OS Nachos Report 3

1. Motivation

- Motivation & Problem Analysis：

  參考文獻並且實際Trace Code後，整理出流程如下：

  1. main呼叫成立userkernel，並且由繼承關係觸發kernel(threaded kernel)的建構函式(與作業1、2相同)
  2. 對於userkernel底下的每個Thread，建立其AddressSpace，其中包含pageTable(存有virtual address與physical address之轉換關係，如HW1)，並且把Code與Data Load()進來，最後觸發Execute()
  3. 藉由Kernel底下的machine->Run()，不斷觸發OneInstruction()，又觸發ReadMem()或WriteMem()的函數，藉以讀取與寫入指令。
  4. ReadMem()或WriteMem()都會經由translate()這個函數，使用pageTable把virtual address轉換為physical address。但若是要求pageTable中的頁數並不在mainMemory中(即為pageTable[vpn].valid==false)，這時候會觸發PageFaultException，標記為失敗並重新再執行一次。

- My Plan to deal with the problem

  一開始想的時候，感覺被不同的TranslationEntry與FrameInfoEntry弄得頭暈目眩，所以一開始先釐清各自的作用：

  1. PageTable: 對於每一個程式，皆有一個自己的pageTable，使其得以進行virtual page與physical frame的轉換，並且在machine執行該程式時，把machine中pageTable的指標指向當下程式的pageTable。由於程式所需的記憶體大小為[code占用的空間]+[initData占用的空間]+[uninitData占用的空間]+[stack(定義為1024)]，所以需要該空間除以PageSize(128bytes)的numPages數量。其中每一個元素：

     - valid：該page是否可以被翻譯(若true，代表該page在physical frame中；若false，代表該page在disk裡面需要被拿出來)
     - virtualPage：該page在Virtual Memory(Disk中)的編號
     - physicalPage：該page在mainMemory中frame的編號

     以matmult.c為例，所需空間為1040+0+4800+1024=6864，所需6864/128~=54個pages。所以pageTable含有54的元素，裡面代表每一頁存在physical frame或Disk當中。

  2. FrameTable: 僅有單一一個，有NumPhysPages(32)頁，每一頁都是mainMemory中128bytes的區段。經由助教提示，每一頁都是一個FrameInfoEntry類：

     - valid: 本來助教的提示為if being used，後來我覺得whether is valid for use思考起來比較順，便如此使用(僅為一個NOT的區別)
     - addrspace: 占有該physical frame程式的address space
     - vpn:該程式的特定某一頁。

  3. SwapTable: 僅有單一一個，有SectorsPerTrack(32)*NumTracks(32)=NumSectors(1024)頁，每一頁都是Disk裡面中128bytes的區段，與Frame大小相同。 經由助教提示，每一頁都是一個

FrameInfoEntry類：

- valid: whether is valid for use
- addrspace: 占有該Virtual Memory程式的address space
- vpn:該程式的特定某一頁。

藉由助教的提示，在整個運行過程中要維護好這三個Table，讓裡面存有的資料與指向的程式保持正確。其中需要更動的地方包含：

1. Addrspace::Load(): 在載入程式時，matmult.c與sort.c兩者皆大於32頁的physical frame，所以需要把部分資料存到Disk當中。
2. Machine::Translate(): 在運行指令時，若該程式的某一頁不在記憶體當中，需要找一個frame空出記憶體空間，並把需要的頁從disk拿出來放進去。

2. Implementation

- 設立FrameInfoEntry Class:此為frameTable與swapTable的構成元素，其中latestTick與usageCount為後續實作LRU與LFU會用到的變數。

```
(addrspace.h)
// N3
class FrameInfoEntry{
  public:
    bool valid; //valid to use or not
    bool lock;
    AddrSpace* addrspace;  // which process is using this page
    unsigned int vpn; // virtual page number
    //which virtual page of the process is stored in this page

    unsigned int latestTick;
    unsigned int usageCount;
};
```

- 根據助教提示，建立MemoryManager類別，並且建立變數與方法：

```
(addrspace.h)
enum VictimType {
    Random,
    LRU,
    LFU
};
class MemoryManager{
  public:
    VictimType vicType;
    MemoryManager(VictimType v);
    int TransAddr(AddrSpace* space, int virAddr);
    bool AcquirePage(AddrSpace* space, int vpn);
    bool ReleasePage(AddrSpace* space, int vpn);
    void PageFaultHandler(int faultPageNum);
```

```
    int ChooseVictim();
};
```

