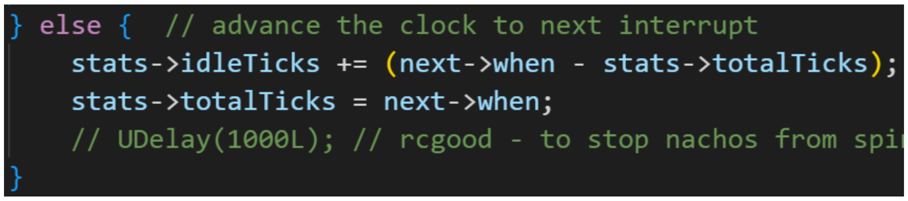

```
ASSERT(level == IntOff);  // interrupts need to be disabled
                          // to invoke an interrupt handler
```
Ensure that interrupts are disabled.

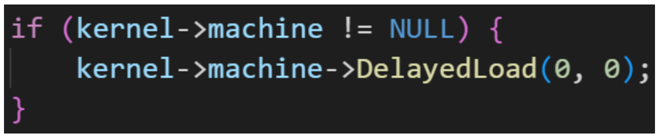

```
if (pending->IsEmpty()) {  // no pending interrupts
    return FALSE;
}
```
If there are no interrupts pending, simply return FALSE.


```
next = pending->Front();
```
Read the next interrupt that is scheduled to be executed.

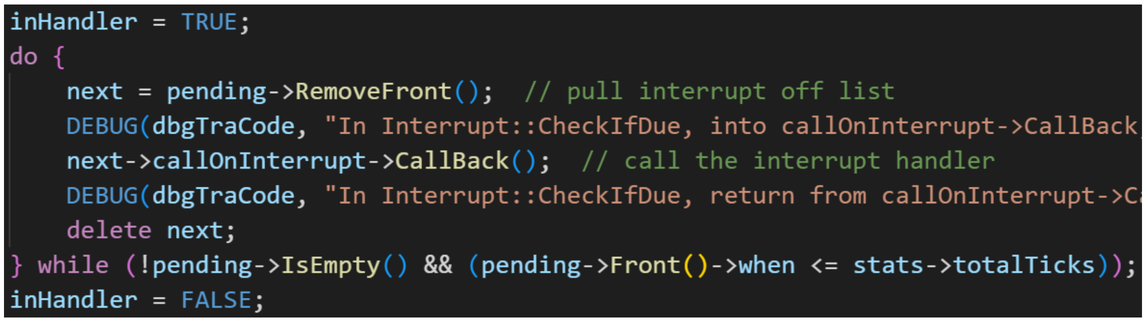

```
if (next->when > stats->totalTicks) {
```
If it is not yet time for the interrupt to occur.


```
if (!advanceClock) {  // not time yet
    return FALSE;
```
If it is not yet time to trigger the interrupt, return FALSE.


```
} else {  // advance the clock to next interrupt
    stats->idleTicks += (next->when - stats->totalTicks);
    stats->totalTicks = next->when;
    // UDelay(1000L); // rcgood - to stop nachos from spi
}
```
When advanceClock is true, it indicates that there are no other processes ready at the moment, allowing the system to advance the time to this interrupt. This will also increase the idle ticks, so idleTicks should be updated as well.

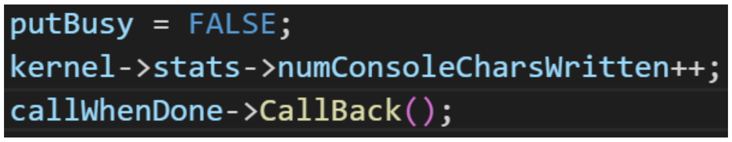

```
if (kernel->machine != NULL) {
    kernel->machine->DelayedLoad(0, 0);
}
```

If a machine simulator exists, a delay load must be executed, which is a step between the kernel and the hardware.


```
inHandler = TRUE;
do {
    next = pending->RemoveFront();  // pull interrupt off list
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack
    next->callOnInterrupt->CallBack();  // call the interrupt handler
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->C
    delete next;
} while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
inHandler = FALSE;
```

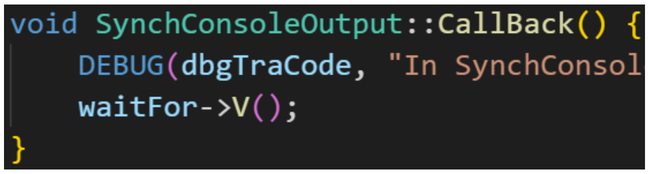
Set inHandler to TRUE, indicating that an interrupt is being handled. Use a while loop to check if any interrupts in the pending list have expired; if they have, remove them and call the appropriate CallBack() from callOnInterrupt based on the next interrupt type to handle the interrupt. Finally, after the interrupt has been processed, set inHandler back to FALSE, indicating that the interrupt handling is complete.

