Nachos Project 1: Thread Management

鄭傑鴻 R10741015

事前準備：

- 複習計算機結構中的 Virtual Address 與 Physical Address，以及 Page Table 間的轉換機制。
    - 翻看過去計算機結構課程的教科書與投影片 。
    - 參考網路上影片與教學以快速回想作法。
- 參考 Nachos 的說明及其他文件。
    - UC Berkely: C++ https://homes.cs.washington.edu/~tom/c++example/
    - Duke: A Roadmap Through Nachos https://users.cs.duke.edu/~narten/110/nachos/main/main.html
- 以 Debugger Mode 進行 Trace Code，了解整個程式的運作模式。
    1. threads/main.cc: 開始執行，若 argv 包含-d 開啟 Debug 模式，並且建立 Kernel。
    2. userprog/userkernel.cc: 針對其餘的 argv 進行 parsing，啟動 filesystem 與 machine。接著針對每一個要運行的使用者程式、新開一個 Thread(threads/thread.cc)、分派 Address Space(userprog/addrspace.cc)、最後用各 thread 的 Fork 方法開始執行。
    3. userprog/addrspace.cc: 在此物件中，Constructor 建立一個新的 pageTable 指標，其內容為此程式的 Virtual Page 與實際 RAM 的 Physical Page 的對應關係。在 Thread 呼叫 Fork()方法執行時，AddrSpace 會用 kernel 啟動的 filesystem，打開對應檔名的檔案、計算放置程式所需的空間(包含 code、data、stack 等所需的空間)、並且用 filesystem 的 ReadAt()方法，把所有需要的資料置入到 RAM 當中。

Q1: 為何結果不如預期？

觀察 nachos code 中的註解與實作方法，可以發現最初的程式僅支援單一程式運行：

參考以下最初 AddrSpace Constructor 的片段：

- 註解：表明現在僅是單一程式執行，是故只有一種未區分的 pageTable 版本。
- 實作：
    - 建立一個 pageTable 指標，用來記錄 32 頁(=NumPhysPages)的轉換關係。
    - 將 Virtual Page 從 0 到 31，依序對應到 RAM 中 Physical Page 的 0 到 31 頁：
        - **問題一：pageTable 對應到相同空間：若存在多個 Thread 同時運行，多個程式的 Virtual Page 都會對應到相同的 Physical Page，造成資料互相覆蓋而產生錯誤。**
    - Filesystem 用 kernel->machine->mainMemory[]直接去存取 code 與 data 的 Virtual Address：

- **問題二：沒有用到 pageTable：在單一程式下 Virtual Address 可以直接等同於 Physical Address。然而在多程式底下需要進行 Virtual Address 與 Physical Address 的轉換。**

- 錯誤產生流程(以 -e ./test/test1 -e ./test/test2 為例)：
  1. Userkernel 對 test1 產生 Thread，並以 AddrSpace 建立 pageTable、分配記憶體空間，以 Fork 開始執行。
  2. Userkernel 對 test2 產生 Thread，以 AddrSpace 建立相同的 pageTable、把 test2 的 code 與 data 放置到與 test1 相同的記憶體空間，以 Fork 開始執行。
  3. 程式相互覆蓋產生錯誤，印出不理想結果。

Q2: 如何解決問題？

由以上 Q1 的討論，可發現問題有兩個：程式間的 pageTable 相同、及未使用到 pageTable 兩大問題。

解決：

1. 程式間的 pageTable 相同：

   根據計算機結構的內容，各程式的 Virtual Page 對應到的 Physical Page 應要不同，避免資料覆寫。在此同時，由於 Virtual Page 與 Physical Page 為 Mapping 的關係，Physical Page 可允許不連續的空間對應。

   解決方法：

   避免覆寫：設計機制來記錄 mainMemory 各個 Physical Page 的使用情形，來確保新的程式進入並開始 Mapping 時，不會對應到已經在使用的記憶體區段。

   觀察有關記憶體的變數(如 mainMemory、PageSize 與 NumPhysPages 等)，皆定義於 machine/machine.h 及 machine/machine.cc：

   - 在 machine/machine.h 中，**對 machine 這個 class 新增 bool 指標 availPhysPages**

```
class Machine {
  public:
    Machine(bool debug);    // Initialize the simulation of the hardware
                // for running user programs
    ~Machine();         // De-allocate the data structures

// Routines callable by the Nachos kernel
    void Run();         // Run a user program

    int ReadRegister(int num);  // read the contents of a CPU register

    void WriteRegister(int num, int value);
                // store a value into a CPU register

// Data structures accessible to the Nachos kernel -- main memory and the
// page table/TLB.
//
// Note that *all* communication between the user program and the kernel
// are in terms of these data structures (plus the CPU registers).

    char *mainMemory;       // physical memory to store user program,
                // code and data, while executing


    bool* availPhysPages;
```

- 在 machine/machine.cc 中，當 machine 初始化(呼叫 constructor)時，**建立長度為 32(=NumPhysPagees)的布林陣列**，其中每個元素表示該 index 的 Physical Page 是否可被使用，並且把其中**每個元素初始化為"true"**(machine 建立後預設每頁都可被使用)

```
Machine::Machine(bool debug)
{
    int i;

    for (i = 0; i < NumTotalRegs; i++)
        registers[i] = 0;
    mainMemory = new char[MemorySize];
    for (i = 0; i < MemorySize; i++)
        mainMemory[i] = 0;

    availPhysPages = new bool[NumPhysPages];
    for(i=0; i<NumPhysPages; i++){
        availPhysPages[i]=true;
    }
```

