Operating System

MP1: System Call

Team Member: 林子謙、陳奕潔

Contributions: Both contributed to system call implementing and report writing.

Contents

1 Code Tracing

1.1 SC_Halt

The SC_Halt system call is designed to shut down NachOS.

1.1.1 Machine::Run()

```
54 void Machine::Run() {
55     Instruction *instr = new Instruction;  // storage for decoded instruction
56     if (debug->IsEnabled('m')) {
57         cout << "Starting program in thread: " << kernel->currentThread->getName();
58         cout << ", at time: " << kernel->stats->totalTicks << "\n";
59     }
60     kernel->interrupt->setStatus(UserMode);
61     for (;;) {
62         DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
63                             << "== Tick " << kernel->stats->totalTicks << " ==");
64         OneInstruction(instr);
65         DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  "
66                             << "== Tick " << kernel->stats->totalTicks << " ==");
67 
68         DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
69                             << "== Tick " << kernel->stats->totalTicks << " ==");
70         kernel->interrupt->OneTick();
71         DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
72                             << "== Tick " << kernel->stats->totalTicks << " ==");
73         if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
74             Debugger();
75     }
76 }
```

Figure 1: Machine::Run()

`Machine::Run()` method initializes the status to **User Mode** and then enters an infinite loop. Within the loop, it **fetches an instruction** by calling `OneInstruction(instr)` on line 64 and **increments the system time** by calling `kernel->interrupt->OneTick()`. You may refer to Figure 1.

1.1.2 Machine::OneInstruction()

Once `Machine::OneInstruction()` is invoked by `Machine::Run()`, it reads an instruction from the program counter and decodes it (on lines 133 to 136, you may refer to Figure 2).

Next, the instruction is handled by going through several switch cases based on its opcode. In the **Sys_Halt** example, `Machine::OneInstruction()` enters the **OP_SYSCALL** case (lines 663 to 666, see Figure 3), invokes `RaiseException()`, and passes the corresponding exception type to it.

```
void Machine::OneInstruction(Instruction *instr) {
#ifdef SIM_FIX
    int byte;  // described in Kane for LWL,LWR,...
#endif

    int raw;
    int nextLoadReg = 0;
    int nextLoadValue = 0;  // record delayed load operation, to apply
                            // in the future

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

    if (debug->IsEnabled('m')) {
        struct OpString *str = &opStrings[instr->opCode];
        char buf[80];

        ASSERT(instr->opCode <= MaxOpcode);
        cout << "At PC = " << registers[PCReg];
        sprintf(buf, str->format, TypeToReg(str->args[0], instr),
                TypeToReg(str->args[1], instr), TypeToReg(str->args[2], instr));
        cout << "\t" << buf << "\n";
    }

    // Compute next pc, but don't install in case there's an error or branch.
    int pcAfter = registers[NextPCReg] + 4;
    int sum, diff, tmp, value;
    unsigned int rs, rt, imm;

    // Execute the instruction (cf. Kane's book)
    switch (instr->opCode) {⋯

    // Now we have successfully executed the instruction.

    // Do any delayed load operation
    DelayedLoad(nextLoadReg, 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;
}
```

Figure 2: Machine::OneInstruction()

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

        case OP_XOR:
            registers[instr->rd] = registers[instr->rs] ^ registers[instr->rt];
            break;

        case OP_XORI:
            registers[instr->rt] = registers[instr->rs] ^ (instr->extra & 0xffff);
            break;

        case OP_RES:
        case OP_UNIMP:
            RaiseException(IllegalInstrException, 0);
            return;

        default:
            ASSERT(FALSE);
    }
```

Figure 3: Machine::OneInstruction() case: OP_SYSCALL

1.1.3 Machine::RaiseException()

Since this instruction is a type of **system call**, it must be handled in **kernel mode**. Therefore, the system invokes `Machine::RaiseException()` to **switch to kernel mode** and clean up all executed instructions by calling `DelayedLoad(0, 0);`, and then calls `ExceptionHandler()` to handle the exception. After returning from `ExceptionHandler()`, the system may **switch back to user mode**. You may refer to Figure 4 for details.

```
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);
}
```

Figure 4: Machine::RaiseException()

1.1.4 ExceptionHandler()

`ExceptionHandler()` first **retrieves the system call exception type from register r2**, as defined in `userprog/syscall.h` (see Figure 5, line 53). In the case of `Halt()`, the system call exception type code is defined as 0.

Next, based on the system call exception type code, the program enters the **SC_Halt** case and calls `SysHalt()`. (As shown in Figure 6.)

```
void ExceptionHandler(ExceptionType which) {
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception "
                      << which << " type: " << type << ", "
                      << kernel->stats->totalTicks);
    switch (which) {···
    ASSERTNOTREACHED();
}
```

Figure 5: ExceptionHandler()_SC_Halt_1

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

Figure 6: ExceptionHandler()_SC_Halt_2

```
void SysHalt() { kernel->interrupt->Halt(); }
```

Figure 7: SysHalt()

1.1.5 SysHalt()

At this point, the program enters the **kernel system call** layer to invoke a system call. (Please refer to Figure 7 for more details.)

1.1.6 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.
}
```

Figure 8: Halt()

Finally, as shown in Figure 8, it prints detailed information about this system execution and deletes the kernel object, which signifies the shutdown of NachOS.

1.2 SC_Create

SC_Create is a kind of system call designed to create a file.

1.2.1 ExceptionHandler()

Since `Create()` is a type of **exception**, it should be handled in `ExceptionHandler()`. In this case, the program enters the **SC_Create** case on line 98, as shown in Figure 9.

Next, the program fetches the arguments provided by the user program by **reading register r4** and stores them in the variable `val`. Since `val` contains the memory address of the filename, the program **retrieves the actual filename address** and stores it in the variable `filename`, which is then passed to the kernel system call `SysCreate()`.

