Lab2 虚拟内存 实验报告

内容一：总体概述

内容二：任务完成情况

任务完成列表

Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise7	Challenge2
Y	Y	Y	Y	Y	Y	Y	Y

Exercise 1 TLB异常处理-源代码阅读

> 1. 阅读code/userprog/progtest.cc，着重理解nachos执行用户程序的过程，以及该过程中与内存管理相关的要点。
> 2. 阅读code/machine目录下的machine.h(cc)，translate.h(cc)文件和code/userprog目录下的exception.h(cc)，理解当前Nachos系统所采用的TLB机制和地址转换机制。

progtest.cc--执行用户程序过程

讨论：

noff执行格式，在test/makefile可以看到

程序入口：/threads/main.cc 检测到-x参数则调用/userprog/progtest.cc 的 startProcess函数。

```
void StartProcess(char *filename)
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
  printf("Unable to open file %s\n", filename);
  return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;      // close file

    space->InitRegisters();   // set the initial register values
    space->RestoreState();    // load page table register
```

```
    machine->Run();     // jump to the user progam
    ASSERT(FALSE);      // machine->Run never returns;
          // the address space exits
          // by doing the syscall "exit"
}
```

函数执行步骤：

1. 打开可执行文件
2. `AdderSpace()`：为可执行文件创建新的地址空间。其过程首先检查可执行文件格式（小端则转化为大端），根据代码段、初始化数据段、未初始化数据段、用户栈大小之和得到地址空间大小，整除页表大小得到页数，然后建立页表：即以`TranslationEntry`为元素的数组。然后利用`executable->ReadAt()`中调用`bcopy()`将代码段与数据段加载到`machine->mainMemory`即内存中。

   ```
   pageTable = new TranslationEntry[numPages];   //建立页表，之后会将cpu的页表指针指向它
   for (i = 0; i < numPages; i++) {
       pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
       pageTable[i].physicalPage = i;
       pageTable[i].valid = TRUE;
       pageTable[i].use = FALSE;
       pageTable[i].dirty = FALSE;
       pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
       // a separate page, we could set its
       // pages to be read-only
   }
   ```

3. `space->InitRegisters()`初始化cpu寄存器的值、置cpu页表指针与页表大小为当前用户程序页表指针与页表大小
4. 运行用户程序`machine->Run()`中`OneInstruction()`来执行指令，取址译码执行，每一次执行后使时钟前进。

   ```
   Machine::Run() {
       ...
       while (1) {
           OneInstruction(instr);
           ...
       }
   }
   ```

```
/*  mipssim.cc Machine::OneInstruction(Instruction *instr)  */
if (!machine->ReadMem(registers[PCReg], 4, &raw))    //读pc寄存器，即读指令到
raw
  return;      // exception occurred
    instr->value = raw;
    instr->Decode();     //译码，将指令解析成opCode和寄存器的值用于执行运算
```

对于不同的指令/opcode执行不同的操作，检测出异常则通过`RaiseException`进入异常处理函数。由于执行指令过程是在`machine->Run()`的for循环中，如有异常退出`OneInstruction()`时PC寄存器还没有更新，所以下次进来再次执行同一条指令。

```
switch (instr->opCode) {
  case OP_ADD:
    sum = registers[instr->rs] + registers[instr->rt];
    if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
        ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
        RaiseException(OverflowException, 0);    //主动进入异常处理
        return;    //异常处理完毕，退出OneInstruction()，pc未更新，于是重新运行一次
本条指令
    }
    registers[instr->rd] = sum;
    break;
  /*
  case ....
  */
}
```

需要读懂`NoffHeader` `OpenFile` `Instruction`类

TLB机制和地址转换机制

`machine.h` `machine.cc`以及一系列定义在其他文件中的machine类的函数是对cpu的模拟，包含了用户程序的运行过程、对寄存器的读写、对内存的读写。

声明与定义物理内存、TLB、页表、寄存器：

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

TranslationEntry *pageTable;
char *mainMemory;    // physical memory to store user program,
                     // code and data, while executing
int registers[NumTotalRegs]; // CPU registers, for executing user programs
```

```
for (i = 0; i < NumTotalRegs; i++)  //初始化寄存器
    registers[i] = 0;
```

```
mainMemory = new char[MemorySize];  //为物理内存分配空间
for (i = 0; i < MemorySize; i++)
    mainMemory[i] = 0;

