CS342301: Operating System
MP2: Multiprogramming
Team Number: 71

Team Members & Contributions

- Members: 108022138 楊宗諺、111062649 許峻源
- Contributions: 合力製作MP2 report、討論page table的implementation details

Trace Codes

1. **Kernel::Kernel()**

- Kernel的constructor主要是在interpret user輸入的command，根據command line argument的指令內容執行不同的動作。舉例來說，如果command內容為 -e 的話，kernel constructor會先將要執行的filename先存在定義好的execfile陣列裡面，等待Kernel::ExecAll()函數被呼叫執行。
- 相較Kernel::Initialize()函數，Kernel()會先對command作前處理，把會使用到的參數與陣列（consoleIn、consoleOut、execfile等等）先設置好，再來呼叫Initialize()函數建立main thread，並用前處理設置好的參數初始化NachOS的global data structures。

2. **Kernel::ExecAll()**

- 在設置好kernel之後，就會呼叫ExecAll()來執行user program。
- ExecAll()會根據先前設置好的execfile陣列呼叫Exec()函數來執行各個file，為每個program創建一個thread。

3. **Kernel::Exec()**

- Exec()主要在幫每個要執行的user program創建一個thread，並透過thread的constructor初始化thread的相關參數（stack、thread status等等）。另外也會透過AddrSpace::AddrSpace()來創建thread的address space。
- 接著呼叫Thread::Fork()函數載入真正要執行的程式碼，這邊把ForkExecute()的function pointer與thread的pointer當作參數傳入。

4. **AddrSpace::AddrSpace()**

- 對於uniprogramming，在創建address space的時候就會先把page table設置好，確定對應的physical address並zero out。等到呼叫AddrSpace::Load()函數的時候，Load()就會根據page table找到對應的physical address，把需要的memory contents load到指定的位置。

5. **Thread::Fork()**

- Fork()會透過Thread::StackAllocate()函數來allocate thread的stack，這邊會把參數function pointer跟thread的pointer傳入StackAllocate()裡面，為了之後呼叫Switch()函數時可以正確執行function pointer指向的function。

- 在Allocate好stack之後，會呼叫Scheduler::ReadyToRun()函數把thread丟進ready queue裡面等待執行。(Note: 呼叫Scheduler之前會先disable所有的interrupt、被丟進ready queue裡面的thread還沒有load好memory)

6. Thread::StackAllocate()

- 根據定義好的StackSize為thread創建一個bounded array for stack，並用stackTop pointer來追蹤當前stack的top的位置。
- 初始化且設值所有需要的CPU registers（machine state），這邊會把function pointer跟thread的pointer設給CPU的registers。

7. Scheduler::ReadyToRun()

- 將thread的status從just_created設置成ready，並把thread丟進ready queue裡面等待執行。

8. Thread::Finish()

- 利用Kernel::ExecAll()函數為所有要執行的program創建thread，並把threads丟進ready queue裡面之後，會呼叫Thread::Finish()函數(這邊呼叫Finish()的是current thread，也就是一開始initialize kernel時建立的main thread)。
- Finish()函數會先disable所有的interrupt，接著呼叫Thread::Sleep()函數(參數傳入True)。

9. Thread::Sleep()

- Sleep()函數會把current thread的status從running設成blocked(這邊是main thread)，並利用Scheduler::FindNextToRun()函數去ready queue裡面找下一個要被執行的thread。
- 如果ready queue裡面沒有任何thread等待被執行，那麼就會呼叫Interrupt::Idle()函數將系統status設為IdleMode並檢查有沒有interrupt要執行，沒有的話就會呼叫Halt()函數。反之，如果有thread要被執行的話，就會呼叫Scheduler::Run()函數來執行。

10. Scheduler::Run()

- Run()函數會利用switch()函數將CPU的使用權從old thread切換到next thread，而在switch之前會先儲存old thread的register states跟address space並且check old thread的stack有沒有overflow，然後把next thread的status設成running。
- 而從next thread switch回來之後，會先利用CheckToBeDestroyed()函數check說old thread是否已經執行完要被delete掉。如果仍要繼續執行old thread的話，就會restore先前儲存的register states和address space。

11. Kernel::ForkExecute()

- ForkExecute()函數會先透過AddrSpace::Load()函數將user program從disk load到physical memory。如果load成功，則會呼叫AddrSpace::Execute()函數準備執行user program。