When the program returns from `SysCreate()`, it **writes the return value into register r2** and **increments the program counter** before returning.

```
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;
```

Figure 9: ExceptionHandler_SC_Create

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

Figure 10: SysCreate()

1.2.2 SysCreate()

After entering the kernel system call layer, the program invokes the `Create()` system call in the `FileSystem` class, passing the filename to it.(Please refer to Figure 10 for more details.)

1.2.3 FileSystem::Create()

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

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

Figure 11: FileSystem::Create()

In this assignment, we use the **stub** file system, which means we are actually calling the UNIX file system calls. As shown in Figure 11, the program calls `OpenForWrite()`, which internally invokes the UNIX `open()` function to obtain a file descriptor. The result, indicating success or failure, is then returned to the upper layer.

1.3 SC_PrintInt

The SC_PrintInt system call is designed to print integers to console display devices.

1.3.1 ExceptionHandler()

`ExceptionHandler()` is invoked when a user program makes a system call or raises an exception due to an illegal operation. This function serves as an intermediary, allowing the operating system to execute kernel-level code in response to events or requests from user programs.

System call arguments are passed from the user program to the kernel via specific CPU registers. The first step in handling the exception is to read register `r2` to determine the type of the system call and store into `type`, as shown on line 53. In this example, the value of `r2` is set to the constant `SC_PrintInt`, which is defined as `16` for the `PrintInt` system call. The function then checks if the exception type is a system call by examining the value of the `which` variable (line

```
void ExceptionHandler(ExceptionType which) {
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception "
                           << which << " type: " << type << ", "
                           << kernel->stats->totalTicks);
    switch (which) {
        case SyscallException:
            switch (type) {
                case SC_Halt:
                    DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
                    SysHalt();
                    cout << "in exception\n";
                    ASSERTNOTREACHED();
                    break;
                case SC_PrintInt:
                    DEBUG(dbgSys, "Print Int\n");
                    val = kernel->machine->ReadRegister(4);
                    DEBUG(dbgTraCode,
                          "In ExceptionHandler(), into SysPrintInt, "
                              << kernel->stats->totalTicks);
                    SysPrintInt(val);
                    DEBUG(dbgTraCode,
                          "In ExceptionHandler(), return from SysPrintInt, "
                              << kernel->stats->totalTicks);
                    // Set Program Counter
                    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;
```

Figure 12: ExceptionHandler() in userprog/exception.cc

60). If `which` matches `SyscallException`, the function proceeds to handle system calls by entering a nested switch case based on the specific system call type stored in `type`.

In the case of `SC_PrintInt`, the argument for the this system call, the integers to be printed, is passed through register `r4` and stored in `val` (line 70). This value, `val`, is then passed as an argument to the function `SysPrintInt`, which executes the print operation.

The `dbgSys` of debugging messages is generated before obtaining the value of `val`, showing that the system is going to handle the print integer system call. As for the `dbgTraCode` of debugging messages, they are before and after the `SysPutInt()` call. These messages help track the execution flow of the function.

After executing `SysPrintInt`, the `ExceptionHandler()` function adjusts the program counter registers to resume execution of the user program on line 79 to 84. Specifically, it sets `PrevPCReg` to the current program counter (`PCReg`), then increments `PCReg` by 4 to point to the next instruction, and finally, it updates `NextPCReg` to the new value of `PCReg` + 4. This ensures that the user program continues executing from the instruction following the system call. If no matching case is found for the type of system call or exception, the function triggers an assertion failure, terminating execution due to an unhandled case (You may refer to Figure 12).

1.3.2 SysPrint()

```
void SysPrintInt(int val) {
    DEBUG(dbgTraCode,
          "In ksyscall.h:SysPrintInt, into synchConsoleOut->PutInt, "
              << kernel->stats->totalTicks);
    kernel->synchConsoleOut->PutInt(val);
    DEBUG(dbgTraCode,
          "In ksyscall.h:SysPrintInt, return from synchConsoleOut->PutInt, "
              << kernel->stats->totalTicks);
}
```

Figure 13: SysPrint() in userprog/ksyscall.h

The `SysPrintInt()` function, as illustrated in Figure 13, provides a kernel-level interface for handling the `SC_PrintInt` system call, enabling user programs to print integer values to the console display device.

Within this function, the `PutInt()` method of the `synchConsoleOut` object is invoked, with the integer value `val` passed as an argument.

Additionally, debug messages are generated immediately before and after the `PutInt()` call, logging the current tick count using `kernel->stats->totalTicks`.

1.3.3 SynchConsoleOutput::PutInt()

```
100 void SynchConsoleOutput::PutInt(int value) {
101     char str[15];
102     int idx = 0;
103     // sprintf(str, "%d\n\0", value);  the true one
104     sprintf(str, "%d\n\0", value);  // simply for trace code
105     lock->Acquire();
106     do {
107         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutInt, into consoleOutput->PutChar, " << kernel->stats->totalTicks);
108         consoleOutput->PutChar(str[idx]);
109         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutInt, return from consoleOutput->PutChar, " << kernel->stats->totalTicks);
110         idx++;
111
112         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutInt, into waitFor->P(), " << kernel->stats->totalTicks);
113         waitFor->P();
114         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutInt, return form  waitFor->P(), " << kernel->stats->totalTicks);
115     } while (str[idx] != '\0');
116     lock->Release();
117 }
```

Figure 14: SynchConsoleOutput::PutInt() in userprog/synchconsole.cc

This `SynchConsoleOutput::PutInt()` method, part of the `SynchConsoleOut` class, manages the actual process of outputting the integers to the console in a synchronized manner, ensuring thread-safe and continuous output. You may refer to Figure 14.

The integer `value` is first converted into a character array `str` using `sprintf()`, with a newline character (`\n`) added for readability and a null terminator (`\0`) to mark the end of the string. Besides, the index `idx` keeps track of the current character being output.

A look is acquired at the start of the function to prevent concurrent modifications during the output operation on line 105. In the `while` loop, characters are printed one by one by calling `PutChar()` on `ConsoleOutput`, with `idx` incremented after each call. After each character is printed, `waitFor->P()`, the consumer, is invoked to synchronize output with the availability of resources, likely ensuring the console is ready for the next character.

