MP2_report_37

Team Member & Contributions

- ### 資應碩二 **107065522** 陳子潔
- ### 數學大四 **105021127** 徐迺茜

工作項目	分工
Trace Code	陳子潔 & 徐迺茜
報告撰寫 (Part I & II)	陳子潔
功能實作	陳子潔
功能測試	徐迺茜
報告撰寫 (Part III)	陳子潔 & 徐迺茜

1. Trace code

- Starting from “threads/kernel.cc **Kernel::ExecAll()**”, “threads/thread.cc **thread::Sleep**”, until “machine/mipssim.cc **Machine::Run()**” is called for executing the first instruction from the user program

threads/kernel.cc Kernel::ExecAll()

- 首先簡單摘錄Thread在NachOS裡面的資料結構

```
class Thread {
  public:
    Thread(char* debugName, int threadID);
    // Make thread run (*func)(arg)
    void Fork(VoidFunctionPtr func, void *arg);
    // Relinquish the CPU if any other thread is runnable
    void Yield();
    // Put the thread to sleep and relinquish the processor
    void Sleep(bool finishing);
    // Startup code for the thread
    void Begin();
    // The thread is done executing
    void Finish();
    void setStatus(ThreadStatus st) { status = st; }
    ThreadStatus getStatus() { return (status); }
    char* getName() { return (name); }
    int getID() { return (ID); }
    // save user-level register state
    void SaveUserState();
    // restore user-level register state
    void RestoreUserState();
    // User code this thread is running.
    AddrSpace *space;
};

  private:
    // the current stack pointer
    int *stackTop;
    // all registers except for stackTop
    // J: 這邊就是Kernel Registers States的樣子
    void *machineState[MachineStateSize];
    int *stack;
    ThreadStatus status;       // ready, running or blocked
    char* name;
    int   ID;
    // Allocate a stack for thread. Used internally by Fork()
    void StackAllocate(VoidFunctionPtr func, void *arg);
    // user-level CPU register state
    int userRegisters[NumTotalRegs];
```

- 從這邊大概可以知道NachOS的Thread執行有以下幾點要注意:

    1. A thread running a user program actually has **two** sets of CPU registers – one for its state while executing **user code**, one for its state while executing **kernel code**.
    2. 每個Thread除了有自己的Register Sets外，也有AddrSpace，其中宣告了TranslationEntry (類似VMM的角色)，而本次作業的pageTable也是在AddrSpace中實作
    3. 一個Thread要執行時 (暫不考慮Context Switch)，須完成以下幾個程序:
        1. InitRegisters(); // set the initial register values
        2. RestoreState(); // load page table register
        3. kernel->machine->Run(); // jump to the user progam

- (參見AddrSpace::Execute(char* fileName))

- 接著從Exec這個子函數開始追蹤
- 簡單來說，要執行一個程式，依序:
  1. 創造一條Thread
  2. 賦予他一個定址空間 (AddrSpace)
  3. 透過Fork載入真正要執行的程式碼
  4. 將記錄Thread數量的變數+1

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]
    threadNum++;

    return threadNum-1;
}
```

- 此外，觀察傳入Fork的(FuncPtr) &ForkExecute
- 可發現此函式會呼叫addrspace.cc裡面的Load函式，將要執行的程式載入Memory中

```
void ForkExecute(Thread *t)
{
        if ( !t->space->Load(t->getName()) ) {
        return;             // executable not found
    }

    t->space->Execute(t->getName());
}
```

- 最後呼叫addrspace.c::Execute
- 這邊會將目前執行緒的定址空間與caller link起來
- 接著初始化user registers
- 並載入這個程式所對應的page table
- 呼叫machine-Run來模擬程式執行

```
void
AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;

    this->InitRegisters();     // set the initial register values
    this->RestoreState();      // load page table register

    kernel->machine->Run();    // jump to the user progam

    ASSERTNOTREACHED();        // machine->Run never returns;
                               // the address space exits
                               // by doing the syscall "exit"
}
```

- 而我們再深入追蹤Fork可以發現，這邊又做了幾件事情:
  1. Allocate a stack
  2. Initialize the stack so that a call to SWITCH will cause it to run the procedure
  3. Put the thread on the ready queue
- StackAllocate裡面又更詳細的初始化了各種Kernel Registers (machineState)

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

- 於是到這邊，我們可以發現ExecAll就是要Main Thread(Kernel)依序去執行(Exec)所有要執行的程式(Thread)