- 在userkernel底下新增變數：
  - 建立frameTable與swapTable：為了方便全域呼叫故放於此，以後需要時便可以直接呼叫kernel->frameTable或kernel->swapTable
  - 建立磁碟空間swap，使用kernel->swap->ReadSector()或kernel->swap->WriteSector()進行特定Sector的讀寫。
  - 建立memeoryManager：也是為了方便全域呼叫，需要時可直接以kernel->memoryManager呼叫。

```
(userkernel.h)
class UserProgKernel : public ThreadedKernel {
  public:
    UserProgKernel(int argc, char **argv);
                // Interpret command line arguments
    ~UserProgKernel();      // deallocate the kernel
    ......

    //N3
    SynchDisk * swap;
    FrameInfoEntry* frameTable;
    FrameInfoEntry* swapTable;
    MemoryManager* memoryManager;
    ...
};
```

1. 改寫Addrspace::Load()的方法：以前在project 1的中，對於程式所需要的Page，用pageTable連結maianMemory的位置。然而目前因程式需要空間過大，需要額外運用到虛擬記憶體，故把磁碟空間當作mainMemory的衍伸，作法如下：
   - 對於每一頁程式所需的page(i)
     - 若仍有空的physical frame(j)，分配之，直到所有page分配完或是32頁physicacl frame分完，並且更新pageTable與frameTable
     - 若目前physical frame全數已在使用，則將各頁放入還未使用的disk(j)當中，並且更新pageTable與swapTable
   - 根據page對應到physical frame或virtual memory：
     - 若對應到physical frame：把該頁用以前的方法 (executable->ReadAt(&(kernel->machine->mainMemory[physical address]), sizeToLoadNow, inFileAddr),PageSize,inFileAddr);)讀取進來(與project1用法類似，但僅讀一頁)
     - 若對應到virtual Memory：把資料讀到暫時的空間後，寫入disk對應的Sector裡面(使用disk的WriteSector()方法)

```
bool
AddrSpace::Load(char *fileName)
{
    ...
```

```
    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);
    size = numPages * PageSize;

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

    // Initialize PageTable
    pageTable = new TranslationEntry[numPages];

    //N3: Indexing: always i for pageTable, j for frameTable; k for swapTable
    int i,j,k;
    for(i=0, j=0; i<numPages && j<NumPhysPages; i++, j++){
        //use physical frame
        while(kernel->frameTable[j].valid==false)j++;
        pageTable[i].virtualPage=1024; //VM no need
        pageTable[i].physicalPage=j; //in physical memory
        pageTable[i].valid=true; // can be used
        pageTable[i].use=false;
        pageTable[i].dirty=false;
        pageTable[i].readOnly=false;
        pageTable[i].latestTick=0;
        //update frameTable
        kernel->frameTable[j].valid=false; //occupied
        kernel->frameTable[j].addrspace = this;
        kernel->frameTable[j].vpn=i;
    }
    for(k=0; i<numPages && k<1024; i++, k++){
        //use VM
        while(kernel->swapTable[k].valid==false)k++;
        pageTable[i].virtualPage=k; //in VM
        pageTable[i].physicalPage=NumPhysPages; // not in physical frame
        pageTable[i].valid=false; // can't be used
        pageTable[i].use=false;
        pageTable[i].dirty=false;
        pageTable[i].readOnly=false;
        pageTable[i].latestTick=0;

        //update swapTable
        kernel->swapTable[k].valid=false;
        kernel->swapTable[k].addrspace = this;
        kernel->swapTable[k].vpn=i;
    }
    ...
}
```

2. 實作PageFaultHandler：從以上的problem analysis中，可以發現PageFault出現並需要解決的地方，在於translate.cc中：當要求程式的頁碼vpn，查pageTable時發現不在實體記憶體中(pageTable[vpn].valid==false)即會觸發。於是在這邊呼叫kernel->memoryManager->PageFaultHandler()，並且把缺失的vpn號碼做為參數傳入：

```
(translate.cc)
ExceptionType
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
{
    ......
// calculate the virtual page number, and offset within the page,
// from the virtual address
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    //N3: for LRU to choose victim
    pageTable[vpn].latestTick=kernel->stats->totalTicks;

    if (tlb == NULL) {      // => page table => vpn is index into table
    if (vpn >= pageTableSize) {
        DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
        return AddressErrorException;
    } else if (!pageTable[vpn].valid) {`
        DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
        //N3: page fault occured
        kernel->memoryManager->PageFaultHandler(vpn);
        return PageFaultException;
    }
    entry = &pageTable[vpn];
    ...
    // N3: LRU and LFU counting
    kernel->frameTable[pageFrame].usageCount++;
    kernel->frameTable[pageFrame].latestTick = kernel->stats->totalTicks;
    ...
    return NoException;
}
```

