OS Project-3 Report

R11921008 羅恩至

Memory Management

1. Motivation and Problem Analysis

此次 project 為實作 Memory management，需要正確執行 test 資料夾中的 matmult.c 和 sort.c 兩個檔案。而在未調整現有記憶體大小的情況下，若直接執行此兩檔案將會出現 core dumped 的報錯(此為實體記憶體空間不足所致)，因此需要透過建立虛擬記憶體，把程式未使用到的 page 和 data 暫時搬離至輔助記憶體中，當有需要時再把 page 和 data 搬回主記憶體，以此來節省主記憶體空間。

2. Implementation

(1) 首先在 userprog/userkernel.h 中新增一個用作虛擬記憶體空間的 SynchDisk，用來存放沒有使用的 page，並在 userprog/userkernel.cc 初始化此 SynchDisk。

```
    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

// These are public for notational convenience.
    Machine *machine;
    FileSystem *fileSystem;
    SynchDisk *Swap_Area;

#ifdef FILESYS
    SynchDisk *synchDisk;
#endif // FILESYS
```

userprog/userkernel.cc

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

    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();
    Swap_Area = new SynchDisk("New Swap Area");
#ifdef FILESYS
    synchDisk = new SynchDisk("New SynchDisk");
#endif // FILESYS
}
```

(2) 接著在 machine/machine.h 中多宣告一些變數，這些變數是用來將未使用到的 page 存放到虛擬記憶體、以及實作 LRU 演算法所需的變數。

```
  TranslationEntry *tlb;      // this pointer should be considered
                              // "read-only" to Nachos kernel code

  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 count_LRU[NumPhysPages]; //for LRU counter
  bool reference_bit[NumPhysPages];//for second chance algorithm.
  TranslationEntry *main_tab[NumPhysPages];

private:
```

(3) 在 userprog/addrspace.h 中宣告兩變數 ID 及 Is_ptable_loaded。其中 ID 用來儲存 thread 的 ID，Is_ptable_loaded 則是用來確認 page table 是否有被載入。

```
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

    int ID;

    static bool usedPhysicalPage[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
    bool pageTableLoaded;
    bool Is_ptable_loaded;

};
```

(4) 接下來需要在 userprog/addrspace.cc 中進行許多修改，操作過程如下：

在 AddrSpace::AddrSpace()中，刪除原本讀取 page table 的方式(bzero 那行)，並新增新的讀取 page table 方式。

```
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);
    ID=(kernel->machine->Identity)++;
    kernel->machine->Identity=(kernel->machine->Identity)++;
}
```

修改 AddrSpace::Load(char *fileName)中的程式碼，以更改記憶體配置的方式，主要分成兩種情況：如果實體記憶體中 frame 的數量足夠，將 page 直接載到實體記憶體；而若 frame 的數量不夠時，page 則會被載至虛擬記憶體中。

```
// 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);
        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].count_LRU++;
                executable->ReadAt(&(kernel->machine->mainMemory[j*PageSize]),PageSize, noffH.code.inFileAddr+(i*PageSize));
```

```
        }else{
            char *buffer;
            buffer = new char[PageSize];
            unsigned int 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
        }
    }
}
```

在 AddrSpace::Execute(char *fileName)中新增 Is_ptable_loaded 變數判斷 page table 是否被載入，預設為 False，而在確認有讀檔之後便會轉換成 True。

```
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"
}
```

更改 AddrSpace::SaveState()，變成在讀檔(Is_ptable_loaded 為 True)之後，才會去取得 page table 的 size。

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

(5) 在 machine/translate.h 中新增宣告兩變數 ID 和 count_LRU

```
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 count_LRU;
};
```

(6) 在 machine/translate.cc 中，需修改 Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)以實作頁面置換。

先宣告一個變數 Swap_out_page，表示 page fault 發生時用來置換的 page。

```
ExceptionType
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
{
    int i, j;
    unsigned int vpn, offset;
    TranslationEntry *entry;
    unsigned int pageFrame;
    int Swap_out_page;
    DEBUG(dbgAddr, "\tTranslate " << virtAddr << (writing ? " , write" : " , read"));
```

修改 else if(!pageTable[vpn].valid)判斷式中的內容，處理 page fault。
確認實體記憶體是否有空間，若有空間則將 page 載入至實體記憶體。

```
    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);
        //return PageFaultException;
        kernel->stats->numPageFaults += 1; // pagefault counter +1
        j=0;
        while(kernel->machine->Occupied_frame[j]!=FALSE&&j<NumPhysPages) {
            j += 1; // find valid frame space
        }
        if(j < NumPhysPages){
            char *buffer;
            buffer = new char[PageSize];
            pageTable[vpn].physicalPage = j; // save physical memory
            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].count_LRU++; // LRU

            kernel->Swap_Area->ReadSector(pageTable[vpn].virtualPage, buffer);
            bcopy(buffer,&mainMemory[j*PageSize],PageSize); // save data into physical memory
```

若實體記憶體空間不足，則使用 LRU 置換演算法，將最久沒有使用的 page swap 出去。之後再用 bcopy 將要 swap out 的 page 暫存在 buffer1，並將要 swap in 的 page 從 Swap_Area 讀進來存到 buffer2，再將 buffer1 寫到 Swap_Area 裡。

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

            //LRU
            int min = pageTable[0].count_LRU;
            Swap_out_page=0;
            for(int cc=0;cc<32;cc++){
                if(min > pageTable[cc].count_LRU){
                    min = pageTable[cc].count_LRU;
                    Swap_out_page = cc;
                }
            }
            pageTable[Swap_out_page].count_LRU++;
            // 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");
        }
    }
```