```
void Kernel::ExecAll()
{
        for (int i=1;i<=execfileNum;i++) {
                int a = Exec(execfile[i]);
        }
        currentThread->Finish();
}
```

- 執行完所有的程式(Thread)後，呼叫Finish準備來釋放Thread的空間，這邊要注意:
  - NOTE: we can't immediately de-allocate the thread data structure or the execution stack,
  - because we're still running in the thread and we're still on the stack!
  - Instead, we tell the scheduler to call the destructor, once it is running in the context of a different thread.
- 所以其實Finish裡面又會呼叫Sleep()，來Block住目前的Thread
- 接著下一條Thread(不重要)會將剛剛執行完的Thread De-Allocate掉

threads/thread.cc thread::Sleep

- 承上，Finish在呼叫Thread的時候其實已經先Disable Interrupt了
- 迴圈判斷kernel->scheduler->FindNextToRun() (是否還有下一條Thread要跑)
  - 若有，則繼續往下跑(有可能只是要De-Allocate上一條Thread而故意創造的而已)
  - 若無，進入Idle Mode，此時會判斷是否沒有任何Interrupt跟Thread要執行了，若無，整個NachOS運作結束(Halt)。

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    status = BLOCKED;

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
            kernel->interrupt->Idle();
        }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

machine/mipssim.cc Machine::Run()

- 這邊做一點簡化，其實就是在一行一行的模擬程式(Thread)執行的解碼過程

- instr就是User Program的某一行程式碼
- OneTick就是模擬CPU Clock往前跑的情形，通常一條指令假設會讓系統前進一個Clock
- 提醒: User Program理所當然的是執行在UserMode上，有需要用到Syscall才會轉到Kernel Mode (參見MP1)

```
void
Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instructi

    kernel->interrupt->setStatus(UserMode);

    for (;;) {
      OneInstruction(instr);
      kernel->interrupt->OneTick();
   }
}
```

- 搭配前面的Kernel::ExecAll()追蹤過程，我們可大致整理出NachOS要執行一個程式的流程:
  1. New一個Thread，並做簡單初始化
  2. 再New一個AddrSpace給此Thread
  3. Thread呼叫Fork，最終目的是將欲執行的程式載入進去Thread
     1. Fork接收到funcPtr(到時候要執行的程式)
     2. 先做StackAllocate，初始化一些Thread的Stack，透過machineState[InitialPCState] = (void*)func;，讓原先的funcPtr成為未來ProgramCounter要執行的程式，
     3. 此時Thread大致初始化完畢，將Interrupt Disable
     4. 透過scheduler->ReadyToRun(this);將剛剛的Thread放入Ready Queue，將來準備讓CPU執行
     5. 重新打開Interrupt
  4. CPU scheduler未來會從Ready Queue中Load準備要執行的Thread，並讀取ProgramCounter的值

2. Implement page table in NachOS

Verification:

```
[os19team37@lsalab ~/NachOS-4.0_MP2/code/test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

- 本次作業的提示

> Hint: The following files “may” be modified:
> userprog/addrspace.*
> threads/kernel.

addrspace.h

- 根據提示，我們首先觀察addrspace.h

```
#define UserStackSize          1024
class AddrSpace {
  public:
    AddrSpace();
    ~AddrSpace();
    bool Load(char *fileName);
    void Execute(char *fileName);
    void SaveState();
    void RestoreState();
  ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode

  private:
    TranslationEntry *pageTable;
    unsigned int numPages;
    void InitRegisters();
};
```