#ifdef USE_TLB
tlb = new TranslationEntry[TLBSize];  //为TLB分配空间
for (i = 0; i < TLBSize; i++)
    tlb[i].valid = FALSE;
pageTable = NULL;
#else // use linear page table
tlb = NULL;
pageTable = NULL;   //在startProcess中指向了用户程序的页表
```

实现地址转换在translate.cc 中的`Machine::Translate()`

1. 进行地址对齐，比如要读4字节，则要求地址最后两位为00，否则抛出地址错误异常。

```
// check for alignment errors 地址对齐，size是读/写几个字节
if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1)))
{
  DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);
  return AddressErrorException;
}
```

2. 计算virtual page number、virtual page offset

```
vpn = (unsigned) virtAddr / PageSize;
offset = (unsigned) virtAddr % PageSize;
```

3. 如果使用页表，判断vpn是否超出页表长度（超出则地址异常）、页表项是否有效（无效则缺页异常）；如果使用tlb则在其中寻找相应页表项，没有则抛出缺页异常。

```
if (tlb == NULL) {    // => page table => vpn is index into table
  if (vpn >= pageTableSize) {
      DEBUG('a', "virtual page # %d too large for page table size %d!\n",
      virtAddr, pageTableSize);
      return AddressErrorException;
  } else if (!pageTable[vpn].valid) {
      DEBUG('a', "virtual page # %d too large for page table size %d!\n",
      virtAddr, pageTableSize);
      return PageFaultException;
    }
  entry = &pageTable[vpn];
} else {
        for (entry = NULL, i = 0; i < TLBSize; i++)
          if (tlb[i].valid && (tlb[i].virtualPage == vpn)) {
              entry = &tlb[i];      // FOUND!
```

```
                break;
            }
        if (entry == NULL) {        // not found
            DEBUG('a', "*** no valid TLB entry found for this virtual
page!\n");
            return PageFaultException;    // really, this is a TLB fault,
            // the page may be in memory,
            // but not in the TLB
        }
}
```

4. 检查读写权限、检查物理页号是否超出物理页数（声明在machine.h为32，每页128字节）
5. 标记页表的use位，如果是写入则同时标记dirty位
6. 利用物理页号和vpo计算物理地址

Exercise 2+3 TLB MISS异常处理

在`code/userprog/Makefile`中添加

```
DEFINES = -DUSE_TLB
```

阅读代码可知在抛出缺页异常时，machine.cc中`RaiseException`首先将导致缺页的虚拟地址放到`BadVAddrReg`寄存器中，然后调用`ExceptionHandler`函数进行中断处理。

```
void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);

//  ASSERT(interrupt->getStatus() == UserMode);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);      // finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which);    // interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
```

在`ExceptionHandler`中添加对缺页异常的处理

```
if(which == PageFaultException) {
    int BadVAddr = machine->ReadRegister(BadVAddrReg);    //从寄存器中取得导致缺页的虚拟地址
    if(machine->tlb != NULL){
        DEBUG('a', "=> TLB miss (no TLB entry)\n");
        machine->tlbReplace(BadVAddr);    //调用tlb miss处理函数
    }
}
```

FIFO

```
void Machine::tlbReplaceTLBFIFO(int BadVAddr){
    unsigned int vpn = BadVAddr / PageSize;
    int position = -1;

    for(int i = 0; i < TLBSize; i ++){    //如果有失效位则直接替代
        if(tlb[i].valid == false){
            position = i;
            break;
        }
    }
    if(position == -1){     //否则把TLB第一位丢掉（FIFO），后面的前移
        for(int i = 0; i < TLBSize - 1; i ++)
            tlb[i] = tlb[i + 1];
        position = TLBSize - 1;
    }
    tlb[position].virtualPage = pageTable[vpn].virtualPage;
    tlb[position].physicalPage = pageTable[vpn].physicalPage;
    tlb[position].valid = true;
    tlb[position].use = false;
    tlb[position].dirty = false;
}
```

LRU

为页表/TLB项增加一个`lastUsedTime`变量，记录上次使用间隔，并在初始化TLB后赋值为0

```
class TranslationEntry {
  public:
    ......
    int lastUsedTime;    //越大说明越久没用了
};
```

