電機所碩二

張光耀

R04921005

Operating System

Project 3 - Memory Management

1. Problem analysis

```
morris@ubuntu:~/Downloads/nachos-4.0_original/code/userprog$ ./nachos -e ../test
/matmult
Total threads number is 1
Thread ../test/matmult is executing.
Assertion failed: line 118 file ../userprog/addrspace.cc
Aborted (core dumped)
```

```
morris@ubuntu:~/Downloads/nachos-4.0_original/code/userprog$ ./nachos -e ../test
/sort
Total threads number is 1
Thread ../test/sort is executing.
Assertion failed: line 118 file ../userprog/addrspace.cc
Aborted (core dumped)
```

```
morris@ubuntu:~/Downloads/nachos-4.0_original/code/userprog$ ./nachos -e ../test
/sort -e ../test/matmult
Total threads number is 2
Thread ../test/sort is executing.
Thread ../test/matmult is executing.
Assertion failed: line 118 file ../userprog/addrspace.cc
Aborted (core dumped)
```

上圖為執行 sort, matmult, sort & matmult 的原始執行狀況，從上面三張圖可以發現，單獨跑 test/sort or test/matmult 這兩支程式 (sort：單一矩陣大小排序、matmult：兩矩陣乘積)，使用原本 nachos 現有設定的 main memory 大小都會發生記憶體不足的情形，一起執行更是一樣的結果。直覺的解決方式是可以夠過 memory size 來嘗試跑這兩支程式，但其實在實際的應用來說，記憶體的容量是physical 上的限制，沒有辦法去調整硬體的極限，此時，作業系統提供了一個嶄新的服務：虛擬記憶體 (virtual memory) 管理，夠過作業系統的調配，使得 main memory內的page 在記憶體不足的情況可以貯存到 hard disk 等大型儲存裝置，透過 page replacement 演算法挑選 swap out 的 page ，將現在將要使用的 page swap in main memory，讓程式端有個擴充記憶體的假象，藉以紓緩此記憶體不足的現象。本次 project 就是要完成 nachos上 virtual memory 管理的部分，讓上述兩程式不管單獨或同步執行都能通過。

2.How you implement to solve the problem in Nachos

userkernel.h 〉 No Selection

```
#include "kernel.h"
#include "filesys.h"
#include "machine.h"
#include "synchdisk.h"
class SynchDisk;
class UserProgKernel : public ThreadedKernel {
  public:
    UserProgKernel(int argc, char **argv);
                // Interpret command line arguments
    ~UserProgKernel();      // deallocate the kernel

    void Initialize();      // initialize the kernel

    void Run();         // do kernel stuff

    void SelfTest();        // test whether kernel is working

    SynchDisk *Swap_Area;     // create swap area for virtual memory
// These are public for notational convenience.
    Machine *machine;
    FileSystem *fileSystem;
    bool debugUserProg;
#ifdef FILESYS
    SynchDisk *synchDisk;
#endif // FILESYS

  private:
        Thread* t[10];  // single step user program

    char*   execfile[10];
    int execfileNum;
};

#endif //USERKERNEL_H
```

上圖為 **userkernel.h** ，首先在 kernel 中增加一個 SynchDisk 物件，名稱為 Swap_Area（紅框部分），這是要創造一個硬碟區域用來貯存那些沒辦法進入 main memory 的 page，先讓那些 page 存進此 swap area。

userkernel.cc 〉 No Selection

```
void
UserProgKernel::Initialize()
{
    ThreadedKernel::Initialize();   // init multithreading

    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();
    Swap_Area = new SynchDisk("New Disk for Swap Area");//Create swap area for virtual memory
#ifdef FILESYS
    synchDisk = new SynchDisk("New SynchDisk");
#endif // FILESYS

}
```

上圖為 **userkernel.cc** ，因為上一頁描述了增加 swap area ，所以在 class 的初始化動態分配了此 Swap area 的記憶體空間。

machine.h 〉 class Machine

```
    TranslationEntry *pageTable;
    unsigned int pageTableSize;
    bool ReadMem(int addr, int size, int* value);
    int  Identity;
    int SectorNum;//record sector number
    int FrameName[NumPhysPages];
    bool Occupied_frame[NumPhysPages];//record which frame in the main memory is occupied.
    bool Occupied_virpage[NumPhysPages];

    // start for page replacement //
    int LRU_times[NumPhysPages]; //for LRU
    // end //

    TranslationEntry *main_tab[NumPhysPages];
```