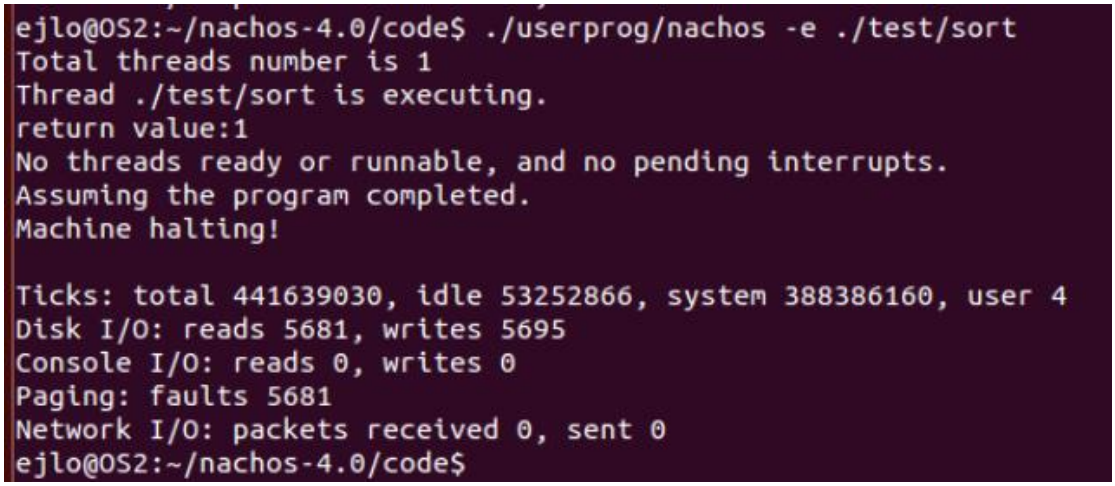
3. Result

執行 matmult.c，return value 7220，共發生 80 次 page fault

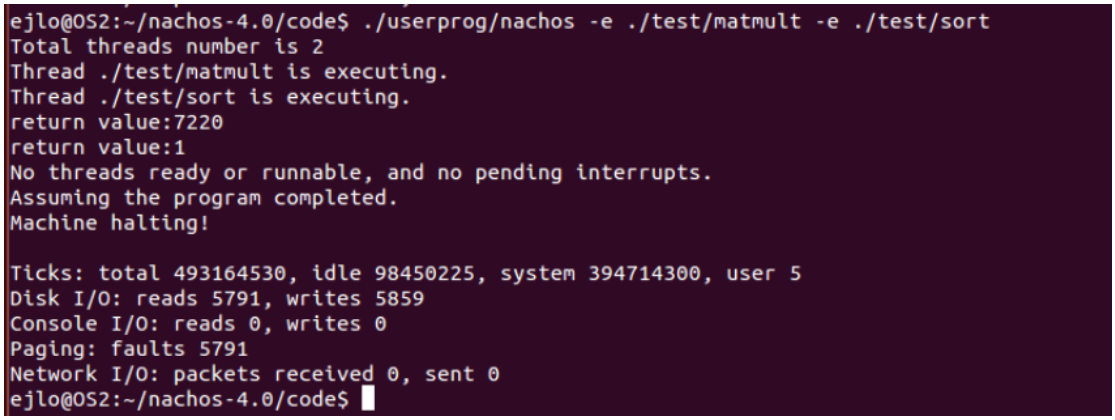
```
ejlo@OS2:~/nachos-4.0/code$ ./userprog/nachos -e ./test/matmult
Total threads number is 1
Thread ./test/matmult is executing.
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7627030, idle 1301666, system 6325360, user 4
Disk I/O: reads 80, writes 102
Console I/O: reads 0, writes 0
Paging: faults 80
Network I/O: packets received 0, sent 0
ejlo@OS2:~/nachos-4.0/code$
```

執行 sort.c，return value 1，共發生 5681 次 page fault



同時執行 matmult.c 和 sort.c，return value 7220 and 1，共發生 5791 次 page fault



討論：

(1) 根據上方的結果，可以看到 matmult.c 和 sort.c 兩個檔案均能正常執行，符合此次 Project 的需求。

(2) 在實作這次 Project 時依舊遇到了一些困難，首先是感覺要先對課程第 8 章和第 9 章的內容，熟悉 Page table、page fault、physical memory 和 virtual memory 之間的切換等知識後，做 project 才會比較上手；再來是當完成程式修改之後要進行編譯(make file)時，不知為何在 test file 那邊一直出現 error message，所幸不影響此次 project 的結果，matmult.c 和 sort.c 均能正常執行。