```
Machine::Machine(bool debug)
{
    ......
    #ifdef USE_TLB
    tlb = new TranslationEntry[TLBSize];
    for (i = 0; i < TLBSize; i++){
        tlb[i].valid = FALSE;
        tlb[i].lastUsedTime = 0;
    }
    ......
    #else // use linear page table
    ......
}
```

每次访问TLB时（即在地址翻译过程中）给取的TLB项的`lastUsedTime`减一（若为0则不变）

```
/* Machine::translate() */
for (entry = NULL, i = 0; i < TLBSize; i++)
  if (tlb[i].valid && (tlb[i].virtualPage == vpn)) {
      entry = &tlb[i];      // FOUND!
      if(tlb[i].lastUsedTime > 0)
          tlb[i].lastUsedTime--;
      break;
  }
```

每次TLB miss时，对TLB进行置换，给所有项此值加一，新项此值为0

```
void Machine::tlbReplaceTLBFIFO(int BadVAddr){
    unsigned int vpn = BadVAddr / PageSize;
    int position = -1;
    for(int i = 0; i < TLBSize; i ++){
        if(tlb[i].valid == false){
            position = i;
            break;
        }
    }
    if(position == -1){
        int maxLUT = 0;
        for(int i = 0; i < TLBSize; i ++){
            if(tlb[i].lastUsedTime > maxLUT){
                position = i;
                maxLUT = tlb[i].lastUsedTime;
                continue;
            }
            tlb[i].lastUsedTime++;
        }
    }
```

```
    tlb[position].virtualPage = pageTable[vpn].virtualPage;
    tlb[position].physicalPage = pageTable[vpn].physicalPage;
    tlb[position].valid = true;
    tlb[position].use = false;
    tlb[position].dirty = false;
    tlb[position].lastUsedTime = 0;    //马上访问故为0
}
```

测试

给thread增加`tlbtimes` `tlbhits`变量，每次翻译地址时若使用TLB则前者加一，若命中则后者也加一。利用test/sort.c进行测试，因为还没实现Exit()系统调用，所以将sort.c最后调用Exit()改为调用Halt()。在machine的析构函数中输出TLB命中情况。

- FIFO

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Machine halting!

Ticks: total 8935, idle 0, system 10, user 8925
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
tlbtimes: 11214 tlbhits: 10614  accuracy: 0.946495
```

- LRU

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Machine halting!

Ticks: total 8754, idle 0, system 10, user 8744
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
tlbtimes: 10842 tlbhits: 10423  accuracy: 0.961354
```

Exercise 4 内存全局管理数据结构

> 设计并实现一个全局性的数据结构（如空闲链表、位图等）来进行内存的分配和回收，并记录当前内存的使用状态。

machine.h中声明bitmap,定义allocMem、freeMem为用户程序分配和清除物理内存

```
/*  machine.h  */
class Machine{
  public:
    ......
    int tlbSize;
    int allocMem(){ return mybitmap->Find(); }
    void freeMem();

  private:
    BitMap *mybitmap;
}
```

```
/*  machine.cc  */
Machine::Machine(bool debug)
{
  ......
  mybitmap = new BitMap(NumPhysPages);
  ......
}

void Machine::freeMem(){
  for(int i = 0; i < NumPhysPages; i++){
    if(pageTable[i].valid && mybitmap->Test(i))   //通过页表遍历知道哪些物理页存储的是当前用户程序的数据
      mybitmap->Clear(i);     //然后将对应位图的位置0
  }
}
```

按页装载机：给用户线程分配空间时，初始化页表时利用位图分配物理地址，并在装载程序与数据时注意读入的地址，一定要注意用物理地址乘页表大小！这样才是按页分配。

```
AddrSpace::AddrSpace(OpenFile *executable)
{
    ......
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
        pageTable[i].physicalPage = machine->allocMem();    //利用位图分配物理地址
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
    ......
    if (noffH.code.size > 0) {
        executable->ReadAt(&(machine-
>mainMemory[pageTable[noffH.code.virtualAddr].physicalPage * PageSize]),
```

```
      noffH.code.size, noffH.code.inFileAddr);     //将代码读到物理内存的某一物理页的开头，这里一定乘页表大小
        // executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
    //  noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        executable->ReadAt(&(machine->mainMemory[pageTable[noffH.initData.virtualAddr].physicalPage * PageSize]),
      noffH.initData.size, noffH.initData.inFileAddr);
    }