上圖為 **machine.cc** ，此處為 class machine 的部分，在這邊需要新增幾個 member ，如我們需要記錄哪些 main memory frame 已經被使用，並記錄一些詳細資訊如 ID、Sector number、frame name，除了 main memory 的資訊外，還需要紀錄 virtual memory 地區的使用情形，之後再做 page replacement 才有尋找的依據。

addrspace.cc 〉 No Selection

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

    unsigned int size,tmp;

    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;


   pageTable = new TranslationEntry[numPages];

    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);
```

上圖為 **addrspace.cc** ，我們可以在 load 函數內發現，在原本的版本（紅框部分），若 page 的總數若大於實體 main memory 的 page 數，Nachos作業系統就會將其停止，所以再更改的版本需要將其 comment 掉。

addrspace.cc 〉 AddrSpace::Load(char *fileName)

```
if (noffH.code.size > 0) {
//        DEBUG(dbgAddr, "Initializing code segment.");
//  DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);

            for(int j=0,i=0;i < numPages ;i++){
                j=0;
                while(kernel->machine->Occupied_frame[j] != FALSE && j < NumPhysPages)
                    j += 1;

                //if memory is enough,just put data in without using virtual memory
                if(j<NumPhysPages){
                    pageTable[i].physicalPage = j;
                    pageTable[i].use = FALSE;
                    pageTable[i].dirty = FALSE;
                    pageTable[i].ID =ID;
                    pageTable[i].readOnly = FALSE;
                    pageTable[i].valid = TRUE;
                    kernel->machine->Occupied_frame[j]=TRUE;
                    kernel->machine->FrameName[j]=ID;
                    kernel->machine->main_tab[j]=&pageTable[i];
                    pageTable[i].LRU_times++;
                    executable->ReadAt(&(kernel->machine->mainMemory[j*PageSize]),PageSize, noffH.code.
                        inFileAddr+(i*PageSize));
                }
              //Use virtual memory technique
              else{
                   char *buffer;
                   buffer = new char[PageSize];
                   tmp=0;
                   while(kernel->machine->Occupied_virpage[tmp]!=FALSE){tmp++;}
                   pageTable[i].virtualPage=tmp;
                   pageTable[i].ID =ID;
                   pageTable[i].valid = FALSE;
                   pageTable[i].dirty = FALSE;
                   pageTable[i].readOnly = FALSE;
                   pageTable[i].use = FALSE;
                   kernel->machine->Occupied_virpage[tmp]=true;
                   executable->ReadAt(buffer,PageSize, noffH.code.inFileAddr+(i*PageSize));
                   kernel->Swap_Area->WriteSector(tmp, buffer); //call virtual_disk write in virtual memory

                  }
              }
    }
```

上圖為 **addrspace.cc** ，同樣是在 load 函數內，首先需要將欲執行的程式分配到 main memory 內並記錄與對應 page table，此處優先會去填 main memory，當發現 main memory 的 frame 數不夠時，就會將其餘的 page 貯存到 virtual memory 內並同樣對應 page table，但此處需注意，這些被存到 virtual memory 的 page 在 page table 上需要被標記成 invalid ，當程式需要這個 page 時才知道要發出page fault 的 trap ，讓作業系統知道他要去 virtual memory 找尋 page。這裡有個很重要的函數為 ReadAt，用法為 ReadAt(檔案貯存的位置, 檔案大小, 開始讀取的offset)。另外還有 WriteSector，其作用是把 page 寫入 virtual memory。

addrspace.cc 〉No Selection

```
void
AddrSpace::Execute(char *fileName)
{
    Is_ptable_loaded=FALSE;
    if (!Load(fileName)) {
    cout << "inside Load(FileName)" << endl;
    return;             // executable not found
    }

