電機所碩二

張光耀

R04921005

Operating System

Project 1 - Thread Management

1. Why the result is the congruent with expected ?

Discuss:

分別執行 test 1 與 test 2 時 結果如下：

test 1 (為遞減輸出):

```
morris@ubuntu:~/Downloads/nachos-4.0/code/userprog$ ./nachos -e ../test/test1
Total threads number is 1
Thread ../test/test1 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:6
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 200, idle 66, system 40, user 94
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
```

test 2 (為遞增輸出):

```
morris@ubuntu:~/Downloads/nachos-4.0/code/userprog$ ./nachos -e ../test/test2
Total threads number is 1
Thread ../test/test2 is executing.
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 200, idle 32, system 40, user 128
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
```

當同時執行 test 1 與 test 2 :

```
morris@ubuntu:~/Downloads/nachos-4.0/code/userprog$ ./nachos -e ../test/test1 -e
 ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../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
Print integer:7
Print integer:8
Print integer:9
Print integer:10
Print integer:12
Print integer:13
Print integer:14
Print integer:15
Print integer:16
Print integer:16
Print integer:17
Print integer:18
Print integer:19
Print integer:20
Print integer:17
Print integer:18
Print integer:19
Print integer:20
Print integer:21
Print integer:21
Print integer:23
Print integer:24
Print integer:25
return value:0
Print integer:26
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 800, idle 67, system 120, user 613
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
```

故可以得知，以這個範例來說，雖然說是同時執行，但作業系統首先會執行 test 1 (遞減輸出) ，再來 switch 到 test 2 ，在這裡可以發現，後面的結果變成遞增輸出，由此可以推斷應該是發生了程式碼片段的讀取錯誤，在 test 2 的程式碼讀入後，洗掉了原本 test 1 片段，以至於兩個程式片段都讀到相同的記憶體區塊，造成 thread 的執行錯誤。我們可以從尚未修改之addrspace.h 與 addrspace.cc 發現，的確尚未處理 mutithread

```
//----------------------------------------------------------------------
// AddrSpace::AddrSpace
//  Create an address space to run a user program.
//  Set up the translation from program memory to physical
//  memory.  For now, this is really simple (1:1), since we are
//  only uniprogramming, and we have a single unsegmented page table
//----------------------------------------------------------------------

AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    pageTable[i].physicalPage = i;
//  pageTable[i].physicalPage = 0;
    pageTable[i].valid = TRUE;
//  pageTable[i].valid = FALSE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
//    bzero(kernel->machine->mainMemory, MemorySize);
}
```

所以所有的process 都會共用到相同的physicalpage，如此一來每個process當然就執行到同一份 code ，而產生如此的錯誤，應該想辦法解決 context switch 的問題。

2. Explain all the functions that you traced in the files (Slide page 7).

userkernel.cc :

```
UserProgKernel::UserProgKernel(int argc, char **argv)
        : ThreadedKernel(argc, argv)
{
    debugUserProg = FALSE;
    execfileNum=0;
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-s") == 0) {
        debugUserProg = TRUE;
    }
    else if (strcmp(argv[i], "-e") == 0) {
        execfile[++execfileNum]= argv[++i];
    }
         else if (strcmp(argv[i], "-u") == 0) {
        cout << "===========The following argument is defined in userkernel.cc" << endl;
        cout << "Partial usage: nachos [-s]\n";
        cout << "Partial usage: nachos [-u]" << endl;
        cout << "Partial usage: nachos [-e] filename" << endl;
    }
    else if (strcmp(argv[i], "-h") == 0) {
        cout << "argument 's' is for debugging. Machine status  will be printed " << endl;
        cout << "argument 'e' is for execting file." << endl;
        cout << "atgument 'u' will print all argument usage." << endl;
        cout << "For example:" << endl;
        cout << "   ./nachos -s : Print machine status during the machine is on." << endl;
        cout << "   ./nachos -e file1 -e file2 : executing file1 and file2."  << endl;
    }
    }
}
```

