MP1: System Call Report

110000132 詹振暘, 110021121陶威綸

Team member contribution

陶威綸

Trace code on SC_Halt, SC_PrintInt

Implement Open, Write function of NachOS.

Report

詹振暘

Trace code on SC_Create, SC_PrintInt

Implement Close, Read function of NachOS.

Report

Part I - System Call explanation

Part I-(a) Explain the purposes and details of each function call listed in the code path above.

SC_Halt

Machine::Run()

Set to UserMode

Continuously Call OneInstruction. (Simulate the fetching instruction process in CPU.)

```
void Machine::Run() {
  Instruction *instr = new Instruction;  // storage for decoded instruction
  kernel->interrupt->setStatus(UserMode);
  for (;;) {
    OneInstruction(instr);
    kernel->interrupt->OneTick();
  }
}
```

Machine::OneInstruction()

The process of simulating the step-by-step executions of CPU instruction.

Fetch instruction in memory

Call RaiseException(SyscallException, 0) for processing system calls.

```
void Machine::OneInstruction(Instruction *instr) {
  if (!ReadMem(registers[PCReg], 4, &raw))
    return;  // exception occurred
  instr->value = raw;
  instr->Decode();
  switch (instr->opCode) {
    case OP_ADD:
                ...
          case OP_ADDI:
                ...
          case OP_SYSCALL:
                RaiseException(SyscallException, 0);
          }
}
```

Machine::RaiseException()

Set to kernel mode.

Call the exception handler.

Return back to user mode.

```
void Machine::RaiseException(ExceptionType which, int badVAddr) {
  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);
}
```

ExceptionHandler()

Load system call type by r2.
Load remaining input by r4, r5, r6.
Call SysHalt.

```
void ExceptionHandler(ExceptionType which) {
  int type = kernel->machine->ReadRegister(2);
  switch (which) {
     case SyscallException:
        switch (type) {
           case SC_Halt:
              SysHalt();
              break;
               }
  }
}
```

SysHalt()

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

Interrupt::Halt()

Shut down kernel (end of OS).

```
void Interrupt::Halt() {
  delete kernel;  // Never returns.
}
```

SC_Create

ExceptionHandler()

```
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);
            break;
```

It handles SC_Create system call, which creates a file and reads its file name from user memory. This system call returns the creation status to the user program by writing it to a register. and update the counter afterward.

SysCreate()

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

SysCreate() is a system call that allows user programs to request file creation. Its core function is to receive the file name and call the file system functions in the kernel to create the file.

FileSystem::Create()

```
  bool Create(char *name) {
     int fileDescriptor = OpenForWrite(name);
     if (fileDescriptor == -1)
        return FALSE;
     Close(fileDescriptor);
     return TRUE;
  }
```

The function called the open function in C's native stdlib.h and checks for errors..

SC_PrintInt

ExceptionHandler() ~

SysPrintInt()

```
void SysPrintInt(int val) {
  kernel->synchConsoleOut->PutInt(val);
}
```

SynchConsoleOutput::PutInt()

Turn value into char array.

Put all the char to a simulated display buffer.

```
void SynchConsoleOutput::PutInt(int value) {
  char str[15];
  int idx = 0;
  sprintf(str, "%d\n\0", value);  // simply for trace code
  lock->Acquire();
  do {
     consoleOutput->PutChar(str[idx]);
     idx++;
     waitFor->P();
  } while (str[idx] != '\0');
  lock->Release();
}
```

SynchConsoleOutput::PutChar()

Use lock to control invalid override.

```
void SynchConsoleOutput::PutChar(char ch) {
  lock->Acquire();
  consoleOutput->PutChar(ch);
  waitFor->P();
  lock->Release();
}
```

ConsoleOutput::PutChar()

Write a character into a simulated display .

Schedule the ConsoleOutput object to the buffer.

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

Interrupt::Schedule()

Insert the Interrupt into the pending queue.

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

Machine::Run() ~
Machine::OneTick()

```
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
}
```

OneTick() advances the system time by one tick, simulating time passage. It manages interrupt status, releases the current thread, and switches to the next one. NachOS simulates a clock that increments with each CPU instruction, starting from system boot. When interrupts are re-enabled, the clock advances by 10 ticks, simulating time progression.

Interrupt::CheckIfDue()

```
bool Interrupt::CheckIfDue(bool advanceClock) {
  PendingInterrupt *next;
  Statistics *stats = kernel->stats;
  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;
     }
  }
  if (kernel->machine != NULL) {
     kernel->machine->DelayedLoad(0, 0);
  }

  inHandler = TRUE;
  do {
     next = pending->RemoveFront();  // pull interrupt off list
     next->callOnInterrupt->CallBack();  // call the interrupt handler
     delete next;
  } while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
  inHandler = FALSE;
  return TRUE;
}
```

The CheckIfDue function checks whether there are any pending interrupts that need to be processed based on the current system time. If any interrupts are due, it advances the clock (if necessary), processes the interrupts, and calls their handlers.

