HW3_report team19 member:楊晶宇、林柏杰、常安彥

I. Trace code

1. Load program :

a. userprog/userkernel.cc - UserProgKernel::Run()

```
91  void
92  UserProgKernel::Run()
93  {
94
95      cout << "Total threads number is " << execfileNum << endl;
96      for (int n=1;n<=execfileNum;n++)
97          {
98          t[n] = new Thread(execfile[n]);
99          t[n]->space = new AddrSpace();
100         t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
101         cout << "Thread " << execfile[n] << " is executing." << endl;
102         }
103 //  Thread *t1 = new Thread(execfile[1]);
104 //  Thread *t1 = new Thread("../test/test1");
105 //  Thread *t2 = new Thread("../test/test2");
106
107 //    AddrSpace *halt = new AddrSpace();
108 //  t1->space = new AddrSpace();
109 //  t2->space = new AddrSpace();
110
111 //    halt->Execute("../test/halt");
112 //  t1->Fork((VoidFunctionPtr) &ForkExecute, (void *)t1);
113 //  t2->Fork((VoidFunctionPtr) &ForkExecute, (void *)t2);
114     ThreadedKernel::Run();
115 //  cout << "after ThreadedKernel:Run();" << endl;  // unreachable
116 }
```

UserProgKernel::Run() 是利用main thread Fork 出其他的thread，分配thread給每一個execfile，然後結束main thread的執行，就可以開始運行execfile。
實作時，用一個for迴圈一次建立一個thread給一個execfile，具體來說就是：1)new 一個Thread的空間、2)分配Thead的AddrSpace、3)Fork thread掛載它之後要執行的函式。

b. userprog/addrspace.cc - AddrSpace::AddrSpace()

```
54  AddrSpace::AddrSpace()
55  {
56      pageTable = new TranslationEntry[NumPhysPages];
57      for (unsigned int i = 0; i < NumPhysPages; i++) {
58      pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
59      pageTable[i].physicalPage = i;
60  //  pageTable[i].physicalPage = 0;
61      pageTable[i].valid = TRUE;
62  //  pageTable[i].valid = FALSE;
63      pageTable[i].use = FALSE;
64      pageTable[i].dirty = FALSE;
65      pageTable[i].readOnly = FALSE;
66      }
67
68      // zero out the entire address space
69  //    bzero(kernel->machine->mainMemory, MemorySize);
70  }
```

在一個thread中建立page table，預設是physicalPage跟i(virtual page)完全相等，也就是virtual address即為physical address。這邊全部valid都設定為true，隱含了所有virtualPage都可以載入physicalPage。不過如果virtualPage number > physicalPage number，也就是程式的size > main memory的size時，就不應該使用這樣的寫法。
因此，真正在實作時，page table的建立應該是放在AddrSpace::Load()之中，畢竟要等到load時才會知道程式的size，才知道page table的大小，才有辦法建立page table。

c. threads/kernel.cc - ThreadedKernel::Run()

```
87  void
88  ThreadedKernel::Run()
89  {
90      // NOTE: if the procedure "main" returns, then the program "nachos"
91      // will exit (as any other normal program would).  But there may be
92      // other threads on the ready list (started in SelfTest).
93      // We switch to those threads by saying that the "main" thread
94      // is finished, preventing it from returning.
95      currentThread->Finish();
96      // not reached
97  }
```

ThreadedKernel::Run() 內部只有一條指令， 它會呼叫currentThread(也就是main thread)的Finish，如此一來main thread就會釋放出CPU給readylist裡的第一個thread，開始運行executeFile。

d. threads/thread.cc - Thread::Finish()

```
173  void
174  Thread::Finish ()
175  {
176      (void) kernel->interrupt->SetLevel(IntOff);
177      ASSERT(this == kernel->currentThread);
178
179      DEBUG(dbgThread, "Finishing thread: " << name);
180
181      Sleep(TRUE);                // invokes SWITCH
182      // not reached
183  }
```

呼叫Sleep準備進行context switch，注意呼叫之前要disable interrupt，以符合Sleep的預設條件。傳入Sleep的參數是true，代表main thread finishing。

e. threads/thread.cc - Thread::Sleep (bool finishing)

```
241  void
242  Thread::Sleep (bool finishing)
243  {
244      Thread *nextThread;
245
246      ASSERT(this == kernel->currentThread);
247      ASSERT(kernel->interrupt->getLevel() == IntOff);
248
249      DEBUG(dbgThread, "Sleeping thread: " << name);
250
251      status = BLOCKED;
252      while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
253      kernel->interrupt->Idle();  // no one to run, wait for an interrupt
254
255      // returns when it's time for us to run
256      kernel->scheduler->Run(nextThread, finishing);
257  }
```