The loop continues until the null terminator (`\0`) is reached, at which point lock is released, allowing other processes to access.

1.3.4 ConsoleOutput::PutChar()

This `ConsoleOutput::PutChar()` method, part of the `ConsoleOut` class, handles outputs a single character to the simulated console. You may refer to Figure 15.

It first verifies that no other character is currently being output by asserting `putBusy == FALSE`. If no other output is in progress, it writes the character `ch` to a file associated with the console using `WriteFile()`. The method then sets `putBusy` to `TRUE` to indicate that an output operation is active. Finally, it schedules an interrupt using `Schedule()`, specifying a delay of `ConsoleTime` and an

```cpp
void ConsoleOutput::PutChar(char ch) {
    ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, &ch, sizeof(char));
    putBusy = TRUE;
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
}
```

Figure 15: ConsoleOutput::PutChar() in machine/console.cc

interrupt type of `ConsoleWriteInt` to manage the timing of the next output (because of console's asychronous nature). This delay and interrupt emulate the behavior of a hardware console that signals once output is complete (this is why the `ConsoleOut` class iherits from `CallBackObj` object-the `ConsoleOut` class allows the interrupt to call back once the output is finished).

1.3.5 Interrupt::Schedule()

```cpp
void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type) {
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);

    DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] << " at time = " << when);
    ASSERT(fromNow > 0);

    pending->Insert(toOccur);
}
```

Figure 16: Interrupt::Schedule() in machine/interrupt.cc

The `Interrupt::Schedule()` simulates scheduling a hardware interrupt to occur at a specified future time. You may refer to Figure 16.

First, it calculates `when`, the simulated time at which the interrupt should occur, by adding `fromNow` to the current tick count (`kernel->stats->totalTicks`). Then, it creates a `PendingInterrupt` object, `toOccur` containing the callback object `toCall`, the timing `when`, and the interrupt type `type`. After verifying the delay `fromNow` is valid, it insert `toOccur` into pending interrupt list, where the interrupt will be processed when its scheduled time arrives.

1.3.6 Interrupt::CheckIfDue()

The `Interrupt::CheckIfDue()` method is called by `Interrupt::Idle()` to check if any scheduled interrupts are ready to execute. You may refer to Figure 17. When `ConsoleOutput::PutChar()` finishes, `waitfor->P()` triggers `Thread::Sleep()`, which then calls `Interrupt::Idle()`. `Interrupt::Idle()` verifies pending interrupts by calling `Interrupt::CheckIfDue()` with `advanceClock`

```
bool Interrupt::CheckIfDue(bool advanceClock) {
    PendingInterrupt *next;
    Statistics *stats = kernel->stats;

    ASSERT(level == IntOff);  // interrupts need to be disabled,
                              // to invoke an interrupt handler
    if (debug->IsEnabled(dbgInt)) {
        DumpState();
    }
    if (pending->IsEmpty()) {  // no pending interrupts
        return FALSE;
    }
    next = pending->Front();

    if (next->when > stats->totalTicks) {
        if (!advanceClock) {  // not time yet
            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 spinning.
        }
    }

    DEBUG(dbgInt, "Invoking interrupt handler for the ");
    DEBUG(dbgInt, intTypeNames[next->type] << " at time " << next->when);

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

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

Figure 17: Interrupt::CheckIfDue() in machine/interrupt.cc

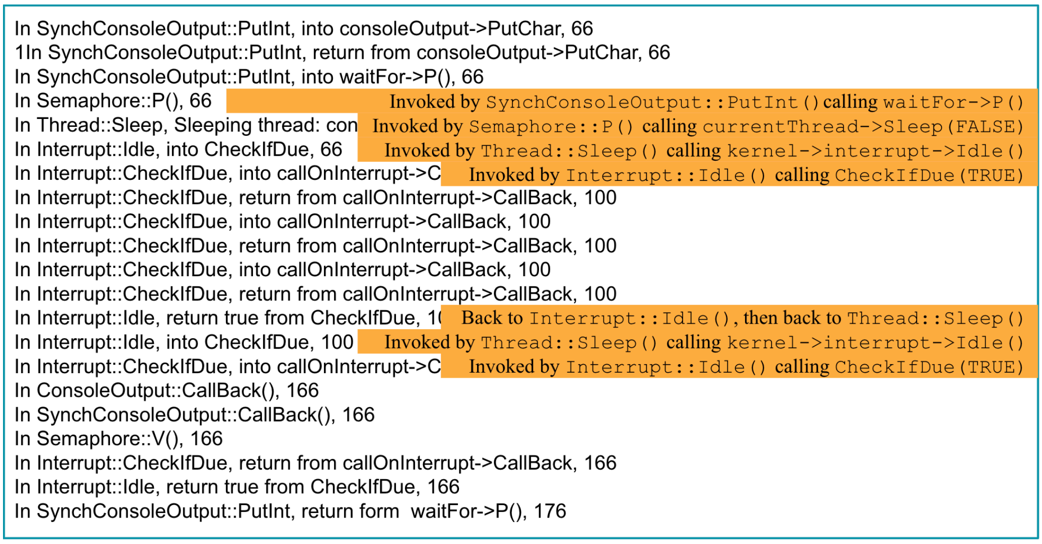


Figure 18: Flow of calling Interrupt::CheckIfDue()

being `TRUE` (meaning the ready queue is empty). You may refer to Figure 18. These operations simulate a process requesting resources and periodically checking for interrupt callbacks.

First, `Interrupt::CheckIfDue()` ensures that interrupts are disabled to prevent concurrent executions. If debugging is enabled, it displays the current tick count and pending interrupts. It then checks if there are any pending interrupts. If not, it returns `FALSE`. In this example, pending interrupts exist, so it retrieves the next scheduled interrupt and checks if it is due by comparing `next->when` with `stats->totalTicks` to see if it is due. If `next->when` is greater than the current time and `advanceClock` is `TRUE`, it updates idle time and advances the clock to `next->when`.

The method then clears the CPU state for interrupt handling and sets `TRUE` to `inHandler`. It processes to remove each due interrupt from the list and invoke its interrupt handler by calling `next->callOnInterrupt->CallBack()`. This call triggers the specific callback associated with the interrupt, enabling the appropriate action to be executed-in this example, signaling that console output is complete. The loop continues until no more pending interrupts are due. Once all due interrupts are handled, `inHandler` is set back to `FALSE` to indicate that the system has exited interrupt handling, and the function returns `TRUE`.

1.3.7 ConsoleOutput::CallBack()

The `ConsoleOutput::CallBack()` method is triggered by `Interrupt::CheckIfDue()` to signal that `ConsoleOutput::PutChar()` has completed. You may refer to Fig-