    //kernel->currentThread->space = this;
    this->InitRegisters();      // set the initial register values
    this->RestoreState();       // load page table register
    Is_ptable_loaded=TRUE;
    kernel->machine->Run();     // jump to the user progam

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

```
//----------------------------------------------------------------------
// AddrSpace::SaveState
//  On a context switch, save any machine state, specific
//  to this address space, that needs saving.
//
//  For now, don't need to save anything!
//----------------------------------------------------------------------

void AddrSpace::SaveState()
{
    if(Is_ptable_loaded){
        pageTable=kernel->machine->pageTable;
        numPages=kernel->machine->pageTableSize;
    }
}
```

上圖為 **addrspace.cc** ，在 Execute 函數當中，先確認Loading 檔案有沒有發生錯誤，利用Is_ptable_loaded去做判斷。接著更改SaveState函數，使得之後在context switch 時再去觸發 if 內的工作，將Process State 給貯存起來。

到上述為止，已經能夠將那些無法被放入 main memory frame 內的 pages 放入 virtual memory內，之後就是下一步的 page replacement algorithm，將在下一小節做完整的介紹。

3. What scheduling methods you based

在 Page replacement algorithm ，首先是實作 Least Recently Used (LRU)，另外也實作了 Random choose （在Extra effort 中呈現 ）

Least Recently Used (LRU)：

translate.h 〉 No Selection

```
// of liability and disclaimer of warranty provisions.

#ifndef TLB_H
#define TLB_H

#include "copyright.h"
#include "utility.h"

// The following class defines an entry in a translation table -- either
// in a page table or a TLB.  Each entry defines a mapping from one
// virtual page to one physical page.
// In addition, there are some extra bits for access control (valid and
// read-only) and some bits for usage information (use and dirty).

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.
    int ID;

    int LRU_times;      //for Least Recently used algorithm

};
```

上圖為 **translate.h** ，首先在 LRU的實作，我是利用一個counter （LRU_times）去記錄哪個在main memory的page被使用到最少次，所以在 TranslationEntry class中 需要增加一個counter去記錄 page的使用次數。

translate.cc 〉 Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)

```
    } else if (!pageTable[vpn].valid) {

            printf("Page fault Happen!\n");
            kernel->stats->numPageFaults += 1;
            j=0;
            while(kernel->machine->Occupied_frame[j]!=FALSE&&j<NumPhysPages)
                j += 1;

                if( j < NumPhysPages){
                     char *buffer; //save page temporary
                     buffer = new char[PageSize];
                     pageTable[vpn].physicalPage = j;
                     pageTable[vpn].valid = TRUE;
                     kernel->machine->Occupied_frame[j]=TRUE;
                     kernel->machine->FrameName[j]=pageTable[vpn].ID;
                     kernel->machine->main_tab[j]=&pageTable[vpn];

                    // pageTable[vpn].LRU_times++; //for LRU

                     kernel->Swap_Area->ReadSector(pageTable[vpn].virtualPage, buffer);
                     bcopy(buffer,&mainMemory[j*PageSize],PageSize);

                }
               else{
                         char *buffer1;
                         char *buffer2;
                         buffer1 = new char[PageSize];
                         buffer2 = new char[PageSize];

                    //Random
                    //Swap_out_page = (rand()%32);


                    //LRU

                    int min = pageTable[0].LRU_times;
                    Swap_out_page=0;
                    for(int cc=0;cc<32;cc++){
                            if(min > pageTable[cc].LRU_times){
                                   min = pageTable[cc].LRU_times;
                                   Swap_out_page = cc;

                            }
                    }
                    pageTable[Swap_out_page].LRU_times++;



                    printf("Page%d swap out!\n",Swap_out_page);
                    bcopy(&mainMemory[Swap_out_page*PageSize],buffer1,PageSize);
                    kernel->Swap_Area->ReadSector(pageTable[vpn].virtualPage, buffer2);
                    bcopy(buffer2,&mainMemory[Swap_out_page*PageSize],PageSize);
                    kernel->Swap_Area->WriteSector(pageTable[vpn].virtualPage,buffer1);

                    main_tab[Swap_out_page]->virtualPage=pageTable[vpn].virtualPage;
                    main_tab[Swap_out_page]->valid=FALSE;

                    pageTable[vpn].valid = TRUE;
                    pageTable[vpn].physicalPage = Swap_out_page;
                    kernel->machine->FrameName[Swap_out_page]=pageTable[vpn].ID;
                    main_tab[Swap_out_page]=&pageTable[vpn];
                    printf("Finish the page replcement!\n");



                     }
    }
```