J. ConsoleOutput::CallBack()


```
putBusy = FALSE;
kernel->stats->numConsoleCharsWritten++;
callWhenDone->CallBack();
```

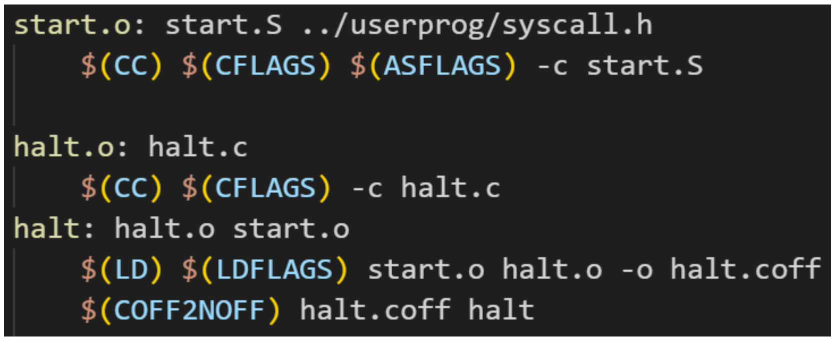
When a character output is completed, putBusy will be set to FALSE, indicating that the next character can now be output. Update the count of successfully output characters, then call the CallBack() from callWhenDone to notify the system that the next character can be output.

K. SynchConsoleOutput::CallBack()


```
void SynchConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In SynchConsol
    waitFor->V();
}
```

After the output is complete, use waitfor's V() to notify the system to release the lock, allowing the next process to output. Compared to ConsoleOutput::CallBack(), this includes an additional locking mechanism to ensure the correctness of output in a multiprocess environment.

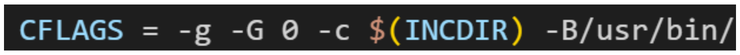
IV. Makefiles



```
start.o: start.S ../userprog/syscall.h
	$(CC) $(CFLAGS) $(ASFLAGS) -c start.S

halt.o: halt.c
	$(CC) $(CFLAGS) -c halt.c
halt: halt.o start.o
	$(LD) $(LDFLAGS) start.o halt.o -o halt.coff
	$(COFF2NOFF) halt.coff halt
```

Using halt as an example, when you type make halt in the terminal, it will jump to the halt label and check the files following the colon. It will then recursively go through each file, ensuring that they are all properly processed.

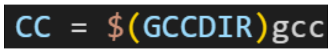


```
CFLAGS = -g -G 0 -c $(INCDIR) -B/usr/bin/
```

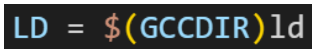
…

CFLAGS is a variable that stores a set of directives to inform the compiler which options to use during the compilation process.

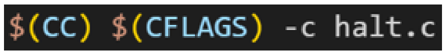
- -g：Let the compiler generate debugging information.
- -G 0：-G is typically used in MIPS architecture compilers, where 0 indicates the size of variables placed in the global pointer area (GPA), and 0 signifies that no optimization is performed.
- -c：Tell the compiler to compile only the source files without performing the linking process, which will produce .o files.
- $(INCDIR)：This is a variable that stores the path to the header files. When the compiler encounters #include, it will search for the header files in this path.
- -B/usr/bin/…：-B makes the following paths binary, allowing the compiler to search for tools like the linker in those paths.



```
CC = $(GCCDIR)gcc
```

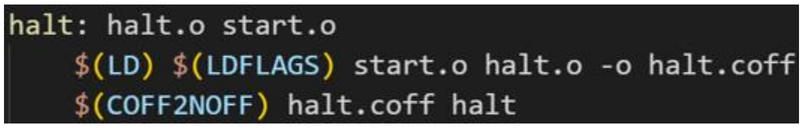


```
LD = $(GCCDIR)ld
```



```
$(CC) $(CFLAGS) -c halt.c
```

The CC variable represents the path to the gcc program (compiler). The LD variable represents the path to the ld program (linker).This line is responsible for creating halt.o from halt.c.

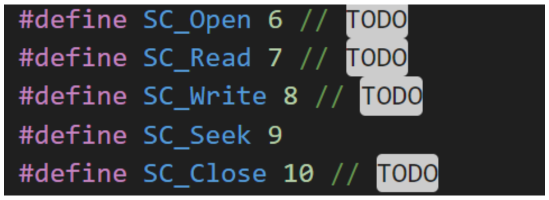


```
halt: halt.o start.o
	$(LD) $(LDFLAGS) start.o halt.o -o halt.coff
	$(COFF2NOFF) halt.coff halt
```

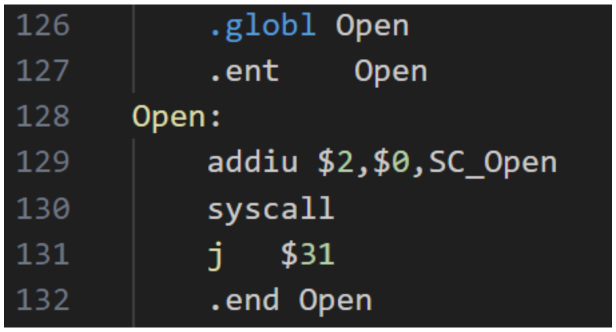
This segment of code is responsible for linking halt.o and start.o together to create halt.coff. The .coff file is executable on MIPS architecture, but since the Linux architecture is x86, it needs to be converted to .noff to become a file that can be executed on the MIPS simulator.