- 可以發現AddrSpace實作了將Program Load進Memory，並且Execute的功能

- TranslationEntry 可以拿來操作程式的PageTable
- 我們在Addrsapce這個Class裡面多宣告一個共享變數，紀錄被使用過的Frame(PhysicalPages)
-
```
static bool usedPhyPage[NumPhysPages];
```

addrspace.c

- 將著來到addrspace.c裡面的Load函式
- 我們在Load裡面新增以下幾行，讓剛剛宣告的usedPhyPage派上用場
- 順便設置一下valid, use, dirty...等Virtual Memory的紀錄值

```
    pageTable = new TranslationEntry[numPages];
    for(unsigned int i = 0, j = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;
        while(j < NumPhysPages && AddrSpace::usedPhyPage[j] == true)
            j++;
        AddrSpace::usedPhyPage[j] = true;
        pageTable[i].physicalPage = j;
        pageTable[i].valid = true;
        pageTable[i].use = false;
        pageTable[i].dirty = false;
        pageTable[i].readOnly = false;
    }
```

```
executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].ph
* PageSize + (noffH.code.virtualAddr%PageSize)]),
noffH.code.size, noffH.code.inFileAddr);

executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/PageSize
* PageSize + (noffH.code.virtualAddr%PageSize)]),
noffH.initData.size, noffH.initData.inFileAddr);

executable->ReadAt(
&(kernel->machine->mainMemory[pageTable[noffH.readonlyData.virtualAddr/Page
* PageSize + (noffH.code.virtualAddr%PageSize)]),
noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
```

- 接著我們改變一下noffH.initData、noffH.code、noffH.readonlyData所讀取到的Memory地址 (因為原先並未修改到這邊，所以所有程式都讀取到同一頁**Page**，共享到不該共享的變數了**!**)
- 簡而言之，修正過後的Memory Address存取位置公式為: page base + page offset

kernel.h & kernel.c

- 根據題目要求: You must put the data structure recording used physical memory in kernel.h / kernel.c
- 不過我不確定是要把整個AddrSpace搬到Kernel.h去，還是只要把usedPhyPage般過去就好了，故本次作業就沒修改到這個檔案了...

3. Explain how NachOS creates a thread(process), load it into memory and place it into scheduling queue as requested in Part II-1 Your explanation on the functions along the code path should at least cover answer for the questions below

How Nachos initializes the memory content of a thread(process), including loading the user binary code in the memory? & How Nachos allocates the memory space for new thread(process)?

我們可大致整理出NachOS要執行一個程式的流程:

1. New一個Thread，並做簡單初始化
2. 再New一個AddrSpace給此Thread
   - addrspace的建構子當中會使用bzero（）來清除Memory
3. Thread呼叫Fork，最終目的是將欲執行的程式載入進去Thread
   1. Fork接收到ForkExecute的funcPtr(到時候要執行的程式)
   2. 接著做StackAllocate，初始化一些Thread的Stack，透過machineState[InitialPCState] = (void*)func;，讓原先的funcPtr成為未來ProgramCounter要執行的程式
   3. 此時Thread大致初始化完畢，將Interrupt Disable
   4. 透過scheduler->ReadyToRun(this);將剛剛的Thread放入Ready Queue，將來準備讓CPU執行
   5. 重新打開Interrupt
4. CPU scheduler未來會從Ready Queue中Load準備要執行的Thread，並讀取ProgramCounter的值

How Nachos creates and manages the page table?

- translate.h裡面會定義TranslationEntry，這個Class有一點類似VMM的角色，據說也能拿來當TLB用
- 接著在addrspace.h當中會定義TranslationEntry *pageTable
- 未來addrspace.c裡面實作Load函式的時候，可以操作pageTable做一些Virtual Memory相關的處理及轉譯

How Nachos translates address?