上圖為 **translate.cc**，為主要實作 LRU 的部分，首先透過判斷page table 上的 valid bit，就知道該 page 在 main memory 裡或 virtual memory 內，若發現其 page 在 virtual memory 內，首先去看看 main memory是不有空的 frame 能夠填進去，若有就將其填入

並紀錄何處已經被佔用，若 main memory已經沒有空位，就要進行 page replacement，首先開兩個 buffer (buffer1 and buffer2) 是為了等等要 swap in / out 的暫存空間，再來以LRU而言，就是要挑選哪個 main memory 的 frame要被 swap out，挑選的規則就是換掉最少使用的那個frame，利用一個 for loop 就能輕易找出最少使用次數的 page，最後利用 bcopy、ReadSector、WriteSector，分別讀取 virtual memory 複製到 main memory，以及將被挑選到要swap out 的 page 複製到 virtual memory，完成這次的page replacement，值得注意的是，在做page replacement的當下也必須好好地maintain page table 的資訊，才不會出現錯誤，並也記錄 page fault 次數。大致上就完成了 LRU algorithm，實驗結果會呈現在 下一小節。

4. Experiment result and discussion

Least Recently Used (LRU)：

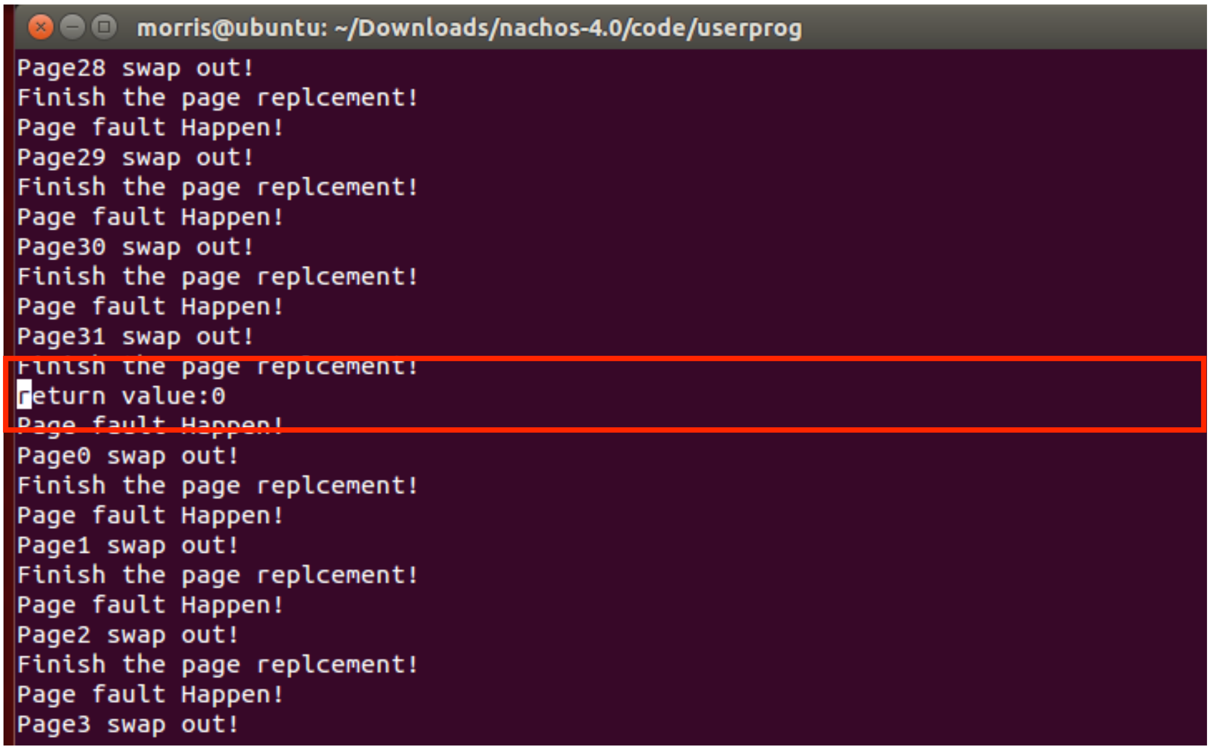
此處結果的呈現，直接同時執行 test/matmult 與 test/sort，將結果重導到output檔案。

```
morris@ubuntu: ~/Downloads/nachos-4.0/code/userprog
morris@ubuntu:~/Downloads/nachos-4.0/code/userprog$ ./nachos -e ../test/sort -e
../test/matmult > output
```