這邊的 code 主要是對 user command line 的解譯，例如說 -e 等等的指令。

```
void
UserProgKernel::Run()
{

    cout << "Total threads number is " << execfileNum << endl;
    for (int n=1;n<=execfileNum;n++)
        {
        t[n] = new Thread(execfile[n]);
        t[n]->space = new AddrSpace();
        t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
        cout << "Thread " << execfile[n] << " is executing." << endl;
        }
//  Thread *t1 = new Thread(execfile[1]);
//  Thread *t1 = new Thread("../test/test1");
//  Thread *t2 = new Thread("../test/test2");

//    AddrSpace *halt = new AddrSpace();
//  t1->space = new AddrSpace();
//  t2->space = new AddrSpace();

//    halt->Execute("../test/halt");
//  t1->Fork((VoidFunctionPtr) &ForkExecute, (void *)t1);
//  t2->Fork((VoidFunctionPtr) &ForkExecute, (void *)t2);
    ThreadedKernel::Run();
//  cout << "after ThreadedKernel:Run();" << endl;  // unreachable
}
```

new thread 這邊較為重要產生新的thread ，並從中可以看到每個thread 會載入記憶體的初始位置，故可以推論 addrspace 為影響此次 project 問題的關鍵所在，如本次範例產生兩個 thread ，記錄記憶體的位置即是重點，若如果沒有統整global的記憶體位置的紀錄，就會發生context switch error的情況。

translate.h :

```
class TranslationEntry {
  public:
    unsigned int virtualPage;   // The page number in virtual memory.
    unsigned int physicalPage;  // The page number in real memory (relative to the
            //  start of "mainMemory"
    bool valid;          // If this bit is set, the translation is ignored.
            // (In other words, the entry hasn't been initialized.)
    bool readOnly;  // If this bit is set, the user program is not allowed
            // to modify the contents of the page.
    bool use;            // This bit is set by the hardware every time the
            // page is referenced or modified.
    bool dirty;          // This bit is set by the hardware every time the
            // page is modified.
};
```

此處的class 主要是定義一個translation table ，記錄每個 virtual page 對應哪個 physical page ，另外提供了一些 access 的控制管理。

addrspace.h :

```
class AddrSpace {
  public:
    AddrSpace();            // Create an address space.
    ~AddrSpace();           // De-allocate an address space

    void Execute(char *fileName);   // Run the the program
                    // stored in the file "executable"

    void SaveState();           // Save/restore address space-specific
    void RestoreState();        // info on a context switch

  private:
    TranslationEntry *pageTable;    // Assume linear page table translation
                    // for now!
    unsigned int numPages;      // Number of pages in the virtual
                    // address space

    bool Load(char *fileName);      // Load the program into memory
                    // return false if not found

    void InitRegisters();       // Initialize user-level CPU registers,
                    // before jumping to user code

};
```

此次的class 主要定義記憶體空間的相關資訊及函式，包括 Execute: 執行程式 SaveState 儲存特定的記憶體空間資訊以便於作context switch，並包含 TranslationEntry 的資料，記錄對應之 virtual page 數值 與 physical page 數值。

addrspace.cc

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
    pageTable[i].physicalPage = i;
//  pageTable[i].physicalPage = 0;
    pageTable[i].valid = TRUE;
//  pageTable[i].valid = FALSE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
//    bzero(kernel->machine->mainMemory, MemorySize);
}
```

創造一記憶體空間去執行使用者的program ，在此處設置 virtual page 與 physical page 的對應關係，在修改之前，此處預設為一對一的關係，不去處理 multiprogramming 的情況，pageTable則是紀錄該頁的性質以及virtual page與 physical page 的對照。

```
bool
AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;

    if (executable == NULL) {
    cerr << "Unable to open file " << fileName << "\n";
    return FALSE;
    }
    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

// 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);