2. Implementation of I/O System Calls in NachOS

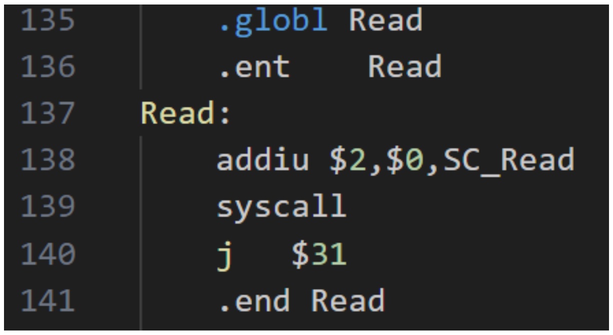
Before calling these functions, we have to define their system call code and define them in the start.s using MIPS assembly language. Put the system call code into register 2.



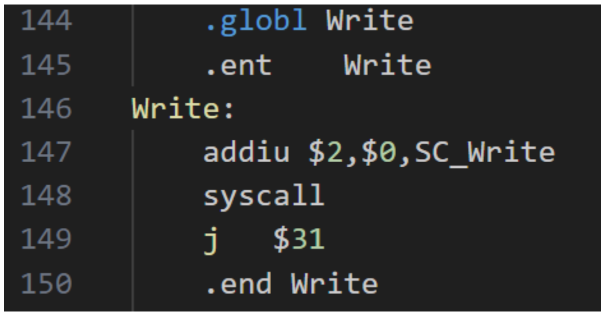
```
#define SC_Open 6 // TODO
#define SC_Read 7 // TODO
#define SC_Write 8 // TODO
#define SC_Seek 9
#define SC_Close 10 // TODO
```



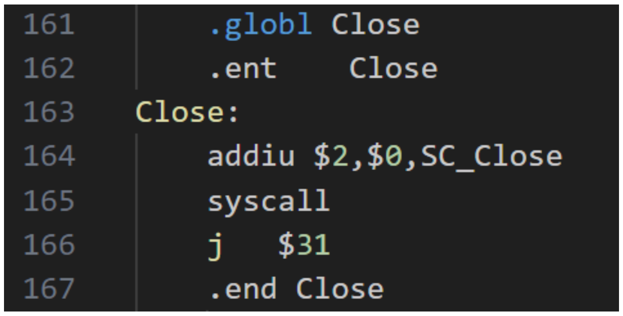
```
	.globl Open
	.ent	Open
Open:
	addiu $2,$0,SC_Open
	syscall
	j	$31
	.end Open
```



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

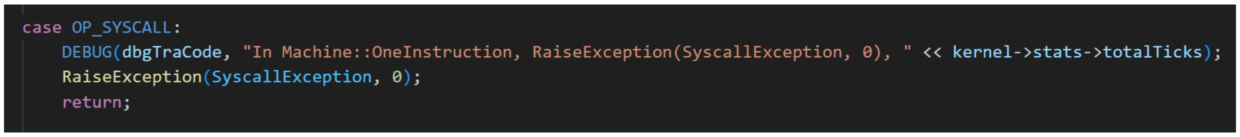


```
	.globl Write
	.ent	Write
Write:
	addiu $2,$0,SC_Write
	syscall
	j	$31
	.end Write
```



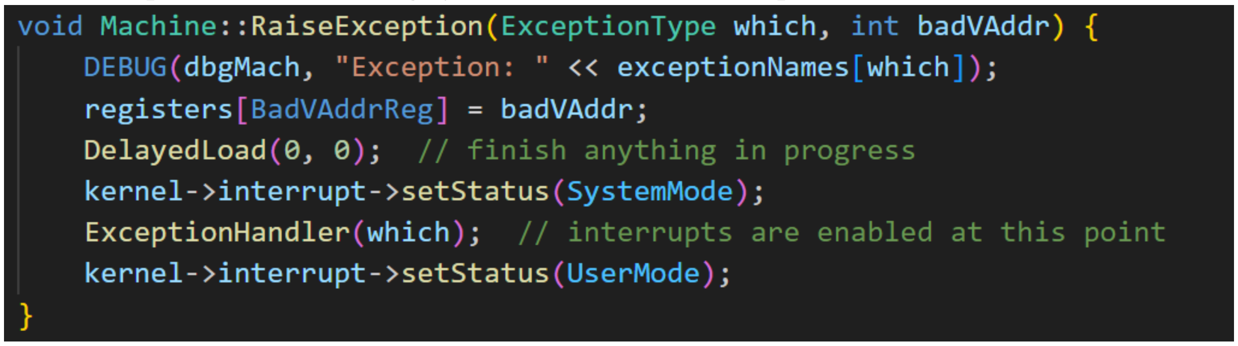
```
	.globl Close
	.ent	Close
Close:
	addiu $2,$0,SC_Close
	syscall
	j	$31
	.end Close
```

Their path of these function are similar to Sc_create.



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

Raise Exception due to invoking system call, call RaiseException().



