作業系統 - 作業三 (System Call & CPU Scheduling)

系級：電機所碩一　姓名：楊冠彥　學號：R11921091

tags: `Operating system`, `Ubuntu 14.04`

Motivation

Problem analysis

當使用NachOS現有的實體記憶體大小，某些測試case會需要大量記憶體，如此將導致core dumped。如下圖所見，三個測試case都發生core dumped的狀況，因為主記憶體中的空間可能不足，因此需要虛擬記憶體來儲存Pages。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code/userprog$ ./nachos -e ../test/matmult
Total threads number is 1
Thread ../test/matmult is executing.
Assertion failed: line 134 file ../userprog/addrspace.cc
Aborted (core dumped)
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code/userprog$ ./nachos -e ../test/sort
Total threads number is 1
Thread ../test/sort is executing.
Assertion failed: line 134 file ../userprog/addrspace.cc
Aborted (core dumped)
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code/userprog$ ./nachos -e ../test/matmult -e ../tes
t/sort
Total threads number is 2
Thread ../test/matmult is executing.
Thread ../test/sort is executing.
Assertion failed: line 134 file ../userprog/addrspace.cc
Aborted (core dumped)
```

Plan

我們的目標是實現一種虛擬記憶體，當主記憶體沒有空間，Page和data仍可以透過Page swap演算法儲存在其他儲存設備中。也就是當主記憶體中的空間足夠時，Page將存儲在主記憶體中，這樣效能會比較好。但一旦空間不足，Page和data將swap出去。當主記憶體空間已被釋放，且需要資料時，再Page和data swap到主記憶體中。Page替換的演算法的排程方法則由LRU（Least Recently Used）達成。其實這樣的作法就是common linux os的作法，以前有玩過ubuntu server kernel的這個部分，所以有跡可循。

Implementatoin

先將multi-programming的部分做個修改，增加輔助的記憶體 SynchDisk *Swap_Space ，用來存放主記憶體放不下的Pages。

`Code`

code/userprog/userkernel.h

```
class UserProgKernel : public ThreadedKernel {
    public:
        ...
        SynchDisk *Swap_Space;  // create swap space for virtual memory
        ...
    private
        ...
};
```

初始化並分配Disk給Swap_Space，接著就能透過這個disk做swap space。

`Code`

code/userprog/userkernel.cc

```
void
UserProgKernel::Initialize()
{
    ...
    Swap_Space = new SynchDisk("New_Swap_Space");   //Create swap space for virtual memory
    ...
}
```

接著，在class Machine增加一些會使用到的變數宣告：

`Code`

code/machine/machine.h

```
class Machine {
    public:
        ...
        int  Identity;
        int SectorNum;//record sector number
        int FrameName[NumPhysPages];
        bool usedPhyPage[NumPhysPages];//record which frame in the main memory is occupied.
        bool usedVirPage[NumPhysPages];

    // start for page replacement //
        int count[NumPhysPages]; //for LRU counter
        bool reference_bit[NumPhysPages];//for second chance algorithm.
        ...
    private:
        ...
};
```

再於 addrspace.h 的 class AddrSpace 宣告必要變數 ID 和 pageTableLoaded；

ID 用來儲存thread的ID；pageTableLoaded用來存判斷page Table是否有被載入的boolean變數。

`Code`

code/userprog/addrspace.h

```
class AddrSpace {
    public:
        ...
        int ID; // store the ID of the thread
        ...
    private:
        ...
        bool pageTableLoaded;
        ...
};
```

接下來，這裡將負責載入程式碼到 memory 裡面，所以相當重要。
我在這裡開一個專屬這個 thread 的 page table，在 load 時一直往下找記憶體直到找到沒被用到的或是到底為止，以此來判斷 memory 夠不夠用，若不夠就需要用到輔助記憶體，需要存好一些重要參數以便要用的時候找得到。

修改幾乎是在下面的AddrSpace Load 函數內完成的。首先會將可執行程式碼分配到主記憶體中，並記錄相應的Page Table。如果主記憶體中的frame number足夠， Page 將被儲存到主記憶體中。當frame number不足時，後面的page將被WriteSector寫入虛擬記憶體。相應的page table也會被記錄並設置為無效。

`Code`

code/userprog/addrspace.cc

```
...
AddrSpace::AddrSpace()
{
      ID=(kernel->machine->Identity)++;
      kernel->machine->Identity=(kernel->machine->Identity)++;

}
...
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);

// 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->usedPhyPage[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;      // 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->usedPhyPage[j]=TRUE;
                      kernel->machine->FrameName[j]=ID;
                      kernel->machine->main_tab[j]=&pageTable[i];        // save the page pointer
                      pageTable[i].count++;
                      // save data to position j
                      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->usedVirPage[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->usedVirPage[tmp]=true;
                     executable->ReadAt(buffer,PageSize, noffH.code.inFileAddr+(i*PageSize));
                     kernel->Swap_Space->WriteSector(tmp, buffer); //call virtual_disk write in virtual memory

                    }
                }
    }



    if (noffH.initData.size > 0) {
        executable->ReadAt(
        &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

    delete executable;          // close file
    return TRUE;            // success
}