CurrentThread status改成Blocked，使用FindNextToRun取得readyList裏頭的第一個thread，做為即將要執行的nextThread，若readyList是空的，就呼叫Idle()等待。若順利取得nextThread，就呼叫Scheduler::Run()進行context switch。

f. threads/scheduler.cc - Scheduler::Run (Thread *nextThread, bool finishing)

(code很長，省略圖片)

先判斷oldThread是不是finishing，如果是就安排他在context switch之後刪除。

NextThread轉為currentThread、status改成Running然後context switch。

g. threads/switch.s - SWITCH (thread *t1, thread *t2)

(code很長，省略圖片)

在這個組語檔案中，真正被執行的是x86 block中所定義的SWITCH。程式會把EBX、ESI等*GPR的值存到Stack中oldThread的位置，再從Stack中t2的位置取出nextThread context中各個*GPR值。

*GPR:通用暫存器，可儲存資料或位址。

h. thread/thread.cc - ThreadBegin()

```
machineState[PCState] =(void *)ThreadRoot;
machineState[StartupPCState] = (void *)ThreadBegin;
machineState[InitialPCState] = (void *)func;
```

SWITCH執行完後，PC裡的值會自動導向ThreadBegin。

```
268   static void ThreadBegin() { kernel->currentThread->Begin(); }
```

currentThread呼叫Begin,

```
148   void
149   Thread::Begin ()
150   {
151       ASSERT(this == kernel->currentThread);
152       DEBUG(dbgThread, "Beginning thread: " << name);
153
154       kernel->scheduler->CheckToBeDestroyed();
155       kernel->interrupt->Enable();
156   }
```

檢查有沒有finishing的oldThread要刪除，然後重新enable interrupt。

i. userprog/userkernel.cc - ForkExecute (Thread *t)

```
85    void
86    ForkExecute(Thread *t)
87    {
88        t->space->Execute(t->getName());
89    }
```

在a.中有提到，ForkExecute規範了Thread執行時要做什麼。這邊thread用getName拿到execfile的filename，接著執行該execfile。

j. userprog/addrspace.cc - AddrSpace::Execute (char *fileName)

```
160   void
161   AddrSpace::Execute(char *fileName)
162   {
163       if (!Load(fileName)) {
164       cout << "inside !Load(FileName)" << endl;
165       return;             // executable not found
166       }
167
168       //kernel->currentThread->space = this;
169       this->InitRegisters();       // set the initial register values
170       this->RestoreState();        // load page table register
171
172       kernel->machine->Run();      // jump to the user progam
173
174       ASSERTNOTREACHED();          // machine->Run never returns;
175                                    // the address space exits
176                                    // by doing the syscall "exit"
177   }
```

先用filename把execfile load進主記憶體，然後初始化register的值。一切準備就緒，可以執行Run來一行一行讀指令了

k. userprog/addrspace.cc - AddrSpace::Load (char *fileName)
(code很長，省略圖片)
用filename把execfile load進主記憶體，這裡假設主記憶體size>execfile size，不管實際狀況硬塞，如果主記憶體size<execfile size，就會因為Assertion而coredump。
要是主記憶體size<execfile size應該要有其他的load方法，這也是之後實作multiprogramming及virtual memory時需要更改的部分。

2. Page Faults

a. machine/ mipssim.cc - Machine::Run()

```
51  void
52  Machine::Run()
53  {
54      Instruction *instr = new Instruction;  // storage for decoded instruction
55
56      if (debug->IsEnabled('m')) {
57          cout << "Starting program in thread: " << kernel->currentThread->getName();
58      cout << ", at time: " << kernel->stats->totalTicks << "\n";
59      }
60      kernel->interrupt->setStatus(UserMode);
61      for (;;) {
62          OneInstruction(instr);
63      kernel->interrupt->OneTick();
64      if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
65        Debugger();
66      }
67  }
```