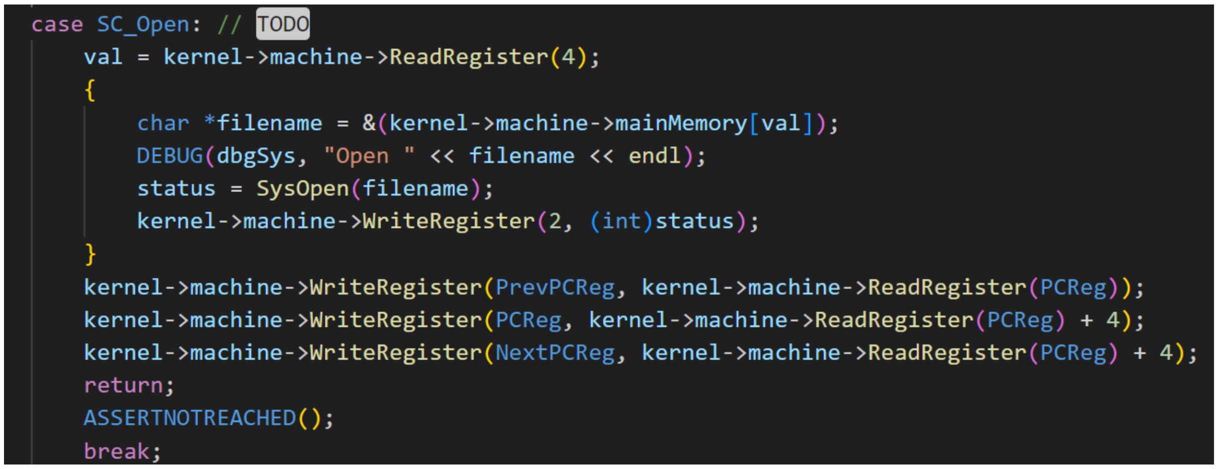
```
void Machine::RaiseException(ExceptionType which, int badVAddr) {
    DEBUG(dbgMach, "Exception: " << exceptionNames[which]);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);  // finish anything in progress
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);  // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
}
```

Change into kernel mode to handle the exception, call ExceptionHandler(). Turn back to user mode after finishing handling the exception.

I. OpenFileId Open(char *name);

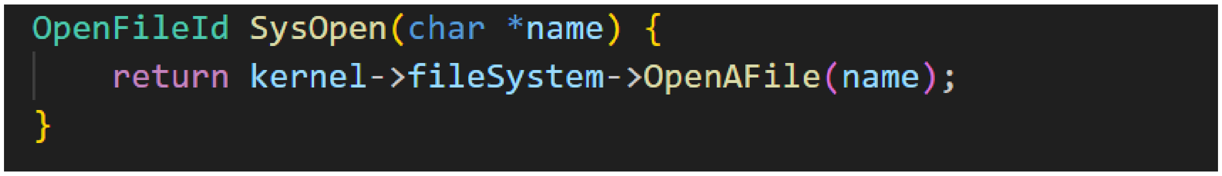
Open a file with the name and return its corresponding OpenFileId. Return -1 if it fails to open the file.

Path: Run() -> OneInstruction() -> RaiseException() -> EXceptionHandler() -> SysOpen() -> FilsSystem :: OpenAFile()



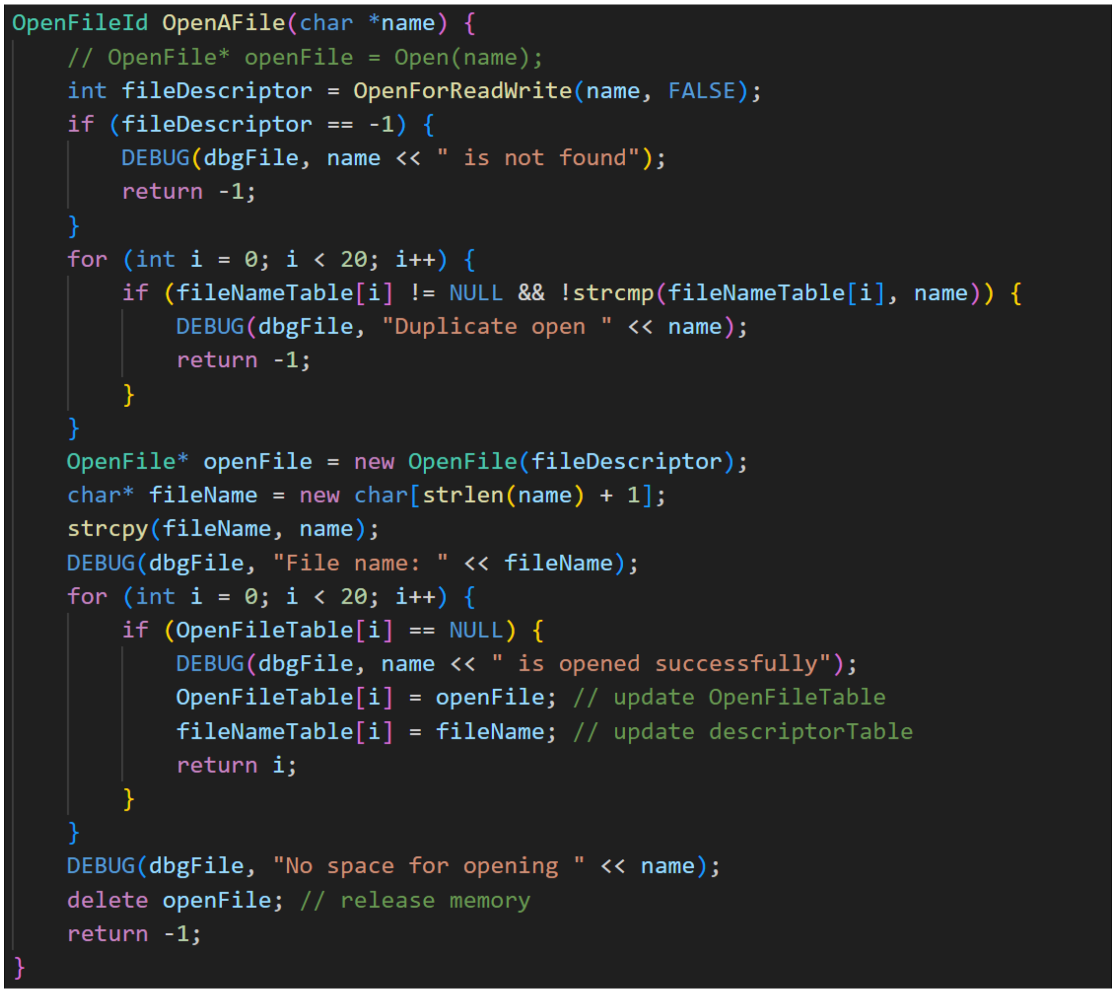
```
case SC_Open: // TODO
    val = kernel->machine->ReadRegister(4);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        DEBUG(dbgSys, "Open " << filename << endl);
        status = SysOpen(filename);
        kernel->machine->WriteRegister(2, (int)status);
    }
    kernel->machine->WriteRegister(PrevPCReg, 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;
```