3. 接著把以上第二點呼叫的MemoryManager::PageFaultHandler()以及其他MemoryManager裡面的function進行實作：

    - 實作AcquirePage(Addrspace* space, int vpn)：經由助教的提示，此方法為ask a frame for vpn, 即是把此程式的第vpn頁，從disk的第k(=space->pageTable[vpn].virtualPage個Sector拿出，並且放入到mainMemory當中。在實體記憶體的32頁中，讓j從0到31，依序去觀察該frameTable[j].valid是否有被占用，並且試圖把該page由虛擬記憶體轉換到實體記憶體。
        - 若成功，更新該space中pageTable的資料，紀錄該頁目前放到了實體記憶體且valid。更新frameTable[j]的資料，紀錄目前存在其中的space指標與頁碼。更新swapTable[k]的資料，被拿出了所以空出該Disk的空間。最後回傳true表示取得成功
        - 若失敗，回傳false表示取得失敗。

```
(addrspace.cc)
bool MemoryManager::AcquirePage(AddrSpace* space, int vpn){
    // Ask a page(frame) for vpn
    // From VM(disk) to frame
    // Assume there already has empty frame!
    FrameInfoEntry* frameTable = kernel->frameTable;
```

```
    FrameInfoEntry* swapTable = kernel->swapTable;
    int k = space->pageTable[vpn].virtualPage; //index for swap table
    for(int j=0; j<NumPhysPages; j++){
        // Find an available physical frame
        if(frameTable[j].valid == true){
            //update frame table: occupied
            frameTable[j].valid = false;
            frameTable[j].addrspace = space;
            frameTable[j].vpn = vpn;
            // N3: LRU LFU counting
            frameTable[j].usageCount = 0;
            frameTable[j].latestTick = kernel->stats->totalTicks;

            //copy data from VM to frame
            char* inBuffer = new char[PageSize];
            kernel->swap->ReadSector(k, inBuffer);
            bcopy(inBuffer, &(kernel->machine->mainMemory[j*PageSize]),
PageSize);

            //update page table
            space->pageTable[vpn].virtualPage = 1024;
            space->pageTable[vpn].physicalPage = j;
            space->pageTable[vpn].valid=true;

            //update swap table
            swapTable[k].addrspace=NULL;
            swapTable[k].valid = true;

            delete[] inBuffer;

            DEBUG(dbgAddr, "OCCUPIED PHYSICAL FRAME "<<j);
            return true;
        }
    }
    // Exceed NumPhysPages 32, return false to indicate
    DEBUG(dbgAddr, "EXCEED NUMPHYSPAGES");
    return false;
}
```

- 實作ReleasePage(Addrspace* space, int vpn)：像是AcquirePage的反向版，目前該頁棲息於mainMemory的第j(=space->pageTable[vpn].physicalPage)頁。在一共1024個Sector的disk空間中，找出還沒被占用的Sector，試圖把該頁的資料存入(使用WriteSector())進虛擬記憶體當中。
  - 若成功，更新該space中pageTable的資料(各程式獨立)，紀錄該頁目前放到了虛擬記憶體且valid。更新swaoTable[k]的資料，紀錄目前存在其中的space指標與頁碼。更新frameTable[j]的資料，被拿出了所以空出該mainMemory的空間。最後回傳true表示取得成功
  - 若失敗，回傳false表示取得失敗。

```
(addrspace.cc)
bool MemoryManager::ReleasePage(AddrSpace* space, int vpn){
    // Free a page at a time
    // Swap out: from frame to disk
```

```
    FrameInfoEntry* frameTable = kernel->frameTable;
    FrameInfoEntry* swapTable = kernel->swapTable;
    int j = space->pageTable[vpn].physicalPage;
    for(int k=0; k<1024; k++){
        if(swapTable[k].valid==true){
            //update swap table
            swapTable[k].valid = false;
            swapTable[k].addrspace = space;
            swapTable[k].vpn = vpn;

            //update page table
            space->pageTable[vpn].valid=false;
            space->pageTable[vpn].virtualPage = k;
            space->pageTable[vpn].physicalPage = NumPhysPages;

            //copy data from frame to disk
            char* outBuffer = new char[PageSize];
            bcopy(&(kernel->machine->mainMemory[j*PageSize]), outBuffer,
PageSize);
            kernel->swap->WriteSector(k, outBuffer);

            //update frame table
            frameTable[j].valid = true;
            frameTable[j].addrspace = NULL;
            frameTable[j].latestTick = 0;

            // for LRU: reset all to zero
            for(int jj=0; jj<32; jj++){
                frameTable[jj].usageCount = 0;
            }

            delete[] outBuffer;
            DEBUG(dbgAddr, "RELEASE FRAME "<<j<<" TO VM "<<k);
            return true;
        }
    }
    // Exceed 1024 Sectors, return false to indicate
    DEBUG(dbgAddr, "EXCEED DISK SECTORS");
    return false;
}
```