}
```

测试，在bitmap找到空位时输出分配的物理页号（1位对应1页）

```
/* bitmap.cc */
int BitMap::Find()
{
    for (int i = 0; i < numBits; i++)
  if (!Test(i)) {
      Mark(i);
        printf("Allocated %d\n", i);
      return i;
  }
    return -1;
}
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Allocated 0
Allocated 1
Allocated 2
Allocated 3
Allocated 4
Allocated 5
Allocated 6
Allocated 7
Allocated 8
Allocated 9
Allocated 10
Allocated 11
Allocated 12
Allocated 13
Allocated 14
Exiting userprog of thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 9631, idle 0, system 10, user 9621
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Thread: main    tlbtimes: 10845 tlbhits: 10426  accuracy: 0.961365
```

Exercise 5 多线程支持

> 目前Nachos系统的内存中同时只能存在一个线程，我们希望打破这种限制，使得Nachos系统支持多个线程同时存在于内存中。

实现Exit系统调用

```
if(which == SyscallException && type == SC_Exit){
    printf("Exiting userprog of thread: %s\n", currentThread->getName());
    machine->freeMem();     //将退出的用户程序占用的物理页在位图中置零
    currentThread->Finish();    //线程结束
    if(currentThread->space!=NULL){
        delete currentThread->space;
    }
}
```

修改`AddrSpace::SaveState()`，用户线程让出cpu时需要使TLB各位无效！如果不这样做，在用户程序切换时，新上cpu的用户程序将会使用切换前用户程序的物理页（存在TLB里），将会出错。

```
void AddrSpace::SaveState()
{
    for(int i = 0; i < machine->tlbSize; i++){
        machine->tlb[i].valid = false;
    }
}
```

写测试，两个用户程序分别运行/test/fibonacci与/test/sort

```
/*  progtest.cc  */
void MultUserprogFunc(int which){
    char *filename = "../test/fibonacci";
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
  printf("Unable to open file %s\n", filename);
  return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;      // close file

    currentThread->space->InitRegisters();    // set the initial register
values
    currentThread->space->RestoreState();   // load page table register
```

```
    machine->Run();     // jump to the user progam
    ASSERT(FALSE);

}

void
StartMultProcess(char *filename)
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
  printf("Unable to open file %s\n", filename);
  return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;      // close file

    space->InitRegisters();   // set the initial register values
    space->RestoreState();    // load page table register

    Thread *thread2 = new Thread("Thread2");
    thread2->Fork(MultUserprogFunc, 2);

    // currentThread->Yield();
    machine->Run();     // jump to the user progam
    ASSERT(FALSE);      // machine->Run never returns;
          // the address space exits
          // by doing the syscall "exit"
}
```

在`machine::Run()`中设置每1000条指令切换用户程序时

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Allocated 0
Allocated 1
Allocated 2
Allocated 3
Allocated 4
Allocated 5
Allocated 6
Allocated 7
Allocated 8
Allocated 9
Allocated 10
Allocated 11
Allocated 12
Allocated 13
Allocated 14
Allocated 15
Allocated 16
Allocated 17
Allocated 18
Allocated 19
Allocated 20
Allocated 21
Allocated 22
Allocated 23
Allocated 24
Allocated 25
Allocated 26
Exiting userprog of thread: Thread2
Thread: Thread2 tlbtimes: 4743  tlbhits: 4448   accuracy: 0.937803
Exiting userprog of thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 12549, idle 0, system 30, user 12519
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Thread: main    tlbtimes: 10862 tlbhits: 10433  accuracy: 0.960505
```