In ExceptionHandeler(), read the system call code from R2 to deal with the exception regard to the ExceptionType. In this case, the system code is SC_Open. Read the address of filename from R4, call SysOpen() passing the filename. Then write the return value into R2.



```
OpenFileId SysOpen(char *name) {
    return kernel->fileSystem->OpenAFile(name);
}
```

Call OpenAFile() defined in FileSystem.

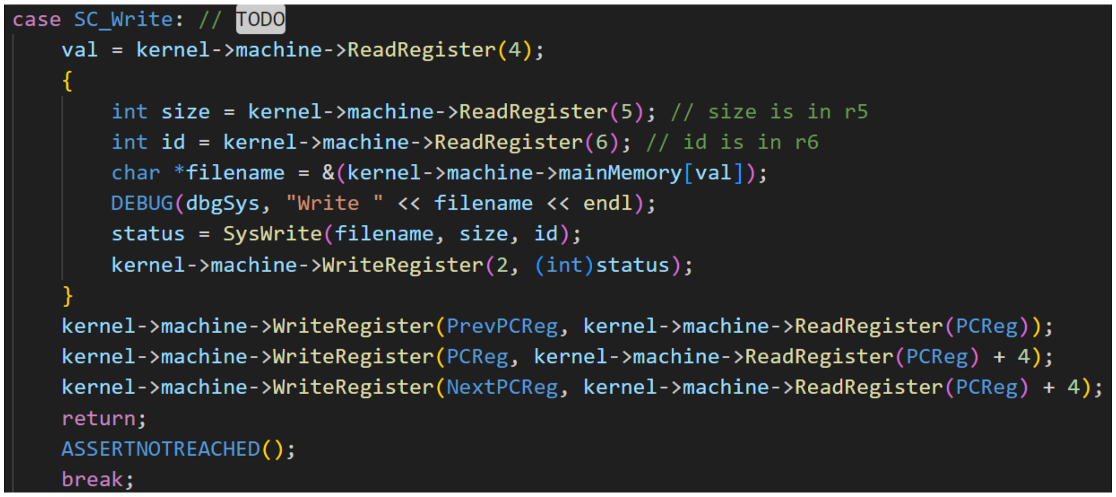


```
OpenFileId OpenAFile(char *name) {
    // OpenFile* openFile = Open(name);
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    if (fileDescriptor == -1) {
        DEBUG(dbgFile, name << " is not found");
        return -1;
    }
    for (int i = 0; i < 20; i++) {
        if (fileNameTable[i] != NULL && !strcmp(fileNameTable[i], name)) {
            DEBUG(dbgFile, "Duplicate open " << name);
            return -1;
        }
    }
    OpenFile* openFile = new OpenFile(fileDescriptor);
    char* fileName = new char[strlen(name) + 1];
    strcpy(fileName, name);
    DEBUG(dbgFile, "File name: " << fileName);
    for (int i = 0; i < 20; i++) {
        if (OpenFileTable[i] == NULL) {
            DEBUG(dbgFile, name << " is opened successfully");
            OpenFileTable[i] = openFile; // update OpenFileTable
            fileNameTable[i] = fileName; // update descriptorTable
            return i;
        }
    }
    DEBUG(dbgFile, "No space for opening " << name);
    delete openFile; // release memory
    return -1;
}
```

To open the file, check whether the file exists. If the file exists, check whether the file is already opened. If the file name is not in the fileNameTable(an array stores the pointer point to the name of the files that are already open), find an empty slot to store the file pointer, and update the fileNameTable.