//----------------------------------------------------------------------
// AddrSpace::Execute
//  Run a user program.  Load the executable into memory, then
//  (for now) use our own thread to run it.
//
//  "fileName" is the file containing the object code to load into memory
//----------------------------------------------------------------------

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::InitRegisters
//  Set the initial values for the user-level register set.
//
```

```
//  We write these directly into the "machine" registers, so
//  that we can immediately jump to user code.  Note that these
//  will be saved/restored into the currentThread->userRegisters
//  when this thread is context switched out.
//----------------------------------------------------------------------

void
AddrSpace::InitRegisters()
{
    Machine *machine = kernel->machine;
    int i;

    for (i = 0; i < NumTotalRegs; i++)
    machine->WriteRegister(i, 0);

    // Initial program LRU_timeser -- must be location of "Start"
    machine->WriteRegister(PCReg, 0);

    // Need to also tell MIPS where next instruction is, because
    // of branch delay possibility
    machine->WriteRegister(NextPCReg, 4);

   // Set the stack register to the end of the address space, where we
   // allocated the stack; but subtract off a bit, to make sure we don't
   // accidentally reference off the end!
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
    DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
}

//----------------------------------------------------------------------
// 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;
    }
}
...
```

在Class TranslationEntry 中增加 ID 和 count。

`Code`

code/machine/translate.h

```
class TranslationEntry {
    public:
        ...
        int ID;

        int count;    //for Least Recently used algorithm

};
```

下面會是實現 LRU 的主要部分。首先，Page Table的有效位置將被識別以確保Page的位置，就是確認在主記憶體中還是虛擬記憶體中。如果Page在虛擬記憶體中，就會檢查現在主記憶體內是否有閒置空間。

Page Swap發生在主記憶體沒有空間的情況下，將兩個緩存區用於Swap in/Swap out。而為了達成LRU，會用一個for迴圈尋找最少使用的page，從虛擬記憶體中讀取，複製到主記憶體，選擇換出Page到虛擬記憶體，這些步驟會包含bcopy、ReadSector、WriteSector這些動作。

`Code`

code/machine/translate.cc

```
...
ExceptionType
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
{
    int i, j;
    ...
    int Swap_out_page;///find the page Swap_out_page
    ...
    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) {

          //printf("Page fault Happen!\n");
          kernel->stats->numPageFaults += 1;     // nachos pagefault counter +1
          j=0;
          while(kernel->machine->usedPhyPage[j]!=FALSE&&j<NumPhysPages)
              j += 1;                             // find valid frame space

                 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->usedPhyPage[j]=TRUE;
                      kernel->machine->FrameName[j]=pageTable[vpn].ID;
                      kernel->machine->main_tab[j]=&pageTable[vpn];  // save the page pointer

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

```

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

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

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

                    //LRU

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

                            }
                    }
                    pageTable[Swap_out_page].count++;

                    //printf("Page%d swap out!\n",Swap_out_page);

                    bcopy(&mainMemory[Swap_out_page*PageSize],buffer1,PageSize);
                    kernel->Swap_Space->ReadSector(pageTable[vpn].virtualPage, buffer2);
                    bcopy(buffer2,&mainMemory[Swap_out_page*PageSize],PageSize);
                    kernel->Swap_Space->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");

                    }

    }
    ...
}
```

Result:

1. 執行 **./nachos -e ../test/matmult**

```
u14@ubuntu:~/r11921091_nachos3/nachos-4.0/code/userprog$ ./nachos -e ../test/mat
mult
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 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
```

2. 執行 **./nachos -e ../test/sort**

```
u14@ubuntu:~/r11921091_nachos3/nachos-4.0/code/userprog$ ./nachos -e ../test/sort
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 440599030, idle 52221566, system 388377460, 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
```

3. 執行 **./nachos -e ../test/matmult -e ../test/sort**

```
u14@ubuntu:~/r11921091_nachos3/nachos-4.0/code/userprog$ ./nachos -e ../test/matmult -e
 ../test/sort
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 489532030, idle 94826365, system 394705660, user 5
Disk I/O: reads 5647, writes 5715
Console I/O: reads 0, writes 0
Paging: faults 5647
Network I/O: packets received 0, sent 0
```

p.s 如果想看細部 Page swap 的時間點可以去translate.cc 將第260行的的註解拿掉，如下：

```
printf("Page%d swap out!\n",Swap_out_page);
```

但因為 sort 的時候交換太多次，所以 linux 終端機會把 output 在視窗上洗掉，導致無法看到 matmult 和 sort 同時執行時 matmult 的 return 結果。因此，我選擇將這行先註解掉。

The difficulties I encountered

還不知道，自己寫得時候還好，畢竟以前做過類似的，唯一的問題就是sort.c的檔案本來是錯的，所以會return 0讓我很confused，但詢問過助教電神宗翰後，電神也更新了sort.c的code，所以就沒問題了！

因為這次太早寫完，所以之後同學問問題後可能會再更新。

Reference

[Nachos hw3 assignments] https://cool.ntu.edu.tw/courses/21578/files/3143901?wrap=1
[Nachos Documentation] https://homes.cs.washington.edu/~tom/nachos/