Operating System MP1

Report

Team 47

2020/10/25

Team Members:

Both members participated in discussion and worked on the implementation together, and have decided to distribute the report part as listed.

廖允寬 (Liao) 106034071 - Worked on SC_PrintInt from part I, and explained the implementations on part II.

洪紫珈 (Evelyn) 107062174 - Worked on SC_Halt and SC_Create from part I, and wrote the difficulties and challenges faced throughout MP1.

PART I

Explain the purposes and details of each function call listed in the code path above.
Explain how the arguments of system calls are passed from user program to kernel in each of the above use cases.

A. SC_Halt

1. Machine::Run()

   This function is called by the kernel when the program starts. After setting the status to 'usermode', it uses a for(;;) loop to run through all the instructions of the user one by one with a 'OneInstruction()' function and updates the time by 'OneTick()'. It is important to note that it is a reentrant function, implying that it can be shared by many processes at the same time and can be interrupted by another process to restart the function.

2. Machine::OneInstruction()

   This function executes an instruction from the user. When Run() calls 'OneInstruction()', the address of an empty instruction set is passed as a parameter. Then, it fetches an instruction from the memory and decodes them into assembly code. If any exception or interrupt occurred, it will call 'ExceptionHandler()' and return to 'Run()' afterward.

3. Machine::RaiseException()

   During the execution of the 'OneInstruction()', if a system call is invoked or an exception occurred, it calls 'RaiseException' and passes both the exception type and the virtual address that caused the trap. At this moment, it first waited for the in-progress threads to complete their operations. Then, it changes the status to 'SystemMode' for handling the exception and changes the status back when the exception is handled.

4. ExceptionHandler()

   During the 'RaiseException()', the exception type is passed to this function (ExceptionHandler()). An exception can be caused by a system call or by unexpected user operation, it will execute accordingly if it is due to any system call and stores the result back into register 2. In most exception types, before returning to 'RaiseException()', it

updates the program counter so that system calls are not produced repeatedly. Here, the system call's exception type is SC_Halt.

5. SysHalt()

   This is a kernel interface for system calls, meaning that it manages the kernel to deal with halt().

6. Interrupt::Halt()

   This 'Halt()' will shut down the Nachos completely by deleting the kernel. This function does not return upon completion.

Halting Process:

When a .c file is executed by the user, it creates a kernel that initializes all the values and spaces for the user. Run(), then, is being executed to complete all the instructions, and oneInstruction() is called for every instruction existing in the user-level program. Whenever an exception, either by user or system, occurs, it calls raiseException() to change the current status from userMode to systemMode. Keep on with exceptionHandler that deals with all types of exceptions. For example, if halt.c was executed, it creates a SC_Halt exception and that simply calls sysHalt and Halt() to delete the kernel and ends the simulation.

B. SC_Create

1. ExceptionHandler()

   During the 'RaiseException()', the exception type is passed to the current function (ExceptionHandler()). Here, the exception is caused by a system call named SC_Create. This function calls SysCreate with a filename it read from the main memory.

2. SysCreate()

SysCreate is a function in the interface that manages the kernel to deal with SysCreate(). This function calls a bool function ‘Create()’ and returns whether the file is successfully created or not (1 = success, 0 = fail).

3. FileSystem::Create()

   It uses the name passed from ‘SysCreate()’ and creates the file with given name. It returns 1 if everything goes fine and returns 0 if something goes wrong. The possible errors can be that the file is already created, no free space for file header, no free entry for file in directory, or no free space for data blocks for the file.

Creating Process:

With createFile.c, this user program calls Create() that creates a syscall of SC_Create. This, again, is done by exceptionHandler(), and thus, uses sysCreate in the kernel interface to call Create(). Here, a file creating process is done.