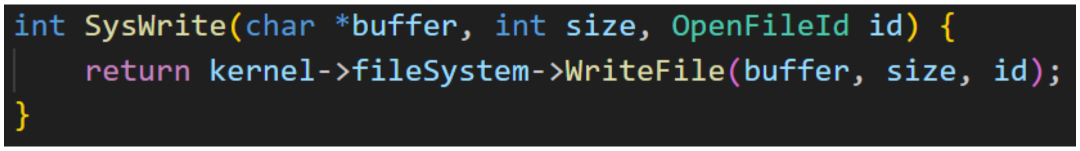
II. int Write(char *buffer, int size, OpenFileId id);

Write “size” characters from the buffer into the file, and return the number of characters written to the file successfully. Return -1, if it fails to write the file.
Path: Run() -> OneInstruction() -> RaiseException() -> EXceptionHandler() -> SysWrite() -> FilsSystem :: WriteFile() -> OpenFile :: Write()



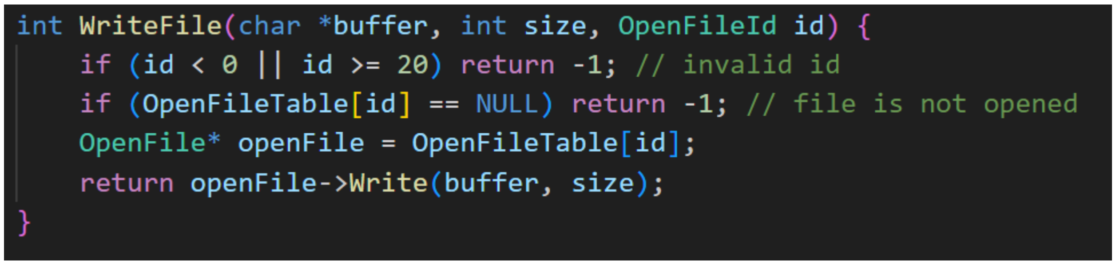
```
case SC_Write: // TODO
    val = kernel->machine->ReadRegister(4);
    {
        int size = kernel->machine->ReadRegister(5); // size is in r5
        int id = kernel->machine->ReadRegister(6); // id is in r6
        char *filename = &(kernel->machine->mainMemory[val]);
        DEBUG(dbgSys, "Write " << filename << endl);
        status = SysWrite(filename, size, id);
        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;
```

In ExceptionHandeler(), read the system call code from R2 to deal with the exception regard to the ExceptionType. In this case, the system code is SC_Write. Read the address of filename from R4, call SysWrite() passing the filename. Then write the return value into R2.



```
int SysWrite(char *buffer, int size, OpenFileId id) {
    return kernel->fileSystem->WriteFile(buffer, size, id);
}
```

Call WriteFile() defined in FileSystem, and retrun the return value.



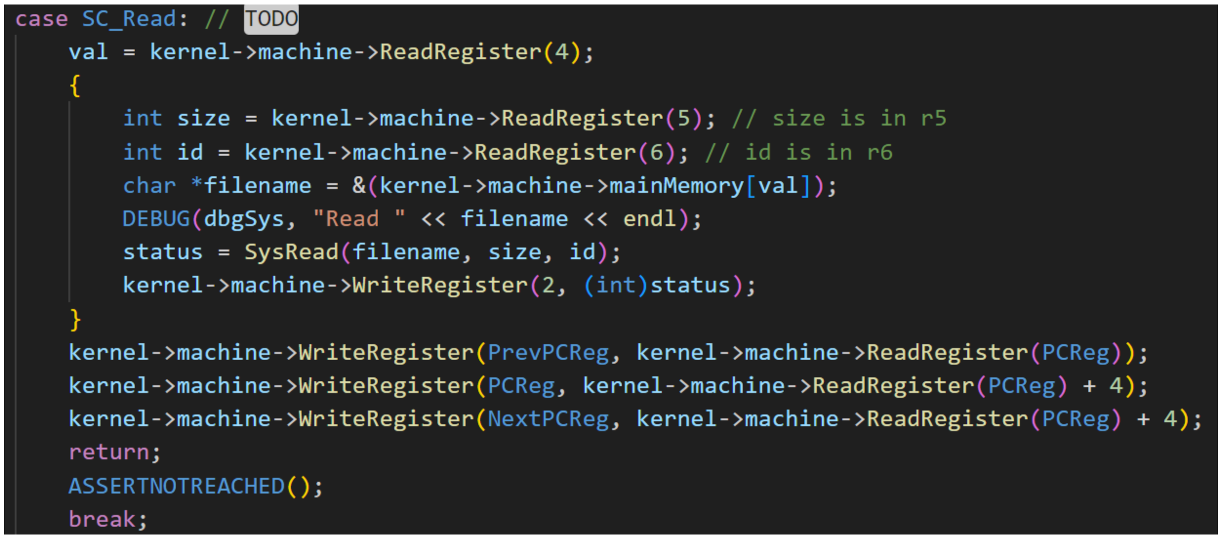
```
int WriteFile(char *buffer, int size, OpenFileId id) {
    if (id < 0 || id >= 20) return -1; // invalid id
    if (OpenFileTable[id] == NULL) return -1; // file is not opened
    OpenFile* openFile = OpenFileTable[id];
    return openFile->Write(buffer, size);
}
```

Check whether the id is valid, and whether the file has been opened. Then call the Write() defined in OpenFile, and retrun the return value.