- 在 userprog/addrspace.cc 中，在執行 Load()方法時，先計算該程式所需占用的記憶體空間，把空間轉換為所需的頁數(=numPages)，**從頁數 0 到 numPages-1，尋找空閒的 Physical Page 來 mapping，並把找到的頁數填到 pageTable 中**：
    - *for virtual page indexed i from 0 to numPages-1:*
        - *from physical page with index j =0:*
            - *j++ until we find an available page for virtual page i*
            - *Assert that we don't exceed 32 physical pages*
        - *Record Virtual Page i to Physical page j in pageTable*
        - *Record Physical page j is currently not available*
        - *i++, j++ until every Virtual Page get corresponding Physical Page*

```
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
            + UserStackSize;    // we need to increase the size
                                // to leave room for the stack
    numPages = divRoundUp(size, PageSize);
//  cout << "number of pages of " << fileName<< " is "<<numPages<<endl;
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages);       // check we're not trying
                                // to run anything too big --
                                // at least until we have
                                // virtual memory

    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

    // executable file opened, calculated its size, and got its # of pages to store
    for(int i=0, j=0; i<numPages; i++, j++){
        // Physical Page
        while(kernel->machine->availPhysPages[j]!=true){
            j++;
            ASSERT(j<=NumPhysPages);
        }
        pageTable[i].physicalPage = j;
        kernel->machine->availPhysPages[j]=false;
    }
```

- 在 userprog/addrspace 中，destructor ~AddrSpace()執行時，把此程式正在用的 Physical Page 還給 kernel->machine

- *for entry page indexed i to numPages-1:*
    - *Make the used Physical Pages available again.*

```
//----------------------------------------------------------------------
// AddrSpace::~AddrSpace
//  Dealloate an address space.
//----------------------------------------------------------------------

AddrSpace::~AddrSpace()
{
   for(int i=0; i<numPages; i++){
        //usingPhysPage[pageTable[i].physicalPage]=false;
        kernel->machine->availPhysPages[pageTable[i].physicalPage]=true;
   }
   delete pageTable;
}
```

2. 未使用到 pageTable：

   觀察現在讀檔的方式，因為 Physical Address 等同 Virtual Address，所以原來的 Code 直接拿 Virtual Address 來存取 mainMemory，並未使用到 pageTable。

   由計算機結構可知，Address 分為兩段：前方 Page Number 供 Virtual Page 與 Physical Page 進行轉換，後面 Page Offset 來指定資料在單一 Page 中的對應位置。

   參考助教的提示中，Translate.h 與 Translate.cc 的相關內容，觀察後發現其內容亦是 Virtual Address 與 Physical Address 的轉換，適用於目前的應用狀況。

   解決方法：

   由於 ReadAt()方法中，其參數為主記憶體的真實位置，故需要用 pageTable 進行轉換：

   - 在 AddrSpace 中，建立新的方法 TranslateVir2Phys():
       - *對於傳入的 Virtual Address，拆分為 Virtual Page Number 與後面的 Offset*
       - *用 pageTable 去查找 Virtual Page Number 對應的 Physical Page Number*
       - *把 Physical Page Number 乘以 Size 變成該 Physical Page 的起始位置，加上剛剛的 Offest，即為最後要求的 Physical Address*

```
int
AddrSpace::TranalateVir2Phys(int virAddr){
    int ret_physAddr = 0;
    int vpn = (unsigned) virAddr / PageSize;
    int offset = (unsigned) virAddr % PageSize;
    ret_physAddr = pageTable[vpn].physicalPage * PageSize + offset;
    return ret_physAddr;
}
```

   - 在 ReadAt()時，從直接讀 Virtual Address，改成讀翻譯完的 Physical Address。

```
// then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        int code_physAddr = TranalateVir2Phys(noffH.code.virtualAddr);
        executable->ReadAt(&(kernel->machine->mainMemory[code_physAddr]),
            noffH.code.size, noffH.code.inFileAddr);
        /*executable->ReadAt(&(kernel->machine->mainMemory[noffH.code.virtualAddr]),
            noffH.code.size, noffH.code.inFileAddr);*/
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
        DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
        int initData_physAddr = TranalateVir2Phys(noffH.initData.virtualAddr);
        executable->ReadAt(&(kernel->machine->mainMemory[initData_physAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
        /*executable->ReadAt(&(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);*/
    }
```

經由以上努力，不同程式的 Virtual Address，會對應到互不干涉的 Physical Address，並且修改了 ReadAt()方法使其得以讀到對的 Physical Address。

Q3: 修正後實驗結果

以原先出錯的./userprog/nachos -e ./test/test1 -e ./test/test2 執行：

```
Total threads number is 2
Thread /home/jch/nachos-4.0/code/test/test1 is executing.
Thread /home/jch/nachos-4.0/code/test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 300, idle 8, system 70, user 222
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
[1] + Done                       "/usr/bin/gdb" --interpreter=
```

可發現在經過以上記憶體對應的調整、即存取位置改變後，問題獲得解決。

Q4: 討論與心得

在這次作業中，我認為我遇到的困難為了解整個 nachos 的架構。一開始解壓縮完，發現整個檔案錯綜複雜，花了我很多時間去 Trace Code、查看 Nachos 相關的檔案、甚至看其他學校的上課影片以了解他設計的原理。此過程接近花了我一兩天的時間才大概了解整個架構。

為了找出程式的錯誤所在，從最一開始試圖理解其中的內容，找出不合理之處，結合 Debugger 確認出錯的原因，也花了我不少時間來回試探以點出問題所在。而在之後為了解決 Virtual Address 與 Physical Address 的轉換，也讓我需要重新回憶過去計算機結構學過的東西，結合 Code 來使其正常運作。不過整體完成後，回首才發現一切豁然開朗，甚至有跡可循，切身的體會並學習到了此作業系統與計算機結構間的實作方法。