CS342301: Operating System
MP1: System Call

Team no. 46

Member Contribution:

陳敬中: Trace Code, Report, Implementation(Open, Write)

白宸安: Trace Code, Report, Implementation(Read, Close)

MP1

Part I: Trace code

(a) SC_Halt

In the halt.c user program, the system call Halt() is called.

```
int main() {
  Halt();   // system call
}
```

As soon as halt.c starts execution, the function Machine::Run() is called by the kernel to simulate the execution of a user-level program on NachOS. Run() explicitly sets the mode of the machine to the User mode, so the user program can be executed. Then Run() calls OneInstruction() and instr is passed to store the new instruction.

```
// mipssim.cc
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);         // set to user mode
    for (;;) {
    DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "== Tick " << kernel->stats->totalTicks << " ==");
        OneInstruction(instr);                      // execute an instruction
    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();
    }
}
```

In OneInstruction(), the 4-byte binary instruction is first fetched from the program counter and stored in “instr→value”. Then, it is decoded. Afterwards, it is executed according to its “opCode". In this case, the instruction is a system call (Halt()), corresponding to OP_SYSCALL, so we call RaiseException() to handle the system call.

```
// mipssim.cc
void Machine::OneInstruction(Instruction *instr) {
    ...
    // Fetch instruction
    if (!ReadMem(registers[PCReg], 4, &raw))
    return;     // exception occurred
    instr->value = raw;
    // decode instruction
    instr->Decode();
    // execute instruction
    switch (instr->opCode) {
    // ...
    case OP_SYSCALL:
        DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallException, 0), " << kernel->stats->totalTicks);
        RaiseException(SyscallException, 0);
        return;
    // ...
    }
}
```

In RaiseException(), the exception type is passed in by “which.” Then, the machine is set to System Mode (Kernel Mode), and then the ExceptionHandler(which) is called to handle the exception indicated by “which.” After, returning from the ExceptionHandler(), the machine is set to User Mode.

```
// machine.cc
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);
}
```

In the ExceptionHandler(), the exception is handled according to the exception type, in this case the system call exception. Then, it identifies the type of system call by “type”, which is SC_Halt in this case. Therefore, it calls the SysHalt() kernel interface to deal with the exception.

```
// Exception.cc
void ExceptionHandler(ExceptionType which) {
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    // ...
    switch (which) {
        case SyscallException:
            switch(type) {
                case SC_Halt:
                    DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
                    SysHalt();
                    cout<<"in exception\n";
                    ASSERTNOTREACHED();
                    break;
                // ...
            }
        // ...
    }
}
```

The Halt() interrupt handler is then called through the SysHalt() interface.

```
// ksyscall.h
void SysHalt() {
  kernel->interrupt->Halt();
}
```

In Interrupt::Halt(), some information and the status of the kernel stats are printed. Then, the kernel is deleted, and the machine halts.

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

(b) SC_Create

This time we are to create a file called file0.test. If the creation is successful, a success message is printed. Then, the machine halts.

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

In this case, the type of system call is SC_Create. First, the filename is read from the memory address specified by Register 4. Then, the SysCreate() kernel interface is called with the filename passed through. The status of the file creation is returned from the interface and written to Register 2. Then, the machine moves on to the next instruction.

```
// Exception.cc
void ExceptionHandler(ExceptionType which) {
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    // ...
    switch (which) {
        case SyscallException:
            switch(type) {
                case SC_Create:
                    val = kernel->machine->ReadRegister(4);
                    {
                    char *filename = &(kernel->machine->mainMemory[val]); // read filename
                    status = SysCreate(filename);                      // Create file
                    kernel->machine->WriteRegister(2, (int) status);      // return status
                    }
                    // 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;
                // ...
            }
        // ...
    }
}
```

In SysCreate(), kernel calls the Create() file system handler for the stub file system to create the file with the specified filename and return the status of file creation.

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

The stub file system implements file system calls as calls to UNIX, so Create() calls the system dependent interface OpenForWrite() to create a file and return the file descriptor. If the file is successfully created, the file is closed for further uses.

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

(c) SC_PrintInt

In add.c, the function call PrintInt() triggers a system call exception.

```
int main() {
    // ...
    PrintInt(result);
    // ...
}
```

The type of system call is SC_PrintInt. The ExceptionHandler() first reads from the register 4 the integer to be printed. Then, it calls the SysPrintInt() system call interface to print the number. It also updates the PC value before returning.

```
// Exception.cc
void ExceptionHandler(ExceptionType which) {
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    // ...
    switch (which) {
        case SyscallException:
            switch(type) {
                case SC_PrintInt:
                    val=kernel->machine->ReadRegister(4);    // read an integer
                    SysPrintInt(val);                        // display value
                    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;
                    // ...
                // ...
            }
        // ...
    }
}
```

In SysPrintInt(), the kernel calls PutInt(val) to invoke a synchronous output routine to write the value to the console.

```
// ksyscall.h
void SysPrintInt(int val) {
  kernel->synchConsoleOut->PutInt(val);
}
```