- 實作ChooseVictim():為了可以使用flag指定選擇victim的方式，我採用與project 2很類似的方式，把argument parsing的方式寫在userkernel的建構函數內。之後選擇victim時，便會觸發ChooseVictim()方法，並根據輸入的flag選擇模式：
  - Random: 加入"-vic random"觸發，隨機選擇32個physical frame的其中之一，並且回傳該index值。
  - Least Recent Used(LRU): 加入"-vic lru"觸發。每次translate的時候，幫當下觸發的frame, 其latestTick紀錄當時的時間(藉由kernel->stats->totalTicks)。而在需要找victim的時候，遍歷32頁的frameTable，找出紀錄時間最早者(代表其最久沒被需要)，作為須被剃除的frame index，並且把該index回傳。

- Least Frequent Used(LFU):加入"-vic lfu"觸發。每次translate的時候，把當下觸發的frame,其usageCount增加1(代表近期內被使用的頻率+1)。需要尋找victim時，遍歷32頁frameTable，找出被使用次數最少者作為victim並回傳，同時把每一個frame的usageCount歸零(讓每一次選擇victim時，僅考慮從上一次pageFault到現今的使用狀況，否則經測試發現會變成永遠同一頁被選中)

```
(addrspace.cc)
int MemoryManager::ChooseVictim(){
    FrameInfoEntry* frameTable = kernel->frameTable;
    FrameInfoEntry* swapTable = kernel->swapTable;
    // input: method of choosing victim
    // output: index j, indicate victim for frameTable
    int ret_j=-1; // index for victim

    if(kernel->memoryManager->vicType==Random){
        //random
        ret_j = rand()%32;
        DEBUG(dbgPage, "RANDOM SWAPOUT "<<ret_j);
    }
    else if(kernel->memoryManager->vicType==LRU){
        //least recent used (LRU)
        int minTick=0;
        int min_j;
        for(int j=0;j<NumPhysPages;j++){
            DEBUG(dbgPage, "Frame "<< j <<" latestTick: "
<<frameTable[j].latestTick);
            if(j==0){
                min_j=0;
                minTick=frameTable[j].latestTick;
            }
            else{
                int curTick = frameTable[j].latestTick;
                if(curTick<minTick){
                    minTick=curTick;
                    min_j = j;
                }
            }
        }
        ret_j = min_j;
        DEBUG(dbgPage, "LRU SWAPOUT "<<ret_j);
    }
    else if(kernel->memoryManager->vicType==LFU){
        int minCount=0;
        int min_j;
        for(int j=0; j<NumPhysPages; j++){
            DEBUG(dbgPage, "Frame "<<j <<"usageCount: "
<<frameTable[j].usageCount);
            if(j==0){
                min_j=0;
                minCount=frameTable[j].usageCount;
            }
            else{
```

```
                int curCount = frameTable[j].usageCount;
                if(curCount<minCount){
                    minCount=curCount;
                    min_j = j;
                }
            }
        }
        ret_j = min_j;
        DEBUG(dbgPage, "LFU SWAPOUT "<<ret_j);
    }
    else{
        ret_j=0;
        DEBUG(dbgPage, "ELSE SWAPOUT "<<ret_j);
    }
    kernel->stats->frameStat[ret_j]++;
    return ret_j;
}
```

- 藉由以上的三函數，實作PageFaultHandler(int faultPageNum):
  - 對於想要進來的page：呼叫AcquirePage()，若成功即代表frameTable有空位可直接進入。
  - 若AcquirePage()失敗，以ChooseVictim()選一個victim，額外呼叫一次ReleasePage()，讓MemeoryManager根據指定的選擇victim方式釋放一頁physical frame，並且重新呼叫AcquirePage()一次，使該Page進入實體記憶體中。

```
(addrspace.cc)
void MemoryManager::PageFaultHandler(int faultPageNum){
DEBUG(dbgAddr, "HANDLING");
// Invoke when page fault occurs
// Exchange between frameTable <-> swapTable
TranslationEntry* pageTable = kernel->currentThread->space->pageTable;
FrameInfoEntry* frameTable = kernel->frameTable;
FrameInfoEntry* swapTable = kernel->swapTable;

int k = pageTable[faultPageNum].virtualPage; // target

while(AcquirePage(kernel->currentThread->space, faultPageNum)==false){
    //while AcquirePage return false: No available physical frame
    //release one for it
    int j_vic = ChooseVictim();
    AddrSpace* addr_vic = frameTable[j_vic].addrspace;
    int vpn_vic = frameTable[j_vic].vpn;
    bool releaseSuccess = ReleasePage(addr_vic, vpn_vic);
    ASSERT(releaseSuccess);
    }
}
```