III. int Read(char *buffer, int size, OpenFileId id);

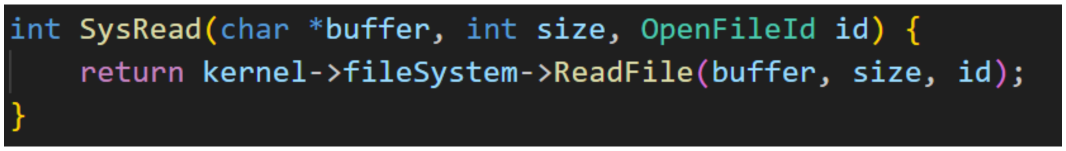
Read “size” characters from the file to the buffer, and return the number of characters read from the file successly. Return -1, if it fails to read the file.

Path: Run() -> OneInstruction() -> RaiseException() -> EXceptionHandler() -> SysRead() -> FilsSystem :: ReadFile() -> OpenFile :: Read ()



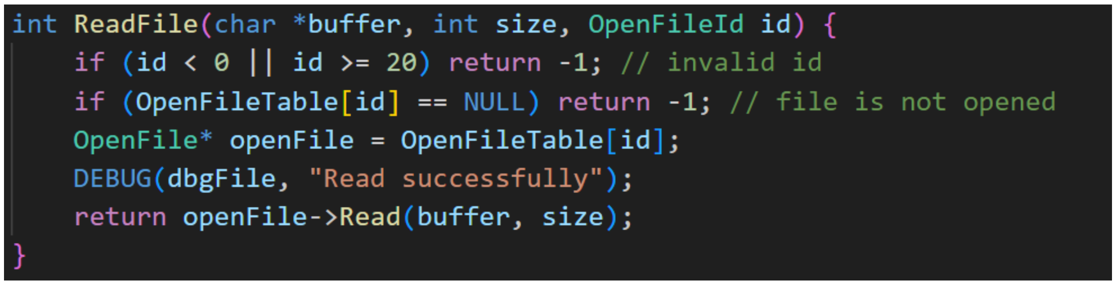
```
case SC_Read: // TODO
    val = kernel->machine->ReadRegister(4);
    {
        int size = kernel->machine->ReadRegister(5); // size is in r5
        int id = kernel->machine->ReadRegister(6); // id is in r6
        char *filename = &(kernel->machine->mainMemory[val]);
        DEBUG(dbgSys, "Read " << filename << endl);
        status = SysRead(filename, size, id);
        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;
```

In ExceptionHandeler(), read the system call code from R2 to deal with the exception regard to the ExceptionType. In this case, the system code is SC_Read. Read the address of filename from R4, call SysRead() passing the filename. Then write the return value into R2.



```
int SysRead(char *buffer, int size, OpenFileId id) {
    return kernel->fileSystem->ReadFile(buffer, size, id);
}
```

Call ReadFile() defined in FileSystem, and retrun the return value.



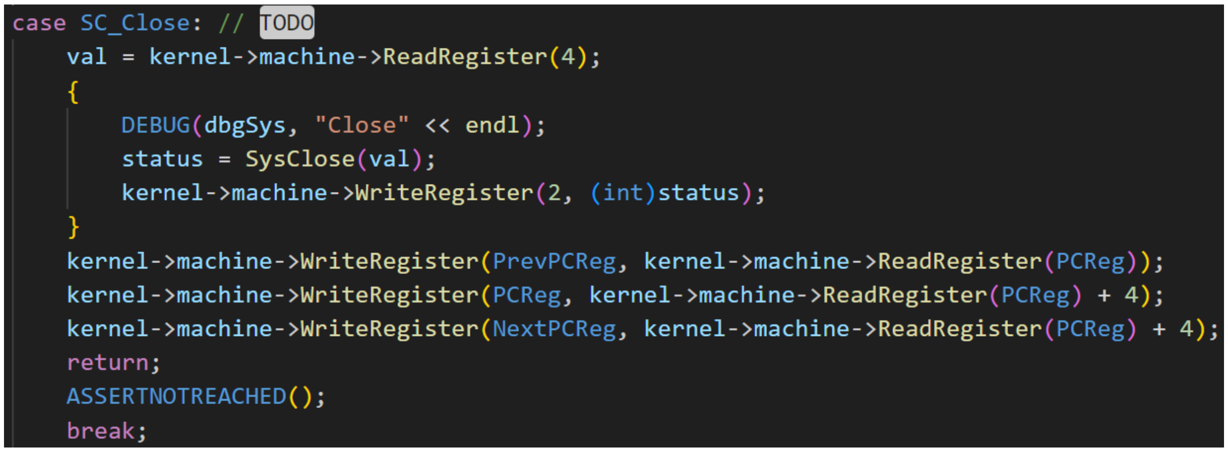
```
int ReadFile(char *buffer, int size, OpenFileId id) {
    if (id < 0 || id >= 20) return -1; // invalid id
    if (OpenFileTable[id] == NULL) return -1; // file is not opened
    OpenFile* openFile = OpenFileTable[id];
    DEBUG(dbgFile, "Read successfully");
    return openFile->Read(buffer, size);
}
```

Check whether the id is valid, and whether the file has been opened. Then call the Write() defined in OpenFile, and retrun the return value.

