

虚拟内存实习报告

姓名： 学号：

日期： 2020/11/28

目录

| | |
|--------------------------|----|
| 内容一：任务完成情况..... | 3 |
| 任务完成列表（Y/N） | 3 |
| 具体 Exercise 完成情况 | 3 |
| Exercise1 | 3 |
| Exercise2 | 16 |
| Exercise3 | 20 |
| Exercise4 | 22 |
| Exercise5 | 23 |
| Exercise6 | 29 |
| Exercise7 | 34 |
| Challenge | 39 |
| 内容二：遇到的困难以及收获..... | 45 |
| 内容三：对课程或 Lab 的意见和建议..... | 46 |
| 内容四：参考文献..... | 46 |

内容一：任务完成情况

任务完成列表 (Y/N)

| | Exercise1 | Exercise2 | Exercise3 | Exercise4 | Exercise5 | Exercise6 | Exercise7 | Challenge |
|------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 第一部分 | Y | Y | Y | Y | Y | Y | Y | Y |

具体 Exercise 完成情况

Exercise1

这一部分要求阅读四份源代码，下面我将逐一分析各部分的具体内容。总的来说，这四份源代码包含了 Nachos 中用户程序与 TLB 机制相关的类和方法。另外，注意到建议中又推荐我们阅读其他源代码，因此我将在这里一并阅读并给出分析。

(一) **progtest.cc、addrspce.h、addrspce.cc**

progtest.cc 中主要包含了用户程序的创建函数和控制台测试函数。用户级程序需要有虚拟地址空间的支持，所以在这里把附录中推荐的 addrspce.h 和 addrspce.cc 一并阅读分析。

一、**progtest.cc**

这份代码的核心部分为 StartProcess 和 ConcoleTest 两个函数，它们分别用于启动用户及程序和测试控制台。

1.StartProcess

这个函数首先利用 Nachos 的文件系统打开参数中传入的文件名对应的文件，如果打开文件失败（返回空指针），那么输出“Unable to open file”的错误提示并返回，否则为这个程序分配一片虚拟地址空间 AddrSpace，然后为地址空间初始化寄存器、恢复机器的状态等信息后，开始运行这个用户级程序。ASSERT 是为了检测 Run 是否正常运行。如果 Run 正常运行，应该不会返回，程序终止时，会进行 exit 的系统调用。

```

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 program
    ASSERT(FALSE);                        // machine->Run never returns;
                                         // the address space exits
                                         // by doing the syscall "exit"
}

```

2.ConsoleTest

这个函数会创建一个控制台，解析输入的字符并响应（在这个控制台上输出字符），直到用户输入 q 字符后停止。为了进行响应和保护，这个函数需要一个全局的控制台变量和两个全局的信号量来支持。

```

void
ConsoleTest (char *in, char *out)
{
    char ch;

    console = new Console(in, out, ReadAvail, WriteDone, 0);
    readAvail = new Semaphore("read avail", 0);
    writeDone = new Semaphore("write done", 0);

    for (;;) {
        readAvail->P();           // wait for character to arrive
        ch = console->GetChar();
        console->PutChar(ch);    // echo it!
        writeDone->P();          // wait for write to finish
        if (ch == 'q') return; // if q, quit
    }
}

```

二、addrSpace.h 和 addrSpace.cc

在上面的分析中，我们可以看到，运行用户级程序需要有地址空间的支持，那么我们不妨来看一下 Nachos 中地址空间的实现。在 Nachos 中，地址空间对应于 AddrSpace 类。它拥有页数和页表两个私有成员，除此之外，还实现了一些公有方法：

```

class AddrSpace {
public:
    AddrSpace(OpenFile *executable);

    ~AddrSpace();

    void InitRegisters();

    void SaveState();
    void RestoreState();

private:
    TranslationEntry *pageTable;

    unsigned int numPages;
};

```

1. 构造函数和析构函数

构造函数会依次执行以下动作：

- (1) 函数首先会检查输入的待打开文件格式是否为 Nachos 支持的 noff 文件格式，如果不是，函数会调用 SwapHeader 函数对这个文件进行转码。SwapHeader 的主要任务是把文件的字节序转换为 Nachos 中的大端序，并重新确定代码段、数据段的大小。

```
static void
SwapHeader (NoffHeader *noffH)
{
    noffH->noffMagic = WordToHost(noffH->noffMagic);
    noffH->code.size = WordToHost(noffH->code.size);
    noffH->code.virtualAddr = WordToHost(noffH->code.virtualAddr);
    noffH->code.inFileAddr = WordToHost(noffH->code.inFileAddr);
    noffH->initData.size = WordToHost(noffH->initData.size);
    noffH->initData.virtualAddr = WordToHost(noffH->initData.virtualAddr);
    noffH->initData.inFileAddr = WordToHost(noffH->initData.inFileAddr);
    noffH->uninitData.size = WordToHost(noffH->uninitData.size);
    noffH->uninitData.virtualAddr = WordToHost(noffH->uninitData.virtualAddr);
    noffH->uninitData.inFileAddr = WordToHost(noffH->uninitData.inFileAddr);
}
```