每10条指令切换用户程序：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Allocated 0
Allocated 1
Allocated 2
Allocated 3
Allocated 4
Allocated 5
Allocated 6
Allocated 7
Allocated 8
Allocated 9
Allocated 10
Allocated 11
Allocated 12
Allocated 13
Allocated 14
Allocated 15
Allocated 16
Allocated 17
Allocated 18
Allocated 19
Allocated 20
Allocated 21
Allocated 22
Allocated 23
Allocated 24
Allocated 25
Allocated 26
Exiting userprog of thread: Thread2
Thread: Thread2 tlbtimes: 6241  tlbhits: 4929   accuracy: 0.789777
Exiting userprog of thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 16097, idle 0, system 30, user 16067
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Thread: main    tlbtimes: 12517 tlbhits: 11005  accuracy: 0.879204
```

可以看到两个用户程序同时存在内存，也可看到切换频率对tlb命中率的影响。

Exercise 6 缺页中断处理

> 基于TLB机制的异常处理和页面替换算法的实践，实现缺页中断处理（注意！TLB机制的异常处理是将内存中已有的页面调入TLB，而此处的缺页中断处理则是从磁盘中调入新的页面到内存）、页面替换算法等。

每次缺页readAt一页，虚拟地址与文件中的虚拟地址是一一对应的，详看readAt、noff格式

Exercise 7 Lazy-loading

> 我们已经知道，Nachos系统为用户程序分配内存必须在用户程序载入内存时一次性完成，故此，系统能够运行的用户程序的大小被严格限制在4KB以下。请实现Lazy-loading的内存分配算法，使得当且仅当程序运行过程中缺页中断发生时，才会将所需的页面从磁盘调入内存。

Challenge 2 倒排页表

> 多级页表的缺陷在于页表的大小与虚拟地址空间的大小成正比，为了节省物理内存在页表存储上的消耗，请在Nachos系统中实现倒排页表。

将tlbmiss和缺页中断的中断区分开

```
enum ExceptionType { NoException,           // Everything ok!
         SyscallException,      // A program executed a system call.
         PageFaultException,    // No valid translation found
         ......
         TLBMissException    //新增
};
```

给TLB异常处理中增加缺页判断

```
if(which == TLBMissException) {
  int BadVAddr = machine->ReadRegister(BadVAddrReg);
  unsigned int vpn = BadVAddr / PageSize;
  if(machine->tlb != NULL){
    DEBUG('a', "=> TLB miss (no TLB entry)\n");
    if(machine->pageTable[vpn].valid == true)    //如果页表对应项valid，就从页表取到
TLB
      machine->tlbReplace(BadVAddr);
    else{                                        //否则，先抛出页表缺页异常
        machine->RaiseException(PageFaultException, BadVAddr);
        machine->tlbReplace(BadVAddr);
    }
  }
}
```

在Machine中增加内存位图、ppnToThread数组、ppnTovpn倒转页表、虚拟磁盘、磁盘位图、分配物理内存页、分配虚拟磁盘页、页面替换等函数

```
class Machine{
  public:
    ......
    int allocMem(){ return memBitMap->Find(); }
    void freeMem();
    int pageReplace();     //随机返回一个页号
    int allocDisk(){ return diskBitMap->Find(); }
```