Thread開始執行時呼叫，要一個空間給decoded instruction，然後把該空間位址傳給OneInstruction()。

b. machine/ mipssim.cc - Machine::OneInstruction(Instruction *instr)
(code很長，省略圖片)
使用ReadMem，從PC指向的記憶體位置取出intruction並decode，如果失敗；稱攻讀取instruction後，先紀錄PC_after但不存入NextPCReg，並根據decode出的opcode去做不同的動作。

c. machine/translate.cc - Machine::ReadMem(int addr, int size, int *value)
(code很長，省略圖片)
先呼叫translate()，把virtual address轉成physical address，接著用physical address 去main Memory把資料讀出來，回傳給OneInstruction做解碼。

d. machine/translate.cc - Machine::Translate(int VA, int* PA, int size, bool writing)
(code很長，省略圖片)
基本功能就是接收virtual Address，透過current thread的page table轉換成physical address 後存好給ReadMem或WriteMem使用；除此之外，檢查Exception是否發生，其中最重要的就是透過valid、確定page是否在page table中，若沒有就回傳PageFaultException。
還有許多其他不同的Exception，像是virtual page number若超過page table entry number就回傳AddressErrorException，如果current thread想要Write一個read-only的page就回傳ReadOnlyException等等。
當然，若一切都順利執行，要回傳NoException。

e. machine/machine.cc - Machine::RaiseException(Exception which, int badVAddr)
(code很長，省略圖片)
切換為SystemMode後呼叫ExceptionHandler，結束ExceptionHandler後回到UserMode。也會紀錄發生Exception的address，需要的話可以使用。

f. userprog/exception.cc - ExceptionHandler(ExceptionType which)

(code很長，省略圖片)

根據不同的Exception運行對應的函式。本次作業要在這邊寫下如何處理PageFaultException。

II. Implement

1. Use Empty file to save the extra code.

a. userprog/addrspace.cc - Load(char* fileName)

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

    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;
    size = numPages * PageSize;

    pageTable = new TranslationEntry[numPages];
    for (unsigned int i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
        pageTable[i].physicalPage = i;
    //  pageTable[i].physicalPage = 0;
        pageTable[i].valid = FALSE;
    //  pageTable[i].valid = FALSE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
```

由於UserStackSize會被調大，導致virtual mem size也會也會一起大到超過physical mem size，所以更改原本的code，把pagetable放到這裡宣告，根據計算出的size(包含code, data, stack)要相對應大小的table，並initialize table。

```
char ch = 'X';
char *temp = new char[strlen(fileName) + 1];
strcpy(temp, fileName);
strncat(temp, &ch, 1);
kernel->fileSystem->Create(temp);
DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);
int remainSize = noffH.code.size;
int currentPosition = noffH.code.inFileAddr;
int currentPosition2 = noffH.code.virtualAddr;
OpenFile *emptyFile = kernel->fileSystem->Open(temp);
while (remainSize > 0)
{
    int s;
    if(remainSize >= PageSize) s = PageSize;
    else s = remainSize;

    DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
    executable->ReadAt(&(kernel->machine->mainMemory[noffH.code.virtualAddr]), s, currentPosition);
    emptyFile->WriteAt(&(kernel->machine->mainMemory[noffH.code.virtualAddr]), s, currentPosition2);
    remainSize -= s;
    currentPosition += s;
    currentPosition2 += s;

};
```

因為作業要求要create 新的file去當作virtual mem space, 所以我們用filesystem的Create函數, 創了一個file, 檔名為原本檔名末端加一個'X', 並且搬動code跟data進去新的file裡。這裡有一個重點是從原本file的code.inFileAddr對應到新的file的position應該要是code.virtualAddr, 這樣之後在ReadAt與WriteAt才能夠直接用virtualAddr去新的file裡搬運資料。

```
    // }

    remainSize = noffH.initData.size;
    currentPosition = noffH.initData.inFileAddr;
    currentPosition2 = noffH.initData.virtualAddr;
    while (remainSize > 0)
    {
        int s;
        if(remainSize >= PageSize) s = PageSize;
        else s = remainSize;

        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(&(kernel->machine->mainMemory[noffH.code.virtualAddr]), s, currentPosition);
        emptyFile->WriteAt(&(kernel->machine->mainMemory[noffH.code.virtualAddr]), s, currentPosition2);
        remainSize -= s;
        currentPosition += s;
        currentPosition2 += s;
    };
    delete emptyFile;
    delete temp;
    delete executable;          // close file
    return TRUE;            // success
}
```

最後, 把initdata用同樣的方式搬進新的file裡, delete兩個file(關閉檔案)

2. Least Recently Used (LRU) -- Stack Implementation with Vector

a. machine/stats.h - class Statistics{}