转码完成后，函数会计算文件的代码和数据段的大小，并为它们分配页表，确认页个数，记录尺寸。从下面的实现中可以看到，**目前 Nachos 分配的地址空间大小为恰好能容纳代码和数据（初始化的和非初始化的）的大小，并向上取整至最小的整数页，而且有最大的物理页数限制**。正如说明中所展示的那样，目前的这一实现并不支持地址空间大小任意的动态变化，而且地址空间太大了还会报错，我们可能需要在后面的实现中改进这个问题。

```
NoffHeader noffH;
unsigned int i, size;

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);
size = numPages * PageSize;

ASSERT(numPages <= NumPhysPages);           // check we're not trying
                                              // to run anything too big --
                                              // at least until we have
                                              // virtual memory

DEBUG('a', "Initializing address space, num pages %d, size %d\n",
      numPages, size);
```

- (2) 接下来，构造函数生成物理地址到虚拟地址的映射。pageTable 是一个 TranslationEntry 类的数组。初始化时，数组中的每一个元素的虚拟地址等于物理地址，有效标记设置为 True，使用标记、修改标记、只读标记设置为 False，并把地址空间清零。清零主要是为了把未初始化的数据都置为 0，且初始化栈部分。

```

pageTable = new TranslationEntry[numPages];
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
}
// zero out the entire address space, to zero the uninitialized data segment
// and the stack segment
bzero(machine->mainMemory, size);

```

(3) 初始化地址空间之后，构造函数会把程序的代码段和数据段放入页表的对应位置
(代码段和数据段)

```

if (noffH.code.size > 0) {
    DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
          | noffH.code.virtualAddr, noffH.code.size);
    executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
                         noffH.code.size, noffH.code.inFileAddr);
}
if (noffH.initData.size > 0) {
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
          | noffH.initData.virtualAddr, noffH.initData.size);
    executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
                         noffH.initData.size, noffH.initData.inFileAddr);
}

```

析构函数执行的内容比较简单，就是把整个页表 delete 掉，释放这部分内存空间。

```

AddrSpace::~AddrSpace()
{
    delete pageTable;
}

```

2.InitRegisters

这个函数为用户寄存器设置初始值。实际上，这个函数会直接在 machine 的全局变量上写寄存器的值。首先把所有寄存器清空，然后把 PC 寄存器和 NexePC 的寄存器赋值为前两条指令的地址，再把栈寄存器的值设置为地址空间的底部，从底（高地址）向下（低地址）分配栈空间。Machine 的结构将在稍后分析。

```

void
AddrSpace::InitRegisters()
{
    int i;

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

    // Initial program counter -- 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('a', "Initializing stack register to %d\n", numPages * PageSize - 16);
}

```

3.SaveState

在上下文切换时，我们需要保存机器的状态，尤其是地址空间。但现在 Nachos 并没有实现对状态的存储，函数体为空。

```
void AddrSpace::SaveState()  
{}
```

4.RestoreState

这个函数会恢复状态信息。目前进行了页表和页数的恢复。

```
void AddrSpace::RestoreState()  
{  
    machine->pageTable = pageTable;  
    machine->pageTableSize = numPages;  
}
```

(二) machine.h、machine.cc、exception.cc

这两份代码包含了仿真用户级程序执行时所需的数据结构和辅助函数，以及各种宏变量。用户级程序和内核会被加载进 Nachos 的“主存”，但内核一般会与用户级程序分开存放。每次执行一条用户级程序中的指令。辅助内容包括：一些宏和枚举定义、指令数据结构 Instruction、机器仿真结构 Machine，以及一些辅助函数。下面让我来对它们逐一分析。

一、宏和枚举定义

machine.h 中的宏和枚举定义可以大致分为以下三类：

1.与内存空间大小相关的宏

这部分的宏包括：

每个页的大小，与磁盘分区大小相同，为 128 字节

最大物理页数：32

最大内存大小：页大小*最大物理页数（32*128 字节）

TLB 大小：4 个表项，Nachos 希望 TLB 小一些。

```
#define PageSize      SectorSize      // set the page size equal to  
                           // the disk sector size, for  
                           // simplicity  
  
#define NumPhysPages 32  
#define MemorySize   (NumPhysPages * PageSize)  
#define TLBSize       4                // if there is a TLB, make it small
```

2.CPU 状态的枚举定义

CPU 状态可以分为：无异常、系统调用异常（执行系统调用）、页错误异常（pageTable 没有有效项）、只读异常（写一个只读页）、总线异常（产生无效物理地址）、地址异常（进行了非对齐的地址引用）、溢出异常（整型溢出）、非法指令异常