```
    char *disk;
    BitMap *memBitMap;
    BitMap *diskBitMap;
    Thread *ppnToThread[NumPhysPages];  //指明各个页属于哪个线程
    int * ppnTovpn;   //与ppnToThread共同构成倒排页表
}
```

在Thread中增加vpnTodpn数、filename文件名、给页表项置为invalid的函数

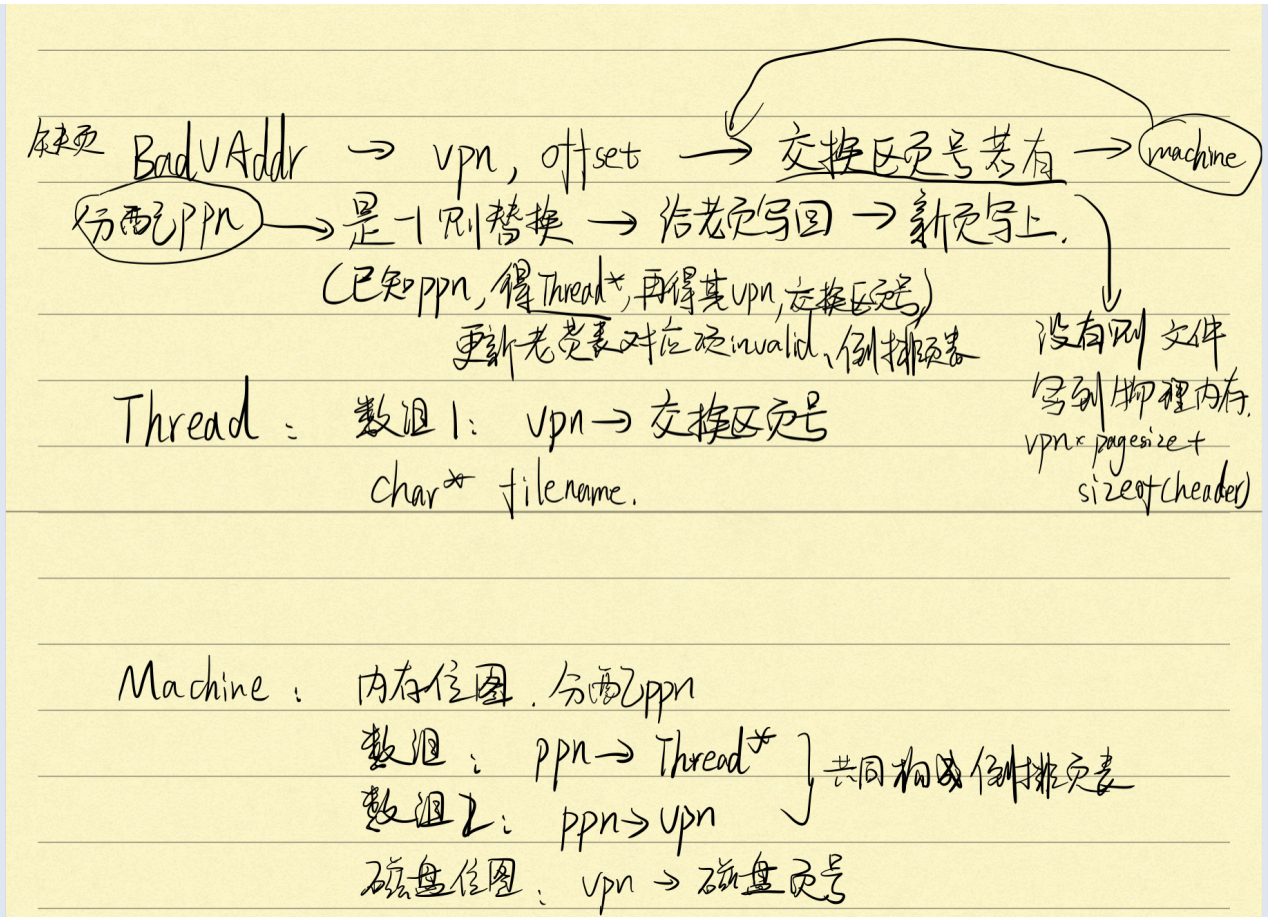
```
class Thread{
  #ifdef USER_PROGRAM
    ......
  public:
    ......
    void setInvalid(int vpn){ this->space->setInvalid(vpn); }
    int *vpnTodpn;
    char *filename;
#endif
}
```

在exception.cc中增加缺页异常处理

```
/*  exception.cc  */

if(which == PageFaultException){
  //求出缺页的虚拟页号
  int BadVAddr = machine->ReadRegister(BadVAddrReg);
  unsigned int vpn = BadVAddr / PageSize;

  //获得一个物理页号，-1则需要执行页面替换
  int ppn = machine->allocMem();
  if(ppn == -1){
    //得到被替换页面属于哪个线程、对应vpn、对应磁盘位置
    ppn = machine->pageReplace();
    Thread* FormThread = machine->ppnToThread[ppn];
    int FormVpn = machine->ppnTovpn[ppn];   //不需要考虑FormVpn为-1，因为肯定已经建立映射
    if(FormThread->vpnTodpn[FormVpn] == -1){  //如果磁盘中曾经没有这一页，则给这一页分配磁盘页
      FormThread->vpnTodpn[FormVpn] = machine->allocDisk();
    }
    int FormDpn = FormThread->vpnTodpn[FormVpn];
    //将物理页内容写回磁盘
    memcpy(machine->disk + FormDpn * PageSize, machine->mainMemory + ppn * PageSize, PageSize);
    FormThread->setInvalid(FormVpn);    //使原本这一页的线程对应页表项invalid
    printf("Write Thread: %s\tvpn %d\tppn %d back to disk\n", FormThread->getName(), FormVpn, ppn);
```

```
  }

  //将文件对应页读入物理内存

  int dpn = currentThread->vpnTodpn[vpn];
  if(dpn != -1){    //如果虚拟磁盘里有，则从虚拟磁盘读到内存
    memcpy(machine->mainMemory + ppn * PageSize, machine->disk + dpn *
PageSize, PageSize);
    printf("Thread: %s\tRead Page %d form Disk to mainMemory\n", currentThread-
>getName(), ppn);
  }
  else    //  虚拟磁盘里没有，则从文件直接读到内存
  {
    OpenFile *executable = fileSystem->Open(currentThread->filename);
    executable->ReadAt(&(machine->mainMemory[ppn * PageSize]), PageSize, vpn *
PageSize + sizeof(NoffHeader));
    printf("Thread: %s\tRead Page %d form FILE to mainMemory\n", currentThread-
>getName(), ppn);
  }