// 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);
            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);
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

    delete executable;          // close file
    return TRUE;            // success
}
```

Load function：是要將code載入至記憶體中，在這有幾個較重要的部分。第一為計算所需的記憶體page數，透過將 code 大小 + initial data 大小 + uninitial data 大小 + stack size先算出總共的數值，再將其除以page size ，算出此次program所需的 page number，故未修改之前，這裡的處理單純是從 main memory 去找程序的切入點，找到要從哪裡開始讀取code context 以及讀取 initial data 的地方與範圍。這邊為下一部份的主要修改的區域。

```
AddrSpace::~AddrSpace()
{
   delete pageTable;
}
```

使用結束後，需釋出該記憶體區段。

3. How you modified Nachos to make it support multiprogramming – important code segments.

首先，會要解決context switch error的問題，必須記錄physical address (在main memory內的實際位置)。每一個process 都會有該執行的記憶體空間，去儲存code、initial data、stack 等初始之記憶體位置，但在此必須更精確的紀錄，這些記憶體位置在physical address 的哪邊，在程序執行時，就會去找pageTable中所對應的physical page的頁數，接著去執行（讀code與接收初始data），在尚未修改的時候，所有process的physical皆是共用的，所以才會在multiprogramming 時出現問題（讀到同一份code）。

```
class AddrSpace {
  public:
    AddrSpace();                // Create an address space.
    ~AddrSpace();               // De-allocate an address space

    void Execute(char *fileName);   // Run the the program
                    // stored in the file "executable"

    void SaveState();           // Save/restore address space-specific
    void RestoreState();        // info on a context switch
    static bool occupied[NumPhysPages];

  private:
    TranslationEntry *pageTable;    // Assume linear page table translation
                    // for now!
    unsigned int numPages;      // Number of pages in the virtual
                    // address space

    bool Load(char *fileName);      // Load the program into memory
                    // return false if not found

    void InitRegisters();       // Initialize user-level CPU registers,
                    // before jumping to user code

};
```

上圖紅底線部份為在 addrspace.h 中，在AddrSpace的class 中加入一布林矩陣紀錄哪些是已經被佔用的page。

```
#include "copyright.h"
#include "main.h"
#include "addrspace.h"
#include "machine.h"
#include "noff.h"

bool AddrSpace::occupied[NumPhysPages] = {0};
```

再來，於addrspace.cc的剛開始先將用於記錄哪些physical page已被佔用的矩陣初始化。

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

接著主要在 addrspace.cc的load function內修改，在計算完code、initial data、stack等等所使用的總頁數後，利用迴圈去尋找occupied的記錄矩陣中哪些physical page尚未使用，再來將pageTable中virtual page 與 physical page的對應關係記錄於其中，並將一些性質一併記錄在pageTable中。

```
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
    // modified
        executable->ReadAt(
        &(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize +
            (noffH.code.virtualAddr%PageSize)]),
        noffH.code.size, noffH.code.inFileAddr);
// end modified
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
    // modified
        executable->ReadAt(
        &(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/PageSize].physicalPage * PageSize +
            (noffH.code.virtualAddr%PageSize)]),
            noffH.initData.size, noffH.initData.inFileAddr);
// end modified
```

在載入的動作時，需要讀取main memory的位置，也就是ReadAt function 的工作，在一段是讀入code context 的位置，利用pageTable找出對應的physical page，首先算出第幾頁，先得到起始位置 (pageTable[noffH.code.virtualAddr/PageSize].physicalPage*PageSize)

再來計算偏移量（(noffH.code.virtualAddr%PageSize)），將兩者相加就是程序的進入點。讀入initial data 的概念也與上述相同。

```
AddrSpace::~AddrSpace()
{

    for(int i = 0; i < numPages; i++)
        AddrSpace::occupied[pageTable[i].physicalPage] = false;
   delete pageTable;
}
```

最後，程序結束就將已佔用physical page的紀錄刪除。

4. Screenshot the final result.

```
morris@ubuntu:~/Downloads/nachos-4.0/code/userprog$ ./nachos -e ../test/test1 -e
 ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../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
```

經過修改後的結果已經變回正常，每個程序的進入點正確，印出符合期待的結果。