```
void ConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

Figure 19: ConsoleOutput::CallBack() in machine/console.cc

ure 19.

The method first sets `putBusy` back to `FALSE`, indicating that the console is ready for the next character. It then increment `kernel->stats->numConsoleCharsWritten` to update the count of characters successfully output. Finally, it calls `callWhenDone->CallBack()`, which signals the next character can process.

1.3.8 SynchConsoleOutput::CallBack()

```
void SynchConsoleOutput::CallBack() {
    DEBUG(dbgTraCode,
            "In SynchConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    waitFor->V();
}
```

Figure 20: SynchConsoleOutput::CallBack() in userprog/console.cc

The `SynchConsoleOutput::CallBack()` method is triggered by `Interrupt::CheckIfDue()` to signal it is safe to send the next character to the console. You may refer to Figure 20.

The method calls `waitFor->V()`, the producer, which releases the semaphore, increasing the availability of resources-in this case, allowing the next character to be output to the console.

1.3.9 Machine::Run()

The `Machine::Run()` method simulates the execution of a user-level program, processing instructions one by one. You may refer to Figure 21.

It first creates `instr` to hold each decoded instruction. After setting the execution mode to user mode, it enters an infinite `for` loop. In each loop iteration, `OneInstruction(instr);` is called to execute a single instruction, followed by `OneTick()` to advance one clock. If single-step debugging is enabled and the specified tick count is reached, `Debugger()` is called to display debug messages.

```
void Machine::Run() {
    Instruction *instr = new Instruction;  // storage for decoded instruction
    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
                      << "== Tick " << kernel->stats->totalTicks << " ==");
        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 << " ==");
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

Figure 21: Machine::Run() in machine/mipssim.cc

1.3.10 Interrupt::OneTick()

The `Interrupt::OneTick()` method is called by `Machine::Run()` to advance the simulated clock by one tick. You may refer to Figure 22.

The method first updates `stats->totalTicks` and either `stats->systemTicks` or `stats->userTicks`, depending on the current execution mode. As time moves on, it is necessary to verify if any pending interrupts are due. The method then temporarily disables interrupts to check for any pending interrpts that are due by calling `CheckIfDue(FALSE)` (since it supposes there exist instructions not executed), re-enabling interrupts afterward.

The second thing to check is whether any interrupt handlers call back and request a context switching using `yieldOnReturn`. If `yieldOnReturn` is set, the method sets the mode to the kernel mode, `SystemMode`, performs the context switching and then restores the previous mode.

1.4 Makefile

A Makefile is a special file that contains paths to various resources, shell commands, and all the dependency records. You can think of it as a large shell script which can be executed partly by using `make` command. We explain the functionality of the Makefile by dividing it into three parts: variable definitions, shell com-

```
void Interrupt::OneTick() {
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

    // advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

    // 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
    if (yieldOnReturn) {         // if the timer device handler asked
                                 // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode;  // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

Figure 22: Interrupt::OneTick() in machine/interrupt.cc

mands and dependencies, and the clean-up and error messages.

1.4.1 Variable Definitions

```
102 include Makefile.dep
103
104 CC = $(GCCDIR)gcc
105 AS = $(GCCDIR)as
106 LD = $(GCCDIR)ld
107
108 INCDIR =-I../userprog -I../lib
109 CFLAGS = -g -G 0 -c $(INCDIR) -B/usr/bin/local/nachos/lib/gcc-lib/decstation-ultrix/2.95.2/ -B/usr/bin/local/nachos/decstation-ultr
110
111 ifeq ($(hosttype),unknown)
112 PROGRAMS = unknownhost
113 else
114 # change this if you create a new test program!
115 PROGRAMS = add halt createFile LotOfAdd
116 endif
117
```

Figure 23: MakeFile: Variable Definitions

As shown in Figure 23, at the beginning, the Makefile includes Makefile.dep, which defines many paths using variables. Next, from lines 104 to 106, it uses the variables defined in Makefile.dep to set the paths for several important tools, such as the compiler, assembler, and linker, in the new variables CC, AS, and LD, respectively. From lines 108 to 109, the Makefile defines INCDIR as the path where the compiler can find the included header files, and CFLAGS as part of the command that specifies how the compiler should compile the files.

1.4.2 Shell Commands and Dependencies

```
191 fileIO_test1.o: fileIO_test1.c
192     $(CC) $(CFLAGS) -c fileIO_test1.c
193 fileIO_test1: fileIO_test1.o start.o
194     $(LD) $(LDFLAGS) start.o fileIO_test1.o -o fileIO_test1.coff
195     $(COFF2NOFF) fileIO_test1.coff fileIO_test1
196
197 fileIO_test2.o: fileIO_test2.c
198     $(CC) $(CFLAGS) -c fileIO_test2.c
199 fileIO_test2: fileIO_test2.o start.o
200     $(LD) $(LDFLAGS) start.o fileIO_test2.o -o fileIO_test2.coff
201     $(COFF2NOFF) fileIO_test2.coff fileIO_test2
```

Figure 24: MakeFile: Shell Commands and Dependencies

Since there are many similar code files, we will use `fileIO_test1.c` and `fileIO_test2.c` as examples to explain as shown in Figure 24.