For integer, NachOS doesn't print the number directly but converts it into a string and prints each char in the string instead. Thus, in PutInt(), the integer "value" is first converted into a string "str." Then, it attempts to acquire a lock to prevent other threads from outputting at the same time. Once the lock is available, PutInt() consumes the lock and starts printing "str" char by char through hardware simulation consoleOutput→PutChar(). After a char is written, WaitFor→P() is called to wait for a callback from the SynchConsoleOutput::CallBack().

```
// synchconsole.cc
void SynchConsoleOutput::PutInt(int value) {
    char str[15];
    int idx=0;
    sprintf(str, "%d\n\0", value);              // convert value into string
    lock->Acquire();                            // acquire lock
    do{
        consoleOutput->PutChar(str[idx]);       // hardware simulation of console write
        idx++;                                  // moves to next char
        waitFor->P();                           // wait for callback
    } while (str[idx] != '\0');                 // end of output
    lock->Release();                            // release the lock
}
```

In ConsoleOutput::PutChar(), it first ensures there is no other PutChar() operation in progress. Then, the char "ch" is written to the console (stdout). Then, putBusy is set to true to indicate that our operation is executing. Lastly, call the kernel to schedule the next PutChar() after "ConsoleTime".

```
// console.cc
void ConsoleOutput::PutChar(char ch) {
    ASSERT(putBusy == FALSE);                    // ensure no putchar
    WriteFile(writeFileNo, &ch, sizeof(char));   // display to console
    putBusy = TRUE;                              // indicate this putchar operation in progress
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
                                                 // schedule next PutChar to call back
}
```

Schedule() schedules the CPU to handle the interrupt at time “when” and inserts the interrupt to a pending list sorted by the scheduled time. In this case, the console write interrupt is scheduled 100 ticks from now.

```
// interrupt.cc
void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type) {
    int when = kernel->stats->totalTicks + fromNow;  // when to interrupt
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
    // ...
    pending->Insert(toOccur);                        // put in pending list
}
```

Every clock tick, an interrupt OneTick() is called to check for any pending interrupts. OneTick() disables the interrupt handler first. Then, it calls CheckIfDue() to check for pending interrupts. After returning from CheckIfDue(), interrupt handler is re-enabled.

```
// interrupt.cc
void Interrupt::OneTick() {
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;
    // ...
    // 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
    // ...
}
```

CheckIfDue() checks for pending interrupts. If there is none, return false. If there is at least one, remove the first one and call its interrupt handler. If there are more interrupts scheduled to be dealt with, CheckIfDue() handles them, too.

```
bool Interrupt::CheckIfDue(bool advanceClock) {
    PendingInterrupt *next;
    Statistics *stats = kernel->stats;
    // ...
    if (pending->IsEmpty()) {     // no pending interrupts
        return FALSE;
    }
    next = pending->Front();
    // ...
    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;
}
```

When the next PutChar() operation is due, its callback function ConsoleOutput::CallBack() is first invoked. “putBusy” is set for the next put operation and then it calls the another callback function SynchConsoleOutput::CallBack().

```
// console.cc
void ConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    putBusy = FALSE;                           // get ready for next put
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();                  // callback to SynchConsoleOutput
}
```

In SynchConsoleOutput::CallBack(), the waiting interrupt is waken and put into the ready queue.

```
void
SynchConsoleOutput::CallBack()
{
    DEBUG(dbgTraCode, "In SynchConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    waitFor->V();             // put PutChar in ready queue and calls back
}
```

Requirement (b):

The arguments of system call in user program will be stored in registers in following way:

r2	return value
r4	arg1
r5	arg2
r6	arg3
r7	arg4

When the ExceptionHandler() function handles the system call, it reads the arguments from the registers.

For example,

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

It reads r2 to know the type of the system call.

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

It reads r4 to get arg1 from the user program.

If we want to pass an object like a string, we can pass the memory address of the object as an argument like the following example:

```
char *filename = &(kernel->machine->mainMemory[val]);
```

and read the actual string from the memory.

For each case:

1. SC_Halt

user program :

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

ExceptionHandler() :

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

Halt() doesn't have any argument, so it only passes the system call code to r2, and the ExceptionHandler() handle the type using switch case.

2. SC_Create

user program :

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

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);
    }
    // ...
    return;
    ASSERTNOTREACHED();
break;
```

Create("file0.test") has 1 argument “file0.test”, so it passes the system call code to r2, and the memory location(int) of “file0.test” to r4.

ExceptionHandler() then reads r4 to get the int val.

Lastly, it reads the string from mainMemory[val].

By doing so, the argument “file0.test” is passed from the user program to kernel.

3. SC_PrintInt

user program :

```
#include "syscall.h"
int
main()
{
  int result;

  result = Add(42, 23);
  PrintInt(result);
  MSG("add~~~~");
  Halt();
  /* not reached */
}
```

ExceptionHandler() :

```
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
    // ...
return;
```

PrintInt(result) has 1 argument result, so it passes the system call code to r2, and the result(int) to r4.

ExceptionHandler() then reads r4 to get the int val, which is the result from user program PrintInt(result).

By doing so, the argument result is passed from user program to kernel.