利用 virtual memory 的技術已經可以讓兩支程式同時進行，如下：

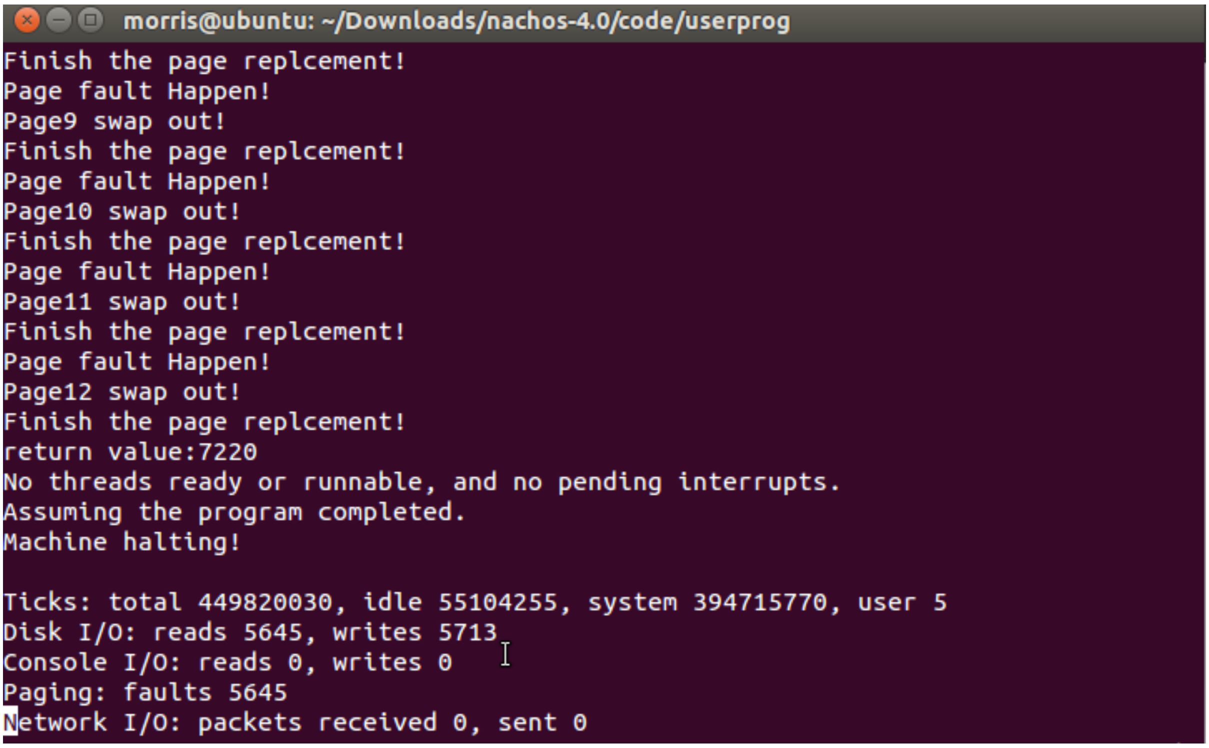
```
morris@ubuntu: ~/Downloads/nachos-4.0/code/userprog
Total threads number is 2
Thread ../test/sort is executing.
Thread ../test/matmult is executing.
Page fault Happen!
Page0 swap out!
Finish the page replcement!
Page fault Happen!
Page1 swap out!
Finish the page replcement!
Page fault Happen!
Page2 swap out!
Finish the page replcement!
Page fault Happen!
Page3 swap out!
Finish the page replcement!
Page fault Happen!
Page4 swap out!
Finish the page replcement!
Page fault Happen!
Page5 swap out!
Finish the page replcement!
Page fault Happen!
Page6 swap out!
```