The process can be divided into two parts: compiling and linking. For the compiling part, from lines 191 to 192, the Makefile uses the variables defined earlier, `CC` and `CFLAGS`, to compile `fileIO_test1.c` into `fileIO_test1.o`. For the linking part, from lines 193 to 195, the Makefile uses several variables to link fileIO_test1.o and start.o into an executable file, fileIO_test1. The process for fileIO_test2 is the same as for fileIO_test1.

1.4.3 Clean-Up and Error Messages

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

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

unknownhost:
	@echo Host type could not be determined.
	@echo make is terminating.
	@echo If you are on an MFCF machine, contact the instructor to report this problem
	@echo Otherwise, edit Makefile.dep and try again.
```

Figure 25: MakeFile: Clean-Up and Error Messages

clean and distclean are two targets defined in the Makefile. These targets are actually commands made up of several predefined variables. The clean target is used to remove all .o, .ii, and .coff files, while distclean is used to clean up the entire executable program. Lastly, unknownhost is a target used to output error messages.

2 System Call Implementation

In this section, we implement four I/O system calls for NachOS: SC_Open, SC_Read, SC_Write, and SC_Close. We begin by examining their usage in fileIO_test1 and fileIO_test2, referencing the #define definitions in userprog/syscall.h where these system calls are initially commented out.

Using these files and the hints from the course slide, we observe that the flow of these I/O system calls follows a similar pattern, as illustrated in Figure 26. First, we explain the common steps (the first two steps) in the flow for all four I/O system call, then we detail the unique aspects of each call separately.

First of all, we uncomment lines 27, 28, 29, and 31 to enable the system calls SC_Open, SC_Read, SC_Write, and SC_Close in userprog/syscall.h. You may refer to Figure 27.

Subsequently, we add assembly code for the Open, Write, Read, and Close system calls, following the existing system calls' logic. Each system calls first loads its corresponding system call value into register r2, executes the system call using the syscall instruction, and finally returns to the caller's address. You may refer to Figure 28.

userprog/syscall.h

```
#define SC_Open 6
#define SC_Read 7
#define SC_Write 8
#define SC_Close 10
```

test/start.S

```
/* add Open, Write, Read, and Close assembly code */
```

userprog/exception.cc

ExceptionHandler()

userprog/ksyscall.h

SysOpen()
SysWrite()
SysRaed()
SysClose()

filesys/filesys.h

OpenAFile()
WriteFile()
ReadFile()
CloseFile()

Figure 26: Flow of implementing the four I/O system calls

```
#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
#define SC_ExecV 13
#define SC_ThreadExit 14
#define SC_ThreadJoin 15
#define SC_PrintInt 16
#define SC_Add 42
#define SC_MSG 100
#ifndef IN_ASM
```

Figure 27: Removing the comments from the code

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

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

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

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

    .globl Halt
    .ent    Halt
```

Figure 28: Adding assembly code in Start.s

2.1 Open

2.1.1 ExceptionHandler()

```
116 case SC_Open:
117     /*
118     #define SC_Open 6
119     Open a file for read & write.
120     */
121     DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Open.");
122     val = kernel->machine->ReadRegister(4);  // Retrieve file name address.
123     {
124         char *filename = &(kernel->machine->mainMemory[val]); // Retrive file name.
125         DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Open, into SysOpen.");
126         fileID = SysOpen(filename);  // Success: get file ID / Fail: get -1
127         DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Open, return from SysOpen.");
128         kernel->machine->WriteRegister(2, fileID); // Write file ID into register.
129     }
130     kernel->machine->WriteRegister(
131         PrevPCReg, kernel->machine->ReadRegister(PCReg));
132     kernel->machine->WriteRegister(
133         PCReg, kernel->machine->ReadRegister(PCReg) + 4);
134     kernel->machine->WriteRegister(
135         NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
136     return;
137     ASSERTNOTREACHED();
138     break;
139
```

Figure 29: ExceptionHandler Case: SC_Open

In `ExceptionHandler()`, we add a case for `SC_Open`. You may refer to Figure 29.

First, we retrieve the system call argument stored in register `r4`, which holds the address of the file name in memory, on line 122. To avoid duplicate assignment, we use a block to define `filename` and `fileID`. The file name the user want to open is obtained by accessing the physical memory at the address from `r4`. The file name is then passed to the `SysOpen()`, which returns an opened file ID (`fileID`). We store this opened file id in register `r2`, as `r2` is the return value register in MIPS.

Finally, we follows the existing code to advance program counter's value and perform necessary checks.

2.1.2 SysOpen()

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

Figure 30: ksyscall.h/SysOpen()

We uncomment the `SysOpen()` function, which calls `OpenAFile()` in `FileSystem`, to handle the actual Open system call. You may refer to Figure 30.

2.1.3 OpenAFile()

```
OpenFileId OpenAFile(char *name) {
    DEBUG(dbgTraCode, "In FileSystem::OpenAFile(), file name = " << name);
    // Handling duplicate file opening
    for (int idx = 0; idx < 20; ++idx) {
        if ((OpenFileNames[idx] != NULL) &&
            !strcmp(OpenFileNames[idx], name)) {
            DEBUG(dbgTraCode,
                    "In FileSystem::OpenAFile(), duplicate opening is found: " << OpenFileNames[idx]);
            return -1;
        }
    }
    // Searching an empty spot to place the opened file control object ptr &
    // to record file name
    for (int idx = 0; idx < 20; ++idx) {
        if (OpenFileTable[idx] == NULL) {
            // Get file fd
            int fileDescriptor = OpenForReadWrite(name, FALSE);
            DEBUG(dbgTraCode, "In FileSystem::OpenAFile(), file descriptor(fd): " << fileDescriptor);
            // Handing non-existent file
            if (fileDescriptor == -1) return -1;

            OpenFileTable[idx] = new OpenFile(fileDescriptor);
            DEBUG(dbgTraCode,
                    "Created a new OpenFile and stored into OpenFileTable");
            OpenFileNames[idx] = name;
            DEBUG(dbgTraCode,
                    "Created a new OpenFile and stored its name into "
                    "OpenFileNames");
            return idx;
        }
    }
    // Handling exceeded file opening (at most 20 files)
    return -1;
}
```

Figure 31: filesys.h/OpenAFile()

Before implementing `OpenAFile()`, we define a new array, `OpenFileNames`, to store the names of opened files. The `OpenAFile()` method returns `-1` to indicate a failure to open the file; otherwise, it returns the file ID, which is the index of the file in `OpenFileTable`. You may refer to Figure 31.