3.Result

- 執行matmult、sort，以及兩個一起執行：

- 執行matmult:藉由三種不同選victim的方法，皆會得到return value:7220。以下以使用LRU為例：

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -e ./test/matmult -vic lru
Total threads number is 1
Thread ./test/matmult is executing.
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1365052, idle 729495, system 3430, user 632127
Disk I/O: reads 68, writes 68
Console I/O: reads 0, writes 0
Paging: faults 68
Network I/O: packets received 0, sent 0
```

- 執行sort：藉由三種不同選victim的方法，皆會得到return value:1。以下以使用Random為例：

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -e ./test/sort -vic random
Total threads number is 1
Thread ./test/sort is executing.
return value:1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 50071552, idle 11202674, system 60930, user 38807948
Disk I/O: reads 1218, writes 1218
Console I/O: reads 0, writes 0
Paging: faults 1218
Network I/O: packets received 0, sent 0
```

- 同時執行matmult與sort：藉由三種不同選victim的方法，皆會得回傳值分別為7220及1。以下以使用LFU為例：

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -e ./test/matmult -e
./test/sort -vic lfu
Total threads number is 2
Thread ./test/matmult is executing.
Thread ./test/sort is executing.
return value:7220
return value:1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 51858552, idle 12358050, system 60580, user 39439922
Disk I/O: reads 1209, writes 1212
```

```
Console I/O: reads 0, writes 0
Paging: faults 1209
Network I/O: packets received 0, sent 0
```

- 參數輸入時先sort再matmult，並且排程方式改為FIFO(-sch f)：藉由三種不同選victim的方法，皆會得回傳值分別為1及7220。以下以使用Random為例：

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -e ./test/sort -e
./test/matmult -vic random -sch f
Total threads number is 2
Thread ./test/sort is executing.
Thread ./test/matmult is executing.
return value:1
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 51638052, idle 12132462, system 65470, user 39440120
Disk I/O: reads 1308, writes 1309
Console I/O: reads 0, writes 0
Paging: faults 1308
Network I/O: packets received 0, sent 0
jch@JCH:~/nachos-4.0/code$
```

- 額外：比較不同挑選victim方式下的pageFault數目：

Vic \ Program	Matmult	Sort	Matmult Sort
Random	89	1218	1341
LRU	68	4999	5090
LFU	35	974	1209

- 討論、觀察與心得：

第三次的project，讓我對第一次project的內容又有了新的感觸。同樣是程式內virtual address所需要的轉換，加入了虛擬記憶體與disk的成分後，整體又變得更加複雜。整體而言，此次作業完整的走過了程式碼被加載，一行行於machine->Run()時，對應到所屬的地址，必要時把頁面從disk取出，最後終於得到要操作的記憶體區段。跟著code走過一輪後，實在對nachos的系統有了進一步的認識。

就實驗結果而言，在運算量大的情況下，LRU所觸發的page fault數量遠超過其他兩種，需要頻繁的I/O來從disk中取回page；而LFU則是在不同情況下，都可以取得最少的page fault數量，有利於整體的效率；而使用random的方法，在同時跑matmult與sort兩個大程式時，其造成的page fault數量接近於LFU，似乎印證了以前計算機課程中，有提及random的挑選方法有時不比其他演算法來的差。

對於LRU與LFU的差別，我試圖新增一個"-d p"的debug flag，並且統計32個frame被替換的情形。以LRU運行時，我發現各個frame被替換的次數皆相對接近(大約每個frame都是被替換180次，相對均勻分布)，在確定LRU的實作應該沒出錯後，似乎僅能假設LRU特別不適合該情況。相對於言，以LFU運行，可以發

現被替換特別頻繁的Frame僅有特定幾個，其他的frame被替換的次數只有個位數，甚至有些Frame全部的Code跑完都從沒移出過記憶體，代表常用的資料直放在主記憶體當中極少移出，有利於整體的效率。