NachOS – MP4

Virtual Memory

宋體淮, R09921135, Electrical Engineering

I. Goal

NachOS native memory manage system did not support virtual memory technique. In this assignment you are required to **implement virtual memory management as well as page fault handling** to let NachOS support larger file size.

II. Assignment

可以觀察到當執行某些 test program 時，例如 `test/sort`、`test/matmult`，由於其需要較大量的記憶體空間因此導致 core dumped，所以需要利用 virtual memory 的技術來儲存 main memory 無法負荷的 pages。

userkernel.h：

- 加入 `SynchDisk` 這個類別來模擬 nachos disk，這是原本 NachOS 就已實做好的類別，用來與真實電腦的 disk 做溝通的一個 interface，詳見 `filesys/synchdisk`。
- `SynchDisk` 的物件具有讀出或寫入資料到真實電腦的 disk 的函式。

```
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
```

userkernel.cc：

- 在 kernel 初始化的地方建立 `SynchDisk` 物件。

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

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

machine.h：

- 接著在 machine 的地方建立一些與 physical memory 與 physical disk 有關的變數，例如需要知道有哪些 page 正被使用。
- 並且建立 LRU 會用到的 counter。
- 值得注意的是，由於每個 process 都會自己的 page table，所以當某 process 被選到為 `currentThread` 時，此處定義的 `pageTable` 會指向該 process 的 page table。

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

  // for page replacement //
  int LRU_counts[NumPhysPages]; //for LRU

  TranslationEntry *main_tab[NumPhysPages];
```

translate.h：

- `TranslationEntry` 就是 page table 的資料結構，在此多定義幾個參數來記錄每個 page 的資訊。

```
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_counts;      //for LRU

};
```

AddrSpace::Load()：

- 在一開始把 user program 載入到主記憶體時就會需要使用到 virtual memory 了。
- 首先藉由 `Occupied_frame` 來尋找 physical memory 中的 free frame。
- 若有的話利用 `ReadAt` 把該 page 載入到主記憶體中，並更新 page table 的資訊，值得注意的是在迴圈裡面就要做 `ReadAt` 來把一個 page 搬到 memory，真的有做到切分 page 這件事。

```
        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)  // check free frame
                  j += 1;

              // if physical memory is enough, just put data in without using virtual memory
              if(j<NumPhysPages){
                  pageTable[i].physicalPage = j;      // record in physical memory position j
                  pageTable[i].use = FALSE;
                  pageTable[i].dirty = FALSE;
                  pageTable[i].ID = ID;
                  pageTable[i].readOnly = FALSE;
                  pageTable[i].valid = TRUE;          // TRUE means the page exists in physical memory
                  kernel->machine->Occupied_frame[j] = TRUE;
                  kernel->machine->FrameName[j] = ID;
                  kernel->machine->main_tab[j] = &pageTable[i];        // save the page pointer
                  pageTable[i].LRU_counts++;
                  // load data to physical memory position j
                  executable->ReadAt(&(kernel->machine->mainMemory[j*PageSize]), PageSize, noffH.code.inFileAddr+
(i*PageSize));
              }
```

- 若無 free frame 的話，這時候就要把放不下的 page 先放到 disk 裡去。一樣使用 `Occupied_virpage` 來尋找 disk 中的 free sector(在這邊是設定 disk sector 數量和 memory frame 數量一樣多)。
- 找到了之後先藉由 `ReadAt` 來把該 frame 載入到 buffer，再把 buffer 的內容透過 `WriteSector` 寫入到 disk 中編號為 `tmp` 的 sector，並且把 valid bit 註記成 `FALSE`。

```
              // use virtual memory technique
              else{
                  char *buffer;
                  buffer = new char[PageSize];
                  tmp = 0;
                  while(kernel->machine->Occupied_virpage[tmp]!=FALSE){tmp++;}  // check free sector
                  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);  // write into virtual memory
              }
          }
      }
```

translate.cc：

- `Machine::Translate()` 會被 `Machine::WriteMem` 或 `Machine::ReadMem` 呼叫，這是因為 CPU 在 fetch instruction 時讀到的記

憶體位址都是每個 process 自己的 virtual address，所以需要藉由 translation 的方式轉換成 physical address。

- 因為有使用 virtual memory 的關係，所以對於每個讀到的 page，都要確認是來自於主記憶體還是 disk，因此要在此執行 page fault handling。
- 首先查看該 page 的 `valid` 參數，若為 false 代表它不在主記憶體中，還躺在 disk 上，所以此時發生了 page fault。
- 接著要看主記憶體還有沒有 free frame 來給它使用，如果有的話就把該 page 透過 `ReadSector` 讀到 buffer 中，再把 buffer 的內容搬到主記憶體，並記得要更新 page table 的資訊。