We first verify if the file name has already been opened. If a duplicate is found, it returns `-1`. If not, we call a for loop to find an empty spot to locate an OpenFile object pointer. If there is an empty space, we call `OpenForReadWrite` with the file name to obtain a file descriptor (the `FALSE` passed is imitating it in `Open()` method). We then check if the file descriptor is valid. There are two cases: the file does not exist or the maximum limit of open files (20) is exceeded. If the file does exist, we store a new `OpenFile` object and record the file name at the same index in `OpenFileNames` then return the index (`idx`) as the opened file ID.

Figure 31 shows the complete code of our implementation.

```
case SC_Write:
    /*
    #define SC_Write 8
    Write “size” characters from the buffer into the file, and
    return the number of characters actually written to the file
    */
    DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Write.");
    val = kernel->machine->ReadRegister(4);
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        numChar = kernel->machine->ReadRegister(5);
        fileID = kernel->machine->ReadRegister(6);

        DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Write, into SysWrite()");
        numChar = SysWrite(buffer, numChar, fileID); // Success: get number of character written into the file / Fail: get -1
        DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Write, return from SysWrite()");
        kernel->machine->WriteRegister(2, numChar); // Write result into register.
    }
    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;
```

Figure 32: ExceptionHandler Case: SC_Write

2.2 Write

2.2.1 ExceptionHandler()

In `ExceptionHandler()`, we add a case for `SC_Open`. You may refer to Figure 32.

Similar to the `SC_Open` case, `SC_Write` case handles additional arguments, the number of characters to write (`numChar`) and the opened file ID (`fileID`), stored in register `r5` and `r6`, respectively. Instead of a file name, we access the physical memory at the address in `r4` to retrieve the buffer containing the data to write. The `numChar` and `fileID` arguments are stored directly in registers probably since they are small enough to fit.

After calling `SysWrite()`, we store its return value, the number of characters actually written, into register `r2`.

2.2.2 SysWrite()

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

Figure 33: ksyscall.h/SysWrite()

In `ksyscall.h`, we add a function `SysWrite()`. In `SysWrite()`, we call `WriteFile()` , which is also in `FileSystem`, to handle the actual Write system call. You may refer to Figure 33.

2.2.3 WriteFile()

```
int WriteFile(char *buffer, int size, OpenFileId id) {
    DEBUG(dbgTraCode, "In FileSystem::WriteFile()");

    if (id < 0 || id >= 20) {  // Handling invalid file ID
        return -1;
    } else {
        if (OpenFileTable[id] == NULL) {  // Handling file not yet opened
            return -1;
        }
    }
    return OpenFileTable[id]->Write(buffer, size * sizeof(char));
}
```

Figure 34: filesys.h/WriteFile()

In `filesys.h`, we uncomment the `WriteFile()` method. This method returns `-1` to indicate a failure to write the file; otherwise, it returns the number of characters written. You may refer to Figure 34.

In `WriteFile()`, we first check the file ID is within the valid range (0 to 19). If the ID is valid, we then verify the file is open by checking `OpenFileTable[id]`. If the file is open, we call `Write` method, passing `buffer` and `size * sizeof(char)` (size of the characters in memory) as arguments.

You may refer to 34 for details.

2.3 Read

2.3.1 ExceptionHandler()

In `ExceptionHandler()`, we add a case for `SC_Read`. You may refer to Figure 35.

Similar to the `SC_Write` case, the `SC_Read` case calls `SysRead()` instead. We store its its return value, the number of characters actually read, into register `r2`.

2.3.2 SysRead()

In `ksyscall.h`, we add a function `SysRead()`. In `SysRead()`, we call `ReadFile()`, which is also in `FileSystem`, to handle the actual Read system call. You may refer to Figure 36.

2.3.3 ReadFile()

In `filesys.h`, we uncomment the `ReadFile()` method. You may refer to Figure 37.

```cpp
case SC_Read:
    /*
    #define SC_Read 7
    Read "size" characters from the file into the buffer, and
    return the number of characters actually read from the file
    */
   DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Read.");
    val = kernel->machine->ReadRegister(4);
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        numChar = kernel->machine->ReadRegister(5);
        fileID = kernel->machine->ReadRegister(6);

        DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Read, into SysRaed()");
        numChar = SysRaed(buffer, numChar, fileID);
        DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Read, return from SysRead()");
        kernel->machine->WriteRegister(2, numChar);
    }
    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;
```

Figure 35: ExceptionHandler Case: SC_Read

```cpp
int SysRaed(char *name, int size, OpenFileId id) {
Click to add a breakpoint el->fileSystem->ReadFile(name, size, id);
}
```

Figure 36: ksyscall.h/SysRead()

```cpp
int ReadFile(char *buffer, int size, OpenFileId id) {
    DEBUG(dbgTraCode, "In FileSystem::RaedFile()");
    if (id < 0 || id >= 20) {  // Handling invalid file ID
        return -1;
    } else {
        if (OpenFileTable[id] == NULL) {  // Handling file not yet opened
            return -1;
        }
    }
    return OpenFileTable[id]->Read(buffer, size * sizeof(char));
}
```

Figure 37: filesys.h/ReadFile()

Similar to `WriteFile()`, this method verifies the file ID and checks the file is open. If these checks pass, it calls the `Read` method on the file instead, passing `buffer` and `size * sizeof(char)` as arguments.

