Motivation

Problem analysis

當分別執行 test1 和 test2 時，結果如下:

```
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
```

Fig.1 test1 results

```
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
```

Fig.2 test2 results

test1 的輸出是從 9 到 6，test2 的輸出是從 20 到 25。

若是同時輸出時，結果如下:

```
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
```

可以發現輸出的範圍有所改變，順序也錯亂了，時而遞增時而遞減。

研判我們預期的輸出可能受到作業系統 context switch 的影響，當執行緒在 test1 和 test2 中切換時，程式碼區段記憶體互相洗掉，因為作業系統沒有特別處理多執行緒的問題。

Plan

當系統執行 context switch 時，作業系統必須儲存當下 process 執行所使用的記憶體位置、所有的資料和堆疊等等，也包括所有 virtual memory 對應的 physical memory(使用 page table)。因此如何正確操作 physical memory 是解決多執行緒的關鍵。
解決辦法便是去仔細記錄每一個 process 執行時的 physical memory，在程式執行時，仔細地找到對應的記憶體位置載入。

Implementation

```
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 usedPhyPages[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

};
```

在 AddrSpace 的 class 中加入一個 usedPhyPages 去紀錄使用過的記憶體。

```
56
57  AddrSpace::AddrSpace(){}
58
```

原本的 AddrSpace 中的 default constructor 裡做簡單的 page assign，當超過兩個以上的 process 在使用 virtual page 去執行程式碼時，就會讀取到相同的 physical memory。我們修改它的架構讓 AddrSpace 的 constructor 不做任何事。

```
104     //   added  ************************************************************************
105     pageTable = new TranslationEntry[numPages];
106     for(int j = 0, k = 0; j < numPages; j++) {
107         pageTable[j].virtualPage = j;
108         while(AddrSpace::usedPhyPages[k] == true && k < NumPhysPages) k++;
109         pageTable[j].valid = true;
110         pageTable[j].use = false;
111         pageTable[j].dirty = false;
112         pageTable[j].readOnly = false;
113         pageTable[j].physicalPage = k;
114         AddrSpace::usedPhyPages[k] = true;
115     } // added  ************************************************************************
```

把所有已經用過的 physical page 初始化

```
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);
// ********************************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
 }
```

把本來的 virtual memory 的對應改成對應到 physical memory 的實際位置。

```
64   AddrSpace::~AddrSpace(){
65       for(int i = 0; i < numPages; i++)
66           AddrSpace::usedPhyPages[pageTable[i].physicalPage] = false;
67       delete pageTable;
68   }
```

最後在 deconstructor 釋放動態記憶體。

Result

```
zhewei@ubuntu:~/Downloads/Project1/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
```

更改完後，即使遇到 multithread 的 context switch 也可以有正常的輸出