C. SC_PrintInt

1. ExceptionHandler()

   ExceptionHandler是nochos進入kernel的入口，會根據在r2的type去判斷應該要處裡哪種system call exception，執行對應的動作後增加program counter，避免一直停留在同一個system call。

2. SysPrintInt()

   SysPrintInt()是一個kernel interface，負責處理system call，這裡呼叫kernel的PutInt()，並把val傳入。

3. SynchConsoleOutput::PutInt()
   sprintf會先把int value轉成字元陣列存入str，用lock->Acquire()確保一次只有一個

writer，接著用PutChar一個一個把str[idx]傳到display，waitFor->P()會等待顯示好一個字元後再繼續，當全部的str都傳到display後會lock->Release()，讓lock free。

4. SynchConsoleOutput::PutChar()
   用lock->Acquire()確保一次只有一個writer，執行consoleOutput->PutChar()將字元顯示到display，waitFor->P()等待顯示好一個字元，lock->Release()，讓lock free。

5. ConsoleOutput::PutChar()

   把一個字元寫到simulated display，然後在未來的某個時間安排interrupt發生。

6. Interrupt::Schedule()

   計算出要interrupt的時間，把interrupt時要呼叫的物件插入一個sorted list

7. Machine::Run()
   This function is called by the kernel when the program starts. After setting the status to 'usermode', it uses a for(;;) loop to run through all the instructions of the user by 'OneInstruction()' and updates the time by 'OneTick()'. It is important to note that it is a reentrant function, meaning that it can be shared by many processes at the same time and can be interrupted by another process to start the function again.

8. Machine::OneTick()
   根據不同的mode去增加totaltick，ChckifDue()會檢查是否有interrupt要執行，ChangeLevel負責disable或enable interrupt，最後檢查是否需要做context switch。

9. Interrupt::CheckIfDue()
   先檢查是否有pending interrupts，如果有的話就會增加tick到下一個interrupt該執行的時候，呼叫interrupt handler處理，把所有該時間點之前該處理的interrupt都結束並且返回。

10. ConsoleOutput::CallBack()
    Simulator calls this when the next character can be output to the display.

11. SynchConsoleOutput::CallBack()
    Call waitFor->V() to release the current thread.

Printing Process:

當User program呼叫system calls，如上面的PrintInt()，會由Exception handler針對不同type(SC_PrintInt)的exception去處理，在這邊會經由SysPrintInt()去呼叫kernel->synchConsoleOut->PutInt(val)，進到PutInt()後會把整數轉為字元陣列，PutChar負責把每個字元顯示到display上面，就完成PrintInt了。

PART II

Explain your implementation.

Revised files:

- ➔ start.S

  我們發現在這邊少了read、write、open、close的實作，參考其他指令的實作方法後，把上述缺少的四個都補齊。

- ➔ syscall.h

  移除掉 #define SC_Open、SC_Read、SC_Write、SC_Close的註解，使其在exception.cc實作時可以拿來使用。

- ➔ ksyscall.h/exception.cc

  1. SC_Open:

  參考SC_Create的作法，先從r4中取得檔案所在的記憶體位置，接著到記憶體去取得檔案名稱，把檔案名稱傳入對應的kernel interface，這裡是傳入SysOpen，將執行結果寫回r2後增加program counter。在ksyscall.h中針對SC_Open去實作它的kernel interface，把open的工作交給filesystem裡的OpenAFile去執行。

```
val = kernel->machine->ReadRegister(4);
char *filename = &(kernel->machine->mainMemory[val]);
fileID = SysOpen(filename);
```

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

2.SC_Read:

從r4取得想讀取資料的檔案的記憶體位置，從r5取得字串的size，從r6取得file id，接著到memory中取得buffer，接著把這些資料傳入對應的kernel interface，這裡是傳入SysRead，將執行結果寫回r2後增加program counter。在ksyscall.h中針對SC_Read去實作它的kernel interface，把read的工作交給filesystem裡的ReadFile去執行。

```
numChar = kernel->machine->ReadRegister(4);
val = kernel->machine->ReadRegister(5);
fileID = kernel->machine->ReadRegister(6);
buffer = &(kernel->machine->mainMemory[numChar]);
status = SysRead(buffer, val, fileID);
```