You may refer to 37 for details.

2.4 Close

2.4.1 ExceptionHandler()

```
case SC_Close:
    /*
    #define SC_Close 10
    Close the file.
    */
   DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Close.");
    val = kernel->machine->ReadRegister(4);
    DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Close, into SysClose()");
    status = SysClose(val);
    DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Close, return from SysClose()");
    kernel->machine->WriteRegister(2, 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;
```

Figure 38: ExceptionHandler Case: SC_Close

In `ExceptionHandler()`, we add a case for `SC_Close`. You may refer to Figure 38.

Similar to the `SC_Open` case, the `SC_Close` case retrieves the value stored in register `r4` directly as the opened file ID, `val`. This ID is then passed to `SysClose()`, which returns a status (`status`) indicating whether the file was successfully closed (`1` for success, `-1` for failure).

2.4.2 SysClose()

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

Figure 39: ksyscall.h/SysClose()

In `ksyscall.h`, we add a function `SysClose()`. In `SysClose()`, we call `CloseFile()` , which also in `FileSystem`, to handle the actual Close system call. You may refer to Figure 39.

```
int CloseFile(OpenFileId id) {
    DEBUG(dbgTraCode, "In FileSystem::CloseFile()");
    if (id < 0 || id > 20) {  // Handling invalid file ID
        return -1;
    } else {
        if (OpenFileTable[id] == NULL) {  // Handling file not yet opened
            return -1;
        }
    }
    delete OpenFileTable[id];
    OpenFileTable[id] = NULL;
    OpenFileNames[id] = NULL;
    return 1;
}
```

Figure 40: filesys.h/CloseFile()

2.4.3 CloseFile()

In `filesys.h`, we uncomment the `CloseFile()` method. You may refer to Figure 40.

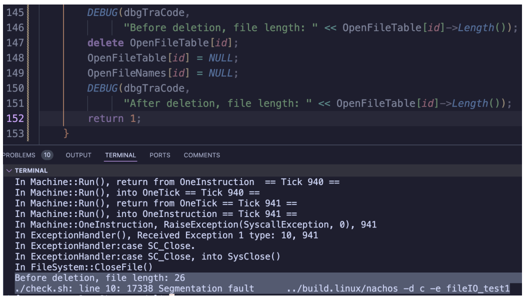
Similar to `WriteFile()`, this method first verifies the file ID and checks the file is open. If these checks pass, we use `delete` to call its `Close` method and close the file. We then set `OpenFileTable[id]` and `OpenFileNames[id]` to `NULL` to free the slot for other files. You may refer to 40 for details.

To confirm the file is closed, we check if calling `Length()` on it after the deletion, which results in a segmentation fault, as illustrated in Figure 41a. In contrast, Figure 41b shows the program running successfully without calling `Length()` after deletion. Therefor, we probably could suggest that the file is indeed closed.

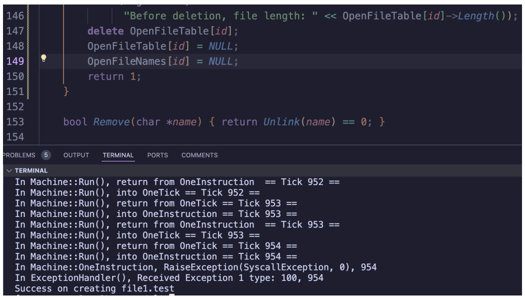
3 Difficulties

SC_PrintInt : In this process, we struggled to identify the timing of character output and how `Interrupt::CheckIfDue()` is invoked. Initially, we used `add.c` to print debug messages, but the small number of characters made it difficult to trace the flow. To address this, we created `consoleIO_test3.c`, as shown in Figure 42, to output more characters. This test helped us pinpoint when `Interrupt::CheckIfDue()` is called and understand the subsequent flow.

Open : When implementing the `Open()` system call, we would like to make sure we handled the file opening and closing process correctly by opening 20 files, then closing one file, and finally opening another file. While reviewing our debug messages, we discovered something tricky about the Unix `Open()` system call.