IV. int Close(OpenFileId id);

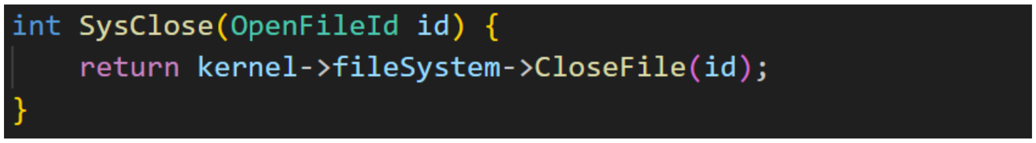
Close the file with id. Return 1 if successfully close the file. Otherwise, return -1. Need to delete the OpenFile after you close the file.

Path: Run() -> OneInstruction() -> RaiseException() -> EXceptionHandler() -> SysClose() -> FilsSystem :: CloseFile()



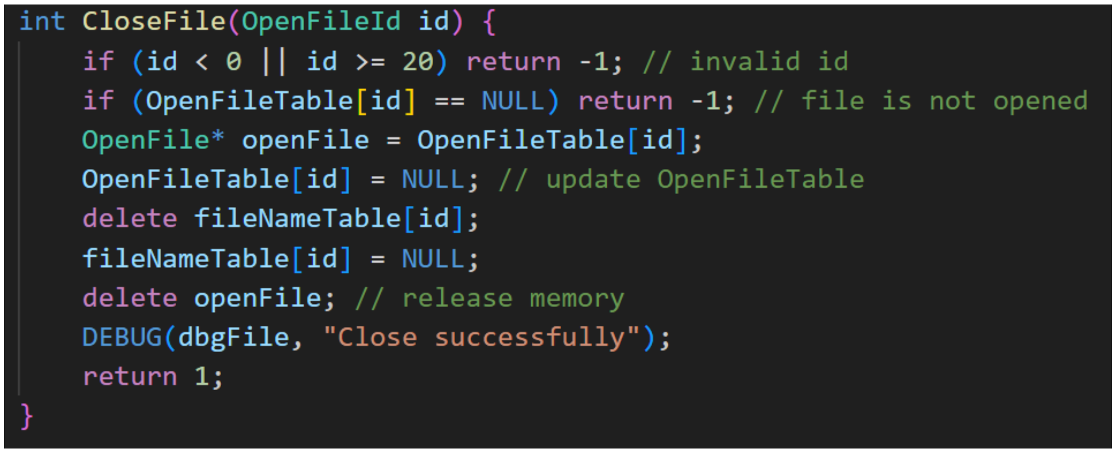
```
case SC_Close: // TODO
    val = kernel->machine->ReadRegister(4);
    {
        DEBUG(dbgSys, "Close" << endl);
        status = SysClose(val);
        kernel->machine->WriteRegister(2, (int)status);
    }
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
    return;
    ASSERTNOTREACHED();
    break;
```

In ExceptionHandeler(), read the system call code from R2 to deal with the exception regard to the ExceptionType. In this case, the system code is SC_Close. Read the address of filename from R4, call SysClose() passing the filename. Then write the return value into R2.



```
int SysClose(OpenFileId id) {
    return kernel->fileSystem->CloseFile(id);
}
```

Call CloseFile() defined in FileSystem, and retrun the return value.



```
int CloseFile(OpenFileId id) {
    if (id < 0 || id >= 20) return -1; // invalid id
    if (OpenFileTable[id] == NULL) return -1; // file is not opened
    OpenFile* openFile = OpenFileTable[id];
    OpenFileTable[id] = NULL; // update OpenFileTable
    delete fileNameTable[id];
    fileNameTable[id] = NULL;
    delete openFile; // release memory
    DEBUG(dbgFile, "Close successfully");
    return 1;
}
```

Check whether the id is valid, and whether the file has been opened. Update the OpenFileTable and the fileNameTable, then delete the corresponding file handler to release the memory. Return 1 at the end.

3. Difficulties

While implementing the function OpenFileId Open(char *name) in part II, we found that we can’t tell whether the file is already opened or not by the OpenFileTable. Thus, we add a private array fileNameTable in the class OpenFile to store the pointer point to the name of the files that are already open. Then, we can check by comparing the file name with the point by the pointer store in the fileNameTable iteratively. However, we don't think this is a good approach because, in reality, we also need to consider the issue of absolute and relative paths. We've also tried other methods, such as using file descriptors to determine this, but none of them have been able to achieve this functionality.

4. Feedback

黃子恩

After tracing the code and implementing several functions on Nachos, we gained a deeper understanding of operating systems. This lab was an invaluable opportunity to apply theoretical knowledge in a practical setting, which has greatly enhanced my grasp of operating system fundamentals.

劉祐廷

This assignment allowed me to actually observe how the OS switches between user mode and kernel mode. However, unlike the code I've seen before, this code often leaves me unsure about which part to trace because functions with the same name often appear more than once. Additionally, the include paths are not written out in full, making it difficult to find the correct files when tracing the code.