**MP1\_report\_109000109**

1. Trace code – SC\_Halt
   1. Machine::Run( )

當Execute 時，系統會將 Instruction Name 吃進 Execute( )

kernel->machine->Run();

並觸發Run( )

OneInstruction(instr);

接著觸發Run( ) 的函式裡的OneInstruction( )

* 1. Machine::OneInstruction( )

OneInstruction( ) 裡是執行逐條指令的過程，當發現 Syscall 時會觸發 RaiseException( )

RaiseException(OverflowException, 0);

* 1. Machine::RaiseException( )

RaiseException( ) 將Syscall 傳入 ExceptionHandler( ) 中ExceptionHandler(which);

* 1. ExceptionHandler( )

ExceptionHandler( ) 會根據 Syscall 的類型判斷該執行的事，而 SC\_Halt 則在 SysHalt( ) 中執行。

* 1. SysHalt( )

case SC\_MSG:

DEBUG(dbgSys, "Message received.\n");

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

{

char \*msg = &(kernel->machine->mainMemory[val]);

cout << msg << endl;

}

SysHalt();

ASSERTNOTREACHED();

break;

* 1. Interrupt::Halt( )

void SysHalt()

{

kernel->interrupt->Halt();

}

在 Halt( ) 中可以發現 kernel被 delete。

void Interrupt::Halt() {

cout << "Machine halting!\n\n";

cout << "This is halt\n";

kernel->stats->Print();

delete kernel; // Never returns.

}

1. Trace code – SC\_Create
   1. 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);

return;

ASSERTNOTREACHED();

break;

判斷傳入的 Exception Type ，並在 SysCreate( ) 處理 SC\_Create 的相關內容。

* 1. SysCreate( )

int SysCreate(char \*filename)

{

// return value

// 1: success

// 0: failed

return kernel->fileSystem->Create(filename);

}

Create( ) 在 filesystem 裡已經有 define

* 1. FileSystem::Create( )

FileSystem::Create(char \*name, int initialSize)

{

Directory \*directory;

PersistentBitmap \*freeMap;

FileHeader \*hdr;

int sector;

bool success;

DEBUG(dbgFile, "Creating file " << name << " size " << initialSize);

directory = new Directory(NumDirEntries);

directory->FetchFrom(directoryFile);

if (directory->Find(name) != -1)

success = FALSE; // file is already in directory

else {

freeMap = new PersistentBitmap(freeMapFile,NumSectors);

sector = freeMap->FindAndSet(); // find a sector to hold the file header

if (sector == -1)

success = FALSE; // no free block for file header

else if (!directory->Add(name, sector))

success = FALSE; // no space in directory

else {

hdr = new FileHeader;

if (!hdr->Allocate(freeMap, initialSize))

success = FALSE; // no space on disk for data

else {

success = TRUE;

// everthing worked, flush all changes back to disk

hdr->WriteBack(sector);

directory->WriteBack(directoryFile);

freeMap->WriteBack(freeMapFile);

}

delete hdr;

}

delete freeMap;

}

delete directory;

return success;

}

1. Trace code – SC\_PrintInt
   1. ExceptionHandler( )

首先看到的一樣是 ExceptionHandler( ) 裡的：

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;

SysPrintInt( ) 會進入具體執行SC\_PrintInt 的地方。

* 1. SysPrintInt( )

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);

}

進入SynchConsoleOutput::PutInt()

* 1. SynchConsoleOutput::PutInt()

SynchConsoleOutput::PutInt(int value)

{

char str[15];

int idx=0;

//sprintf(str, "%d\n\0", value); the true one

sprintf(str, "%d\n\0", value); //simply for trace code

lock->Acquire();

do{

DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, into consoleOutput->PutChar, " << kernel->stats->totalTicks);

consoleOutput->PutChar(str[idx]);

DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return from consoleOutput->PutChar, " << kernel->stats->totalTicks);

idx++;

DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, into waitFor->P(), " << kernel->stats->totalTicks);

waitFor->P();

DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return form waitFor->P(), " << kernel->stats->totalTicks);

} while (str[idx] != '\0');

lock->Release();

}

將 Syscall 參數 儲存至 str，並且使用 Lock 機制，一次只會有一個物件被鎖定。

* 1. SynchConsoleOutput::PutChar( )

void

SynchConsoleOutput::PutChar(char ch)

{

lock->Acquire();

consoleOutput->PutChar(ch);

waitFor->P();

lock->Release();

}

* 1. ConsoleOutput::PutChar( )

void

ConsoleOutput::PutChar(char ch)

{

ASSERT(putBusy == FALSE);

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

putBusy = TRUE;

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

}

和Int 一樣，但參數是 char。

* 1. Interrupt::Schedule( )

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);

}

To call是 interrupt 將要執行的功能或對象。

From now是指一段時間。

* 1. Machine:Run( )

執行完 OneInstruction( ) 之後會進到 OneTick( )。

* 1. Interrupt::OneTick( )

讓系統前往下一個Tick，像是clock的機制，能讓系統有中斷點。

* 1. Interrupt::CheckIfDue( )

檢察系統是否有如期執行instruction，當所有interrupt執行結束，return true。

* 1. ConsoleOutput::CallBack( )

void

ConsoleOutput::CallBack()

{

DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);

putBusy = FALSE;

kernel->stats->numConsoleCharsWritten++;

callWhenDone->CallBack();

}

當一個字元輸出時，會呼叫此函式。

* 1. SynchConsoleOutput:: CallBack( )

void

SynchConsoleOutput::CallBack()

{

DEBUG(dbgTraCode, "In SynchConsoleOutput::CallBack(), " << kernel->stats->totalTicks);

waitFor->V();

}

調用interrupt 顯示下一個字元。

1. Implement four I/O system calls in NachOS

一張含有 文字 的圖片

自動產生的描述

需要更改的有這些 file ：

1. test/start.S

一張含有 文字 的圖片

自動產生的描述

借鏡add 的寫法，starts.s會從 $2 裡面讀取暫存器的值，並呼叫對應的syscall ，而Register 4, 5, 6, 7則依序儲存4個參數分別是($a0, $a1, $a2, $a3)

1. userprog/syscall.h

把define取消註解

1. userprog/exception.cc

有四個部分需要實作，以open為例，

val = kernel->machine->ReadRegister(4); //取出$4的值

char \*filename = &(kernel->machine->mainMemory[val]); // 讓記憶體儲存 filename的pointer

status = SysOpen(filename); // 回傳執行狀況

1. userprog/ksyscall.h

間接呼叫，模擬對kernel進行操作。

一張含有 文字 的圖片

自動產生的描述

1. filesys/filesys.h

一張含有 文字 的圖片

自動產生的描述

* **Must handle invalid file open requests, including the non-existent file, exceeding opened file limit (at most 20 files)**

1. Test
   1. Make

一張含有 文字 的圖片

自動產生的描述

* 1. Teat1

一張含有 文字 的圖片

自動產生的描述

* 1. Teat2

一張含有 文字 的圖片

自動產生的描述

1. 心得：

NachOS 的code真的不少，需要花蠻多時間看完，但是並沒有想像中難，他使用c模擬出整個OS的環境真的很神奇，也讓我對整個系統和各個系統組成都更有概念，另外我覺得比較困難的是透過vim編輯，真的容易出錯，也不好發現錯誤。