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

3.SC_Write:

從r4取得想寫入資料的檔案的記憶體位置，從r5取得字串的size，從r6取得file id，接著到memory中取得buffer，接著把這些資料傳入對應的kernel interface，這裡是傳入SysWrite，將執行結果寫回r2後增加program counter。在ksyscall.h中針對SC_Write去實作它的kernel interface，把write的工作交給filesystem裡的WriteFile去執行。

```
numChar = kernel->machine->ReadRegister(4);
val = kernel->machine->ReadRegister(5);
fileID = kernel->machine->ReadRegister(6);
buffer = &(kernel->machine->mainMemory[numChar]);
status = SysWrite(buffer, val, fileID);
```

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

4.SC_Close:

從r4取得欲關掉檔案的id，傳入對應的kernel interface，這裡是傳入SysClose，將執行結果寫回r2後增加program counter。在ksyscall.h中針對SC_Close去實作它的kernel interface，把close的工作交給filesystem裡的CloseFile去執行。

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

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

➔ filesys.h

1.OpenFileId OpenAFile(char *name)

用for loop找到一個為NULL的OpenFileTable後，call FileSystem裡的Open，傳入name執行開檔的動作，把結果存到OpenFileTable[i]中，若開檔成功就return i，若開檔失敗或沒有找到NULL的OpenFileTable的話就return -1，表示開檔失敗。

```
OpenFileTable[i] = Open(name);
```

2.int WriteFile(char *buffer, int size, OpenFileId id)

先確認id是否在0~20的範圍內、size大小是否非負、OpenFileTable[id]內是否有值，否則就return -1，表示寫檔失敗。寫檔時，用OpenFile裡的Write函式傳入buffer跟size。

```
int nums = OpenFileTable[id]->Write(buffer, size);
```

3.int ReadFile(char *buffer, int size, OpenFileId id)

先確認id是否在0~20的範圍內、size大小是否非負、OpenFileTable[id]內是否有值，否則就return -1，表示讀檔失敗。寫檔時，用OpenFile裡的Read函式傳入buffer跟size。

```
int nums = OpenFileTable[id]->Read(buffer, size);
```

4.int CloseFile(OpenFileId id)

先確認id是否在0~20的範圍內、OpenFileTable[id]內是否有值，否則就return -1，表示關檔失敗。關檔時，呼叫OpenFile解構子，並把OpenFileTable[id]設為NULL。

```
OpenFileTable[id]->~OpenFile();
```

```
OpenFileTable[id] = NULL;
```

PART III

What difficulties did you encounter when implementing this assignment?

Throughout this assignment, both of us have worked together so we can discuss and understand the code in a virtual operating system. We both found it hard on code tracing and understanding the code. In order to examine the code in detail, it is necessary to go through nearly all files and refer back to the definition and the declaration of the variables once we see something unrecognized. I, Evelyn, first had a hard time using MobaXterm, so my partner, Liao, introduced me to VS code and taught me how to utilize all the tools including the use of terminal and the definition seeking.

Moreover, on the implementation part of the assignment, we find it difficult on revising the code. For example, what variables are to be used, the purpose program counter, and why separating so many layers from the user to the kernel. However, we have worked out the problems together by going through other implemented code sections and referring back to the slides in the class.

Still, we are still confused about how readRegister works, it is abstract for us to understand what is returned from calling such a function. Does it read all types of data? How does it differentiate the data type? If the function type of readRegister is ‘int’, then how does it retrieve characters from the register?

PART IV

Any feedback you would like to let us know.

Personally, I think the instructions in the spec are clear and easy to understand, making it easy for students to work through the assignment. Not to mention, this assignment is extremely helpful for students to understand the functions calls, the program counter, the system calls, the exception handling, and definitely passing the instruction from the user to the kernel.