```
// calculate the virtual page number, and offset within the page,
// from the virtual address
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    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) {  // => this page hasn't loaded into physical memory
            //printf("Page fault Occurs!\n");
            kernel->stats->numPageFaults += 1;   // nachos pagefault counter +1
            j=0;
            while(kernel->machine->Occupied_frame[j]!=FALSE && j<NumPhysPages)  // check free frame
                j += 1;

            if (j < NumPhysPages){          // if find valid frame space, save the page in virtual memory into physical memory
                char *buffer; //save page temporary
                buffer = new char[PageSize];
                pageTable[vpn].physicalPage = j;       // save physical memory position
                pageTable[vpn].valid = TRUE;           // page has already in physical memory
                kernel->machine->Occupied_frame[j] = TRUE;
                kernel->machine->FrameName[j] = pageTable[vpn].ID;
                kernel->machine->main_tab[j] = &pageTable[vpn];           // save the page pointer

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

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

            }
```

- 若主記憶體沒有 free frame 就需要執行 page replacement，本次作業我使用 Least-Recently-Used 的方式，最少被 access 到的人就要被換出去(其實比較像 LFU)。
- 由於之前已經有為 page table 增加了 `LRU_counts` 這個參數，用來記錄每個 page 被 reference 的次數，要從所有 page 中找出 `LRU_counts` 最少的 page 作為 `victim`，也就是要被置換掉的 page。
- 另外在這一步被 access 到的 page 都要把 `LRU_counts` 加 1。
- 記得要更新 victim page 的資訊，以及被置換進來的 page 的資訊。

```
        else{                           // there is no free frame, page replacement needs to be invoked
            char *buffer1;
            char *buffer2;
            buffer1 = new char[PageSize];
            buffer2 = new char[PageSize];

            // Least-Recently-Used (LRU) algorithm

            int min = pageTable[0].LRU_counts;
            victim = 0;
            for (int cc = 0; cc < NumPhysPages; cc++){
                 if (min > pageTable[cc].LRU_counts){
                       min = pageTable[cc].LRU_counts;
                       victim = cc;
                 }
            }
            pageTable[victim].LRU_counts++;

            //printf("Page%d swap out!\n", victim);
            bcopy(&mainMemory[victim*PageSize],buffer1,PageSize);
            kernel->Swap_Area->ReadSector(pageTable[vpn].virtualPage, buffer2);
            bcopy(buffer2,&mainMemory[victim*PageSize],PageSize);          // load the page into physical memory
            kernel->Swap_Area->WriteSector(pageTable[vpn].virtualPage, buffer1);  // save victim back to virtual memory

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

        }
    }

    entry = &pageTable[vpn];
}
```

Result：

- 執行 test/sort，回傳值為 0 代表正確執行。

```
henry@henry-VirtualBox:~/nachos/nachos-4.0_MP4/code/userprog$ ./nachos -d f -e ../test/sort
DEBUG is enabled!
Total threads number is 1
Thread ../test/sort is executing.
(FILESYS_STUB) Opening file../test/sort
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 440615030, idle 52227336, system 388387690, user 4
Disk I/O: reads 5536, writes 5550
Console I/O: reads 0, writes 0
Paging: faults 5536
Network I/O: packets received 0, sent 0
```

- 執行 test/matmult，回傳值為 7220 代表正確執行。

```
henry@henry-VirtualBox:~/nachos/nachos-4.0_MP4/code/userprog$ ./nachos -d f -e ../test/matmult
DEBUG is enabled!
Total threads number is 1
Thread ../test/matmult is executing.
(FILESYS_STUB) Opening file../test/matmult
return value:7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7691030, idle 1365666, 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
```

- 兩者同時執行，且結果正確。


henry@henry-VirtualBox:~/nachos/nachos-4.0_MP4/code/userprog$ ./nachos -e ../test/sort -e ../test/matmult
Total threads number is 2
Thread ../test/sort is executing.
Thread ../test/matmult is executing.
return value:0
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