ConsoleOutput::CallBack()

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

This function handles the completion of a console output operation. It marks the console as no longer busy, updates statistics, and triggers the next callback in the process chain to continue handling the output or related tasks.

SynchConsoleOutput::CallBack()

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

After the console output is complete, unblock the state and allow other pending operations (such as sending the next character) to continue

Part I-(b) Questions: Explain how the arguments of system calls are passed from user program to kernel in each of the above use cases.

(a) SC_Halt

```
#include "syscall.h"
int main() {
   Halt();
   /* not reached */
}
```

The user program calls the Halt() function in syscall.h:

```
void Halt();
```

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

In Start.s, The system call number for SC_Halt is loaded into register $2 (v0), and triggers the system call.

```
void Machine::OneInstruction(Instruction *instr) {
   int raw;
   int nextLoadReg = 0;
   int nextLoadValue = 0;  // record delayed load operation, to apply
   if (!ReadMem(registers[PCReg], 4, &raw))
       return;  // exception occurred
   instr->value = raw;
   instr->Decode();
}
```

Here, the syscall instruction is decoded, and the appropriate exception (in this case, a system call) is raised

```
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;
```

When the Exception handler is triggered , the system call number will be read from register $2
In case: SC_Halt, the kernel calls the SysHalt() function to handle the halt request, shutting down the system.

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

Finally, the SysHalt() invokes kernel->interrupt->Halt(), which shuts down the system.

(a) SC_Create

```
#include "syscall.h"

int main(void) {
   int success = Create("file0.test");
   if (success != 1)
       MSG("Failed on creating file");
   MSG("Success on creating file0.test");
   Halt();
}
```

The user calls Create() and passes the “file0.test” string.

```
int Create(char *name);
```

Create file is defined in syscall.h

```
   .globl Create
   .ent    Create
Create:
   addiu $2,$0,SC_Create
   syscall
   j   $31
   .end Create
```

After the user calls Create, the assembly code is executed. The code saves the system call type to register 2 and input to register 4 the calls a syscall to switch to the machine kernel.

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

Makefile links the file start.o, createFile.o, syscall.h. Create function is implemented in start.S.

```
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;
```

OneInstruction fetch instr and parse values in each register. The value in register 4 is the input “file0.text” and in register 2 is the syscall type (SC_Create).

(b) SC_PrintInt

```
#include "syscall.h"

int main() {
   int n;
   for (n=9; n>5; n--) {
       PrintInt(n);
   }
   return 0;
}
```

Test file: call PrintInt(n), pass the parameter to register 4.

```
void PrintInt(int number);
```

syscall.h

```
       .globl  PrintInt
       .ent     PrintInt
PrintInt:
       addiu $2,$0,SC_PrintInt
      syscall
       j       $31
       .end  PrintInt
```

After the user calls PrintInt, the assembly code is executed. The code saves the system call type to register 2 and input to register 4 the calls “syscall” to switch to the machine kernel.

```
consoleIO_test1.o: consoleIO_test1.c
   $(CC) $(CFLAGS) -c consoleIO_test1.c
consoleIO_test1: consoleIO_test1.o start.o
   $(LD) $(LDFLAGS) start.o consoleIO_test1.o -o consoleIO_test1.coff
   $(COFF2NOFF) consoleIO_test1.coff consoleIO_test1
```

Makefile links the file consoleIO_test1.o,start.o, syscall.h. PrintInt function is implemented in start.S.

```
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;
```

Finally, It reads register $4, where the user program passed it. Then, it calls SysPrintInt(val) to print the integer val to the console. Afterward, it updates the program counters
(PrevPCReg, PCReg, and NextPCReg) to ensure the user program resumes correctly after the system call. Finally, control is returned to the user program once the integer has been printed.

Part II - Implementation explanation

First, we remove the // in front of #define SC_Open 6 … #define SC_Close 10, then we implement Open, Close, Read, Write int Start.s as other cases that already existed.