可以看到不斷的有 page fault 發生，並進行 swap in/out 來處理 page fault。因為有太多 page fault 所以中間的過程就不一一列出，下圖只顯示：sorting 結束：



```
morris@ubuntu: ~/Downloads/nachos-4.0/code/userprog
Page28 swap out!
Finish the page replcement!
Page fault Happen!
Page29 swap out!
Finish the page replcement!
Page fault Happen!
Page30 swap out!
Finish the page replcement!
Page fault Happen!
Page31 swap out!
Finish the page replcement!
return value:0
Page fault Happen!
Page0 swap out!
Finish the page replcement!
Page fault Happen!
Page1 swap out!
Finish the page replcement!
Page fault Happen!
Page2 swap out!
Finish the page replcement!
Page fault Happen!
Page3 swap out!
```

結果為正確！**return value 為 0** (原本是矩陣是 1023~0 —> sorting完為: 0~1023) return value為矩陣的第一個元素。下圖為 matmult 的結果：



```
morris@ubuntu: ~/Downloads/nachos-4.0/code/userprog
Finish the page replcement!
Page fault Happen!
Page9 swap out!
Finish the page replcement!
Page fault Happen!
Page10 swap out!
Finish the page replcement!
Page fault Happen!
Page11 swap out!
Finish the page replcement!
Page fault Happen!
Page12 swap out!
Finish the page replcement!
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 449820030, idle 55104255, system 394715770, user 5
Disk I/O: reads 5645, writes 5713
Console I/O: reads 0, writes 0
Paging: faults 5645
Network I/O: packets received 0, sent 0
```

結果為正確！**return value 為 7220**。

可以發現 Paging fault 的次數也顯示在下方，能降低Paging fault 的方式可以夠過調大 page size。透過 virtual memory的技術，已經能夠有效解決 physically上的限制，也是作業系統利用程式化的管理提供硬體上無法提供的功能。RLU algorithm 為接近optimal algorithm，能夠有效降低page fault rate ，但仍產生了一些額外的問題，可以利用一些更進階的 algorithm 如同 second chance 等來加強。

5. Extra effort or observation

因為我實作 RLU 是利用counter 來記錄次數，其實對於作業系統也是個負擔，所以我在這裡另外實作一個簡單的 page replacement algorithm： random choose，隨機選一個 frame 做 swap out，可以降低OS的負擔。

```
    } else if (!pageTable[vpn].valid) {

            printf("Page fault Happen!\n");
            kernel->stats->numPageFaults += 1;
            j=0;
            while(kernel->machine->Occupied_frame[j]!=FALSE&&j<NumPhysPages)
                j += 1;

                if( j < NumPhysPages){
                    char *buffer; //save page temporary
                    buffer = new char[PageSize];
                    pageTable[vpn].physicalPage = j;
                    pageTable[vpn].valid = TRUE;
                    kernel->machine->Occupied_frame[j]=TRUE;
                    kernel->machine->FrameName[j]=pageTable[vpn].ID;
                    kernel->machine->main_tab[j]=&pageTable[vpn];

                // pageTable[vpn].LRU_times++; //for LRU

                    kernel->Swap_Area->ReadSector(pageTable[vpn].virtualPage, buffer);
                    bcopy(buffer,&mainMemory[j*PageSize],PageSize);

                }
                else{
                        char *buffer1;
                        char *buffer2;
                        buffer1 = new char[PageSize];
                        buffer2 = new char[PageSize];

                    //Random
                    Swap_out_page = (rand()%32);
```

紅框為random choose 挑選 swap out frame的部分，可以發現不用記錄任何的counter，也可以達到不錯的效果。