```
bool PageFreeTable[NumPhysPages] = {0};
vector<int> LRU;
int FindFreePage();
```

在Statistics中先創建一個table--PageFreeTable，用來記錄Physical pages有沒有被使用到，0 / FALSE代表是free的，1 / TRUE代表被佔用了。
接著創建一個名為LRU的vector，用來實踐LRU的演算，讓最近被用到的physical page排在後面，而越前面的會越早被erase掉。
最後定義一個函式FindFreePage，會在PageFreeTable中尋找free的pages(值為false)，如果有就回傳page的number，沒有的話就回傳-1。(如下)

```
int Statistics::FindFreePage() {
  // cout << "into findfreepage" << endl;
  // cout << "free2 =  " <<  PageFreeTable[2] << endl;
  for(int i = 0; i < NumPhysPages; i++) {
    // cout << "i = " << i << "val = " << PageFreeTable[i] << endl;
    if(!PageFreeTable[i]) {
      PageFreeTable[i] =1;
      return i;
    }
  }
  return -1;
}
```

b. machine/translate.cc -- Translate(int virtAddr, int* physAddr, int size, bool writing)

```
    for(i = 0; i < kernel->stats->LRU.size(); i++)
    {
        if(kernel->stats->LRU[i] == pageFrame){
            DEBUG(dbgAddr, "ERASE" << pageFrame);
            kernel->stats->LRU.erase(kernel->stats->LRU.begin() + i);
            DEBUG(dbgAddr, "PUSHBACK" << pageFrame);
            kernel->stats->LRU.push_back(pageFrame);
        }
    }
    return NoException;
}
```

在translate function中，如果順利到達函式最後要return NoException的地方，就代表要使用的page有其對應的physical page，所以不會產生page fault。
而為了實踐LRU，沒有產生page fault的page要再被移到vector的底部，作為最近有被用到的證明，被替換的順位會移到最後。如上圖，找到這個Page frame編號在LRU vector的位置，之後erase掉並重新push back在後面。

c. userprog/exception.cc -- ExceptionHandler(ExceptionType which)

case PageFaultException

```
51  ExceptionHandler(ExceptionType which)
52  {
53      int type = kernel->machine->ReadRegister(2);
54      int val, status;
55      int victim;
56
57      switch (which) {
58      case PageFaultException:
59          DEBUG(dbgAddr, "StartFindingFree"<<kernel->stats->LRU.size());
60          victim = kernel->stats->FindFreePage();
61          kernel->stats->numPageFaults++;
62          if(victim != -1)
63          {
64              kernel->swapIn(victim);
65              DEBUG(dbgAddr, "OnlyPush"<<victim);
66              kernel->stats->LRU.push_back(victim);
67          }else{
68              victim = kernel->stats->LRU[0];
69              kernel->swapOut(victim);
70              kernel->swapIn(victim);
71              DEBUG(dbgAddr, "Erase&Push"<<victim);
72              kernel->stats->LRU.erase(kernel->stats->LRU.begin());
73              kernel->stats->LRU.push_back(victim);
74          }
75          return;
```

如果translate()回傳PageFaultException, 則Raise exception之後要來處理page fault的問題: 首先要查看是否有free的physical page可以直接拿來用，實作方法就是呼叫先前定義的Statistic::FindFreePage()。如果回傳一個非-1的數, 即可直接swap in該number的page frame之中, 並將number push back進LRU vector後面代表最近被用到。

反之, 如果回傳的值是-1就要進行replacement，而在LRU vector中第一項就是首當其衝要被替換掉的。讓victim代表第一位, 也就是最久遠以前被使用到的frame的number, 然後帶進UseProgKernel::swapOut()以及UserProgKernel::swapIn() function裡面, 去將資料做出搬出搬進的動作。最後, 把第一位從vector中erase掉, 重新Push back回尾端, 代表最近被使用到。

d. userprog/userkernel.cc -- swapOut(int victim), swapIn(int victim)

```
154
155   void
156   UserProgKernel::swapOut(int victim)
157   {
158       for(int i = 1; i <= execfileNum; i++)
159           if(t[i]->space->swapOut(victim, t[i]->getName()))
160               break;
161   }
162
163   void
164   UserProgKernel::swapIn(int victim)
165   {
166       // cout << "UserProgKernel ReadRegister(BadVAddrReg) = " << machine->ReadI
167       currentThread->space->swapIn(victim, machine->ReadRegister(BadVAddrReg));
168   }
169
```

swapOut : 每個執行中的thread去呼叫swapOut, 參數為victim(被swapout的pagenum)以及filename, 因為不知道victim page裡面裝的是哪個thread的資料, 所以如果return 1代表成功找到並swapout。
swapIn:Register[BadVAddrReg](根據描述, Addressing失敗的virtual addr都會存在此reg)所以將此參數丟入currentThread的space去做swapIn

e. userprog/addrspace.cc -- swapIn