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

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

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

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

After modifying Start.s, we start implementing new cases in exception.cc, since we already modified Start.s, now switch(which) can identify the four types, now we can start writing SC_Open, SC_Close, SC_Write, SC_Read cases.

```
case SC_Open:
   DEBUG(dbgSys, "Start opening the file.\n");
   val = kernel->machine->ReadRegister(4);
   {
       char *filename = &(kernel->machine->mainMemory[val]);
       fileID = SysOpen(filename);
       kernel->machine->WriteRegister(2, fileID);
   }
   DEBUG(dbgSys, fileID << " opened.\n");
   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;
case SC_Close:
   fileID = kernel->machine->ReadRegister(4);
   status = SysClose(fileID);
   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;
case SC_Write:
   val = kernel->machine->ReadRegister(4);
   {
       char *buffer = &(kernel->machine->mainMemory[val]);
       numChar = kernel->machine->ReadRegister(5);
       fileID = kernel->machine->ReadRegister(6);
       status = SysWrite(buffer, numChar, fileID);
       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;
case SC_Read:
   val = kernel->machine->ReadRegister(4);
   {
       char *buffer = &(kernel->machine->mainMemory[val]);
       numChar = kernel->machine->ReadRegister(5);
       fileID = kernel->machine->ReadRegister(6);
       status = SysRead(buffer, numChar, fileID);
   }
   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;
```

SC_Open: Reads the file name address from register $4, calls the SysOpen function to open the file, and stores the returned fileID (file identifier) in register $2.

SC_Close: Reads the fileID from register $4, calls the SysClose function to close the file, and writes the status (success or failure) into register $2.

SC_Write: Reads the buffer address from register $4, the number of characters to write from $5, and the fileID from $6. Then, it calls SysWrite to write the data to the file and stores the result in register $2.

SC_Read: Reads the buffer address from register $4, the number of characters to read from $5, and the fileID from $6. Then, it calls SysRead to read data into the buffer and returns the result in register $2.

```
OpenFileId OpenAFile(char *name) {
        int i = 0;
        while (i < 20 && OpenFileTable[i] != NULL) {
            i++;
        }
        // file table is full.
        if (i == 20) {
            //cout << "File table is full." << endl;
            return -1;
        }
        int fileDescriptor = OpenForReadWrite(name, FALSE);
        if (fileDescriptor == -1) { return -1;}
        for (int j = 0;j < 20;j++){
            if (NameTable[j] != NULL && !strcmp(NameTable[j], name)){
                //cout << "File table is duplicated." << endl;
                return -1;
            }
        }
        //cout << "File with id : " << i << " is " << fileDescriptor << endl;
        OpenFileTable[i] = new OpenFile(fileDescriptor);
        NameTable[i] = name;
        return i;
    }

int CloseFile(OpenFileId id){
    if(id < 0 || id >= 20 || OpenFileTable[id] == NULL) return -1;
    delete OpenFileTable[id];
    OpenFileTable[id] = NULL;
    NameTable[id] = NULL;
    return 1;
}



int WriteFile(char *buffer, int size, OpenFileId id){
   if(id < 0 || id >= 20 || OpenFileTable[id] == NULL) return -1;
   if (OpenFileTable[id] == NULL){
       return -1;
   }
   return OpenFileTable[id]->Write(buffer, size);
}
int ReadFile(char *buffer, int size, OpenFileId id){
   if (id < 0 || id >= 20 || OpenFileTable[id] == NULL) return -1;
   int bytesRead = OpenFileTable[id]->Read(buffer, size);
   if (bytesRead >= 0) return bytesRead;
   else return -1;
}
```

OpenAFile:

Check if any element in the table is available.
Check if the file has already opened.
Insert the file to the table by its index(file descriptor).

CloseFile:

Check if the file exists.
Remove the name in the opened file list.
Release element with idx in OpenFileTable.

WriteFile:

Check if the file exists.
Use the Write function in class OpenFile to write files.

ReadFile:

Check if the file exists.
Use the Read function in class ReadFile to read files.

Difficulties encountered:

陶威綸:

1. Encountering Difficulties during trace code. Since seeing the Makefile, .S file for the first time, I have used lots of time to understand the cross relations.
2. Some of the parameters passing methods are not declared in the source code, (ex. how parameters pass to register 4, 5, 6). This makes me frustrated on understanding the code deeply.

詹振暘:

Trace code is the hardest part in the beginning since I can't understand how start.s link the file, and the meaning of the code inside it. When coding “Close File” part, I’ve encountered several issues with passing parameters through functions and spending a lot of time debugging. Some method

Verification:

```
[os24team51@localhost test]$ ../build.linux/nachos -e fileIO_test1
fileIO_test1
File with id : 0 is 6
Success on creating file1.test

[os24team51@localhost test]$ ../build.linux/nachos -e fileIO_test2
fileIO_test2
File with id : 0 is 6
Passed! ^_^
```

The line(file with id : 0 is 6) is our code for debugging.

FeedBack:

陶威綸:

Give us the answer of trace code before or after submission, or we may misunderstand the purpose of the function.

詹振暘:

If TA can provide some test files for checking boundary conditions will be great.