(a) Call of `Length()` after the deletion



(b) No call of `Length()` after the deletion

Figure 41: Result of two cases

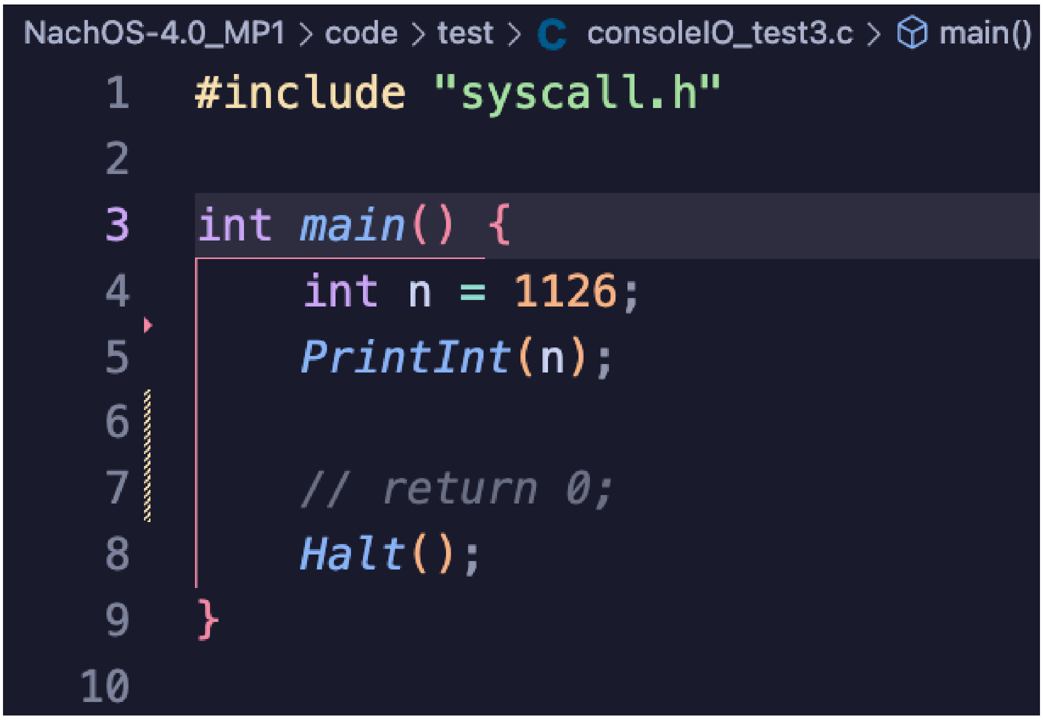


Figure 42: Difficulties: SC_PrintInt

```
86          for (int idx = 0; idx < 20; ++idx) {
87              if (OpenFileTable[idx] == NULL) {
88                  // Get file fd
89                  int fileDescriptor = OpenForReadWrite(name, FALSE);
90                  DEBUG(dbgTraCode, "In FileSystem::OpenAFile(), file descriptor(fd): " << fileDescriptor);
91                  // Handing non-existent file
92                  if (fileDescriptor == -1) return -1;
93
94                  OpenFileTable[idx] = new OpenFile(fileDescriptor);
95                  DEBUG(dbgTraCode,
96                        "Created a new OpenFile and stored into OpenFileTable");
97                  OpenFileNames[idx] = name;
98                  DEBUG(dbgTraCode,
99                        "Created a new OpenFile and stored its name into "
100                       "OpenFileNames");
101                 return idx;
102             }
103         }
104         // Handling exceeded file opening (at most 20 files)
105         return -1;
106     }
```

Figure 43: Difficulties: SC_Open Code

You may first refer to Figure 43 for the details of our code. Initially, we opened file1.test, and then we opened another 19 files, from test0.test to test18.test (these files have been created before we tried to open them). So far, the debug messages look great, every opened file had its own file descriptor as shown in Figure 44. Afterwards, we tried to close test18.test, and the debug message we added in OpenFile destructor (as shown in Figure 45) showed that we had successfully close the file (as shown in Figure 46). When we tried to opened test19.test(this file had also been created before), we found out that the returned value of `Open()` (i.e. file descriptor) is -1, you may refer to Figure 47. As shown in our implementation code (Figure 43), it means that the return value of test19.test was returned on line 92, this means that we indeed closed the file, controled our `OpenFileTable` (because it went into line 87) but we failed to call `Open()` on our 21st try! It was supposed to be returned on line 107!

You may first refer to Figure 43 for the details of our code. Initially, we opened file1.test, followed by another 19 files, from test0.test to test18.test (these files had been created prior to opening them). Up to this point, the debug messages looked great—each file had its own file descriptor, as shown in Figure 44. Next, we closed test18.test, and the debug message from the OpenFile destructor (shown in Figure 45) confirmed that the file had been successfully closed, as shown in Figure 46.

However, when we tried to open test19.test (which had also been created before), we discovered that the returned value of `Open()` (i.e., the file descriptor) was -1. You may refer to Figure 47. According to our implementation code (Figure 43), this return value was produced on line 92, indicating that while we successfully closed the file and managed our `OpenFileTable` (as the code reached line 87), we failed to call `Open()` on our 21st attempt!

The file descriptor was expected to be returned on line 105.

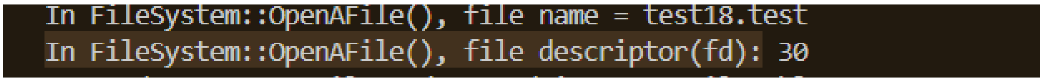


Figure 44: Difficulties: SC_Open Debug Message 1

Figure 45: Difficulties: SC_Open Debug Message 2

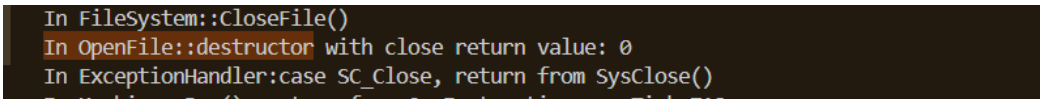


Figure 46: Difficulties: SC_Open Debug Message 3

Write : We encountered some difficulties while implementing the `Write()` system call. Initially, we were puzzled as to why we couldn ' t fetch the arguments, `numChar` and `fileID`, by passing the address obtained from the register into memory. This issue troubled us for quite some time. You may refer to the code shown in Figure 48. Eventually, we resolved the problem by printing debug messages and discovered that the values of these arguments were actually stored in the registers. We didn ' t need to fetch them from memory; instead, we could load them directly from the registers and use them immediately.

After several days, we realized that we had forgotten to check the types of variables passed by the user in `fileIO_test1`. The arguments passed to `Write()` included an address (pointer) and two integers. Naturally, we could load `numChar` and `fileID` directly from the registers since they were not addresses!

Close : In the `CloseFile()` method, we want to confirm that the target `OpenFile` object is truly deleted. However, since the `OpenFile` destructor does not return a value, we cannot use `retVal = delete OpenFileTable[id];` to verify deletion through a debug message. As a result, we decided to consult the TA to ensure our implementation is correct.

```
In FileSystem::OpenAFile(), file name = file19.test
In FileSystem::OpenAFile(), file descriptor(fd): -1
```

Figure 47: Difficulties: SC_Open Debug Message 4

```
case SC_Write:
    /*
    #define SC_Write 8
    Write "size" characters from the buffer into the file, and
    return the number of characters actually written to the file
    */
    DEBUG(dbgTraCode, "In ExceptionHandler:case SC_Write.");
    val = kernel->machine->ReadRegister(4);
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        numChar = kernel->machine->ReadRegister(5);
        fileID = kernel->machine->ReadRegister(6);
```

Figure 48: Difficulties: SC_Write