```
void AddrSpace::swapIn(int victim, unsigned int virtAddr) {
    unsigned int vpn = (unsigned) virtAddr / PageSize;
    // cout << "spaceswapIn vic =" << victim << "virtAddr  = " << virtAddr << endl;
    char *fileName = kernel->currentThread->getName();
    // cout << fileName << endl;
    char ch = 'X';
    char *temp = new char[strlen(fileName) + 1];
    strcpy(temp, fileName);
    strncat(temp, &ch, 1);
    // cout << temp << endl;
    OpenFile *file = kernel->fileSystem->Open(temp);
    // cout << victim * PageSize << " " << PageSize << " " << virtAddr << endl;
    file->ReadAt(&(kernel->machine->mainMemory[victim * PageSize]), PageSize, vpn* PageSize);
    delete file;
    delete temp;
    // cout << "spaceswapIn-t2" << endl;
    kernel->machine->pageTable[vpn].physicalPage = victim;
    kernel->machine->pageTable[vpn].valid = true;
    kernel->currentThread->space->pageTable[vpn].physicalPage = victim;
    kernel->currentThread->space->pageTable[vpn].valid = true;

}
```

進來的virtAddr就是剛剛userkernel傳入的Register[BadVAddrReg]的值，所以做的事情就是打開load時我們創來當mem的file, 換算virtualAddr所在的pagenum(vpn), 去file裡面的vpn的起始位置(vpn*PageSize)拿PageSize大小的資料, 並且放到mainMemory的victim page(該page已被swapout或是本來就是空的), 最後把pagetable的valid bit設為true, physicalPage設為victim。

f. userprog/addrspace.cc -- swapOut

```
bool AddrSpace::swapOut(int victim, char *fileName)
{
    // cout << "spaceswapOut vic = " << victim << "fileName = " << endl;
    for(int i = 0; i < numPages; i++) {
        if(pageTable[i].physicalPage == victim && pageTable[i].valid){
            if(pageTable[i].dirty) {
                char ch = 'X';
                char *temp = new char[strlen(fileName) + 1];
                strcpy(temp, fileName);
                strncat(temp, &ch, 1);
                // cout << temp << endl;
                OpenFile *file = kernel->fileSystem->Open(temp);
                file->WriteAt(&(kernel->machine->mainMemory[victim * PageSize]), PageSize, i*PageSize);
                delete file;
                delete temp;
            }
            pageTable[i].physicalPage = 0;
            pageTable[i].valid = FALSE;
            pageTable[i].use = FALSE;
            pageTable[i].dirty = FALSE;
            pageTable[i].readOnly = FALSE;
            return true;
        }
    }
    return false;
}
```

原理和swapIn差不多，如果這個thread的pagetable存在victim page且valid bit是1，若dirtybit是1，就把現在mem裡的victim page存入file中對應的位置(i代表是virtual page num)。最後把pagetable validbit、dirtybit都設為0(reset)

III. Result

```
jinyu@ubuntu:~/hw3/nachos-master-564d0473dc44505ad78cb715aab73080e721b4fe-nachos-4.0-hw1/nachos-4.0-hw1/code$ ./user
Total threads number is 3
Thread ./test/merge is executing.
Thread ./test/quick is executing.
Thread ./test/bubble is executing.
Print integer:804
Print integer:804
Print integer:805
Print integer:806
Print integer:807
return value:1
Print integer:20
Print integer:20
Print integer:21
Print integer:22
Print integer:22
return value:2
Print integer:909
Print integer:909
Print integer:910
Print integer:914
Print integer:915
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 62525700, idle 32, system 6252630, user 56273038
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 26733
Network I/O: packets received 0, sent 0
jinyu@ubuntu:~/hw3/nachos-master-564d0473dc44505ad78cb715aab73080e721b4fe-nachos-4.0-hw1/nachos-4.0-hw1/code$
```

經過觀察，產生最多page fault(PF)的program為bubble sort，因為它會一直重複allocate arr[0] ~arr[1023]，所以我們透過調大pagesize，調小NumPhysPages(in machine.h)去讓buuble sort一次allocate較大的資料量減少PF(太大也不行，proccess切換會造成更多PF)，最後發現最佳的size落在168，NumPhysPages = 24(NumPhysPages * PageSize = 4032)。

執行結果共有26733次page faults

Contribution: 楊晶宇(33%) 常安彥(33%) 林柏杰(33%)