12. AddrSpace::Load()

- Load()函數會把user program的object file轉換成noff format，並從disk load到page table所對應到的physical address。
- 對於uniprogramming，Load()函數中會根據noffH內的Size資訊，計算出program 的大小除以PageSize並RoundUp計算program所需要的page數量，並利用assert機制做memory limit的error handling（不能超過physical memory的frame數量）。

13. AddrSpace::Execute()

- 進到Execute()之後，基本上user program就要開始執行了。而在進到Machine::Run()執行第一個指令之前，需要先initialize CPU registers的值，設置好program counter register、next program counter register和stack register。另外也需要restore當前thread的page table。
- 在處理好register跟page table之後，就會進到Machine::Run()執行user program的指令。

14. Thread.h

- class thread 扮演了Nachos PCB的角色，其中包含了 name, ID, Status 等PCB應該包含的變數。

15. Thread::Translate()

- 主要功能為把Virtual Address 轉換為Physical Address。在函數中把Virtual address除以Pagesize取商數乘以PageSize，再加上Offset，計算出Physical Address 。

Verification

- 下面是我們這次實作multiprogramming的結果，可以同時執行consoleIO_test1與consoleIO_test2。

```
[os22team71@localhost 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
```

- 我們也實作了MemoryLimitException的機制，並做了一個testcase來測試。

```
[os22team71@localhost test]$ ../build.linux/nachos -e consoleIO_test4
consoleIO_test4
Unexpected user mode exception 8
Assertion failed: line 203 file ../userprog/exception.cc
Aborted
```

Implementation Details

1. Class Kernel

- 我們在class Kernel裡額外加入了兩個public data members，一個是UsedFrames陣列，size為physical memory的page數量，用來追蹤正在使用的frames，另外一個則是整數變數FreePages，用來記錄目前沒有被使用的frame數量。

```
bool UsedFrames[NumPhysPages];
unsigned int FreePages;
```

- 我們把這兩個參數的initialization放在kernel的constructor裡面。

```
FreePages = NumPhysPages;
for(int i=0; i<NumPhysPages; ++i)
        UsedFrames[i]=false;
```

2. AddrSpace::AddrSpace()、Load()

- 原先uniprogramming的page table是在呼叫AddrSpace()的時候initialize的，我們把initialize page table的動作延後到要Load memory content的時候才做。目的是為了根據program實際需要的page數量來建立page table，而不是直接利用physical memory的page數量來建立。
- page table建立好之後，為program的每個page與physical memory frame做對應，這邊我們實作是用while loop去check每個frame的使用狀況。如果有frame沒有被使用的話，就把那個沒有被使用的frame拿來用（要zero out），並set up所有相關的bits（valid bit、use bit、dirty bit與readOnly bit）。另外也需要將FreePages的數量減1。
- 並且在每次安排physical memory frame時，檢查FreePages的數量，如果小於等於0時表示此時Memory已滿，呼叫RaiseException() 發出MemoryLimitException

```
pageTable = new TranslationEntry[numPages];
for (int i = 0, phy_index=0; i < numPages; i++) {
    if(kernel->FreePages <= 0){
            machine->RaiseException(MemoryLimitException, 0);
    }
    while(phy_index < NumPhysPages && kernel->UsedFrames[phy_index] == true){
            ++phy_index;
    }
    kernel->FreePages--;
    kernel->UsedFrames[phy_index]=true;
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    pageTable[i].physicalPage = phy_index;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    bzero(&kernel->machine->mainMemory[phy_index*PageSize], PageSize);
}
```

- 接著改變Memory的存取位置，改由PageTable來對照找出Physical Address。藉由呼叫Translate()可以將Vitual Address 轉變為Physical Address，將其儲存在int addr內，而addr 內就是Memory該儲存Data的Physical Address。

```
// create a int to store the result of Translate func
    unsigned int addr;
// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
    Translate(noffH.code.virtualAddr, &addr, false);// use Translate() to calculate phy_addr
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
        &(kernel->machine->mainMemory[addr]),
            noffH.code.size, noffH.code.inFileAddr);
    }
```

```
//----------------------------------------------------------------------
// AddrSpace::Translate
//  Translate the virtual address in _vaddr_ to a physical address
//  and store the physical address in _paddr_.
//  The flag _isReadWrite_ is false (0) for read-only access; true (1)
//  for read-write access.
//  Return any exceptions caused by the address translation.
//----------------------------------------------------------------------
ExceptionType
AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite)
```

Difficulties

這次的實作雖然需要改動的file數量不多, 但是在trace code時卻蠻複雜的, 需要很了解thread, program, memory之間的互動, 才能知道需要改動的code部份, 實做時test了不少次才完成這次的作業。