在addrspace.h與machine.h皆分別定義了Translate

> ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode);*
> *ExceptionType Translate(int virtAddr, int* physAddr, int size, bool writing)

- 由於C++支援function overloading，而我用grep -nr "Translate"查看的結果，認為應該主要還是使用translate.c裡面所實作的Translate來做address translate
- 至於Transalte函式內部所做的事情基本上就是判斷這個程式所使用的Page是否合法、size是否超過…等

How Nachos initializes the machine status (registers, etc) before running a thread(process)

- machineStates主要都是在thread.c裡面的建構子中初始化的
- 以後在Fork的時候也會呼叫StackAllocate做一些mahineStates的設定

Which object in Nachos acts the role of process control block

- 我們查看Thread.h，並觀看註解，可以發現這個Class長的很像process control block

```
// The following class defines a "thread control block"
// -- which represents a single thread of execution.
//
//  Every thread has:
//     an execution stack for activation records ("stackTop" and "stack")
//     space to save CPU registers while not running ("machineState")
//     a "status" (running/ready/blocked)
//
//  Some threads also belong to a user address space; threads
//  that only run in the kernel have a NULL address space.

class Thread {
  private:
    int *stackTop;
    void *machineState[MachineStateSize];

  public:
    Thread(char* debugName, int threadID);
    ~Thread();

    void setStatus(ThreadStatus st) { status = st; }
    ThreadStatus getStatus() { return (status); }
        char* getName() { return (name); }
        int getID() { return (ID); }

  private:
    int *stack;
    ThreadStatus status;
    char* name;
    int   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);

// A thread running a user program actually has **two** sets of CPU registe
// one for its state while executing **user code**, one for its state
// while executing **kernel code**.
    int userRegisters[NumTotalRegs];
  public:
    void SaveUserState();
    void RestoreUserState();
    AddrSpace *space;
};
```

When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?

- thread.c的Fork中，會呼叫scheduler->ReadyToRun(this)
- 此行會將已經分配好資源的Thread放入Ready Queue，以供未來CPU排班執行

Reference

1. 向 NachOS 4.0 作業進發 (1) (實作好幫手!!!) (https://morris821028.github.io/2014/05/24/lesson/hw-nachos4/?fbclid=IwAR06r7ZH28w_hDLS4-h5Yjge63SZxq2VDtv28Rpa9JKhF51jTH3RlGM1wNk)
2. OS::NachOS::HW1 (http://blog.terrynini.tw/tw/OS-NachOS-HW1/)
3. CSE120/Nachos中文教程.pdf (讚!!!) (https://github.com/zhanglizeyi/CSE120/blob/master/Nachos%E4%B8%AD%E6%96%87%E6%95%99%E7%A8%8B.pdf)
4. C++：哪些變數會自動初始化？ (https://www.itread01.com/content/1550033287.html?fbclid=IwAR1lsuTWlDjVVTe_V2ot1z7-Nf2oKj5XEsE63mdPrLQ2Bp6wlGcuxCWn9aI)
5. C/C++ 中的 static, extern 的變數 (https://medium.com/@alan81920/c-c-%E4%B8%AD%E7%9A%84-static-extern-%E7%9A%84%E8%AE%8A%E6%95%B8-9b42d000688f)
6. C 語言程式的記憶體配置概念教學 (https://blog.gtwang.org/programming/memory-layout-of-c-program/)
7. 列舉（Enumeration） (https://openhome.cc/Gossip/CppGossip/enumType.html)
8. [C++]關於Callback Function (http://gienmin.blogspot.com/2013/03/ccallback-function.html)
9. [教學]C/C++ Callback Function 用法/範例 (http://dangerlover9403.pixnet.net/blog/post/83880061-%5B%E6%95%99%E5%AD%B8%5Dc-c++-callback-function-%E7%94%A8%E6%B3%95-%E7%AF%84%E4%BE%8B-(%E5%85%A7%E5%90%ABfunctio))
10. 虛擬函式（Virtual function） (https://openhome.cc/Gossip/CppGossip/VirtualFunction.html)
11. 深入理解C++中public、protected及private用法 (https://www.jb51.net/article/54224.htm)
12. C++中this指针的理解 (https://blog.csdn.net/ljianhui/article/details/7746696)
13. UNIX v6的进程控制块proc结构体和user结构体 (https://www.suntangji.me/2017/12/18/proc%E7%BB%93%E6%9E%84%E4%BD%93%E5%92%8Cuser%E7%BB%93%E6%9E%84%E4%BD%93/)
14. 如何與 GitHub 同步筆記 (https://hackmd.io/c/tutorials-tw/%2Fs%2Flink-with-github-tw)