  //修改页表与倒排页表
  machine->pageTable[vpn].valid = true;
  machine->pageTable[vpn].physicalPage = ppn;
  machine->ppnToThread[ppn] = currentThread;
  machine->ppnTovpn[ppn] = vpn;
}
```

下面是手写的流程



测试：首先在addrspace.cc中初始化页表时将各页设为invalid，并不分配物理页，也不读入文件

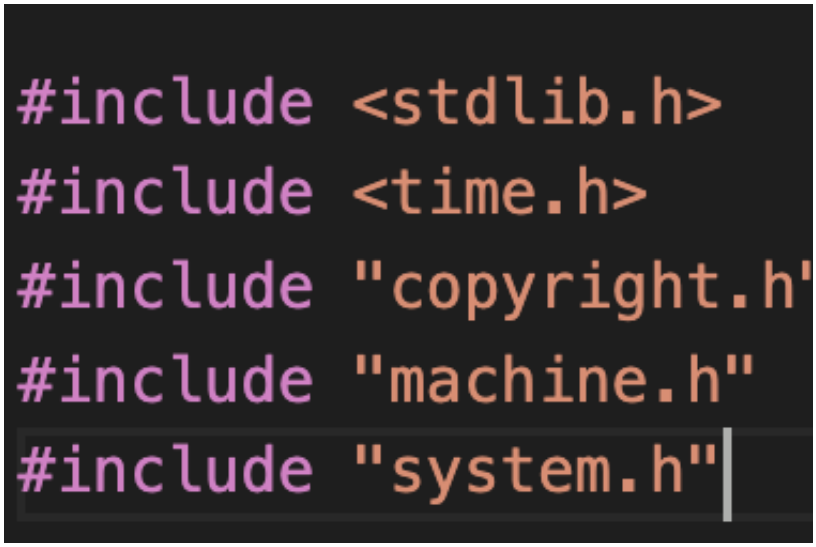
```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userprog$ ./nachos -x ../test/sort
Allocated 0
Thread: main    Read Page 0 form FILE to mainMemory
Allocated 1
Thread: main    Read Page 1 form FILE to mainMemory
Allocated 2
Thread: main    Read Page 2 form FILE to mainMemory
Allocated 3
Thread: main    Read Page 3 form FILE to mainMemory
Allocated 4
Thread: main    Read Page 4 form FILE to mainMemory
Allocated 5
Thread: main    Read Page 5 form FILE to mainMemory
Allocated 6
Thread: main    Read Page 6 form FILE to mainMemory
Allocated 7
Thread: main    Read Page 7 form FILE to mainMemory
Allocated 8
Thread: Thread2 Read Page 8 form FILE to mainMemory
Allocated 9
Thread: Thread2 Read Page 9 form FILE to mainMemory
Allocated 10
Thread: Thread2 Read Page 10 form FILE to mainMemory
Allocated 11
Thread: Thread2 Read Page 11 form FILE to mainMemory
Allocated 12
Thread: Thread2 Read Page 12 form FILE to mainMemory
Allocated 13
Thread: Thread2 Read Page 13 form FILE to mainMemory
Allocated 14
Thread: Thread2 Read Page 14 form FILE to mainMemory
Allocated 15
Thread: Thread2 Read Page 15 form FILE to mainMemory
Exiting userprog of thread: Thread2
Thread: Thread2 tlbtimes: 4743  tlbhits: 4448   accuracy: 0.937803
Exiting userprog of thread: main
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 12549, idle 0, system 30, user 12519
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
Thread: main    tlbtimes: 10862 tlbhits: 10433  accuracy: 0.960505
```

内容三 遇到的困难及解决方法

- translate.h中声明一个线程的指针时遇到头文件互相引用的问题，解决方法：前置声明

```
class Thread;
class TranslationEntry{
  public:
  ......
      Thread* BelongToThread;
  ......
}
```

- 如果把stdlib.h和time.h放到最后include就会报错，为什么



- 莫名错误，最后是tlb中raise缺页放在了tlb替换之前，这里是正确顺序。

内容四 收获及感想

熟悉了虚拟内存！和同学讨论流程并写了下来，非常有助于写代码。

内容五 对课程的意见或建议

无。