(执行了未实现或者保留指令)。

```
enum ExceptionType { NoException,
    SyscallException,
    PageFaultException,
    ReadOnlyException,
    BusErrorException,
    AddressErrorException,
    OverflowException,
    IllegalInstrException,
    NumExceptionTypes
};
```

3.寄存器宏

Nachos 实现了 MIPS 中的 32 个通用目的寄存器，外加上 8 个为了支持用户级程序运行而添加的寄存器。这些寄存器在宏中定义如下：

32-33: 储存乘法结果

34: 当前 PC

35: 下一条指令的 PC (用于分支延迟)

36: 前一条指令的 PC (用于 debug)

37: 延迟加载的目标寄存器

38: 延迟加载中要加载的值存放在这里

39: 出现异常时失效的虚拟内存地址存放在这里

```
#define StackReg      29      // User's stack pointer
#define RetAddrReg     31      // Holds return address for procedure calls
#define NumGPRegs      32      // 32 general purpose registers on MIPS
#define HiReg          32      // Double register to hold multiply result
#define LoReg          33
#define PCReg          34      // Current program counter
#define NextPCReg      35      // Next program counter (for branch delay)
#define PrevPCReg      36      // Previous program counter (for debugging)
#define LoadReg         37      // The register target of a delayed load.
#define LoadValueReg   38      // The value to be loaded by a delayed load.
#define BadVAddrReg    39      // The failing virtual address on an exception
```

MIPS 的 32 个通用寄存器定义如下：

Table 5.3. MIPS Registers

| Register Number | Conventional Name | Usage |
|-----------------|-------------------|--|
| \$0 | \$zero | Hard-wired to 0 |
| \$1 | \$at | Reserved for pseudo-instructions |
| \$2 - \$3 | \$v0, \$v1 | Return values from functions |
| \$4 - \$7 | \$a0 - \$a3 | Arguments to functions - not preserved by subprograms |
| \$8 - \$15 | \$t0 - \$t7 | Temporary data, not preserved by subprograms |
| \$16 - \$23 | \$s0 - \$s7 | Saved registers, preserved by subprograms |
| \$24 - \$25 | \$t8 - \$t9 | More temporary registers, not preserved by subprograms |
| \$26 - \$27 | \$k0 - \$k1 | Reserved for kernel. Do not use. |
| \$28 | \$gp | Global Area Pointer (base of global data segment) |
| \$29 | \$sp | Stack Pointer |
| \$30 | \$fp | Frame Pointer |
| \$31 | \$ra | Return Address |
| \$f0 - \$f3 | - | Floating point return values |
| \$f4 - \$f10 | - | Temporary registers, not preserved by subprograms |
| \$f12 - \$f14 | - | First two arguments to subprograms, not preserved by subprograms |
| \$f16 - \$f18 | - | More temporary registers, not preserved by subprograms |
| \$f20 - \$f31 | - | Saved registers, preserved by subprograms |

二、Instruction 类

这个类用于定义一条指令。里面既存放了未解码的二进制指令，又存放了解码后的二进制指令，包括：指令的 op 码（展示要做什么）、指令要操作的寄存器、符号扩展的立即数。除此之外，类中还封装了一个 Decode 方法，用于进行解码工作。

```
class Instruction {
public:
    void Decode();      // decode the binary representation of the instruction
    unsigned int value; // binary representation of the instruction

    char opCode;        // Type of instruction. This is NOT the same as the
                        // opcode field from the instruction: see defs in mips.h
    char rs, rt, rd;   // Three registers from instruction.
    int extra;          // Immediate or target or shamt field or offset.
                        // Immediates are sign-extended.
};
```

Decode 方法的执行流程如下 (in mipssim.cc)：首先解析使用的寄存器 rs、rt、rd (他们在指令中的位置是固定的)，然后到 opTable 中查找 op 值 (OP 段在指令中也是固定的)，再根据 op 格式进一步确定立即数情况并进一步细化 opCode 的值。

```
void
Instruction::Decode()
{
    OpInfo *opPtr;

    rs = (value >> 21) & 0x1f;
    rt = (value >> 16) & 0x1f;
    rd = (value >> 11) & 0x1f;
    opPtr = &opTable[(value >> 26) & 0x3f];
    opCode = opPtr->opCode;
    if (opPtr->format == IFMT) {
        extra = value & 0xffff;
        if (extra & 0x8000) {
            extra |= 0xffff0000;
        }
    } else if (opPtr->format == RFMT) {
        extra = (value >> 6) & 0x1f;
    } else {
        extra = value & 0xffffffff;
    }
    if (opCode == SPECIAL) {
        opCode = specialTable[value & 0x3f];
    } else if (opCode == BCOND) {
        int i = value & 0x1f0000;

        if (i == 0) {
            opCode = OP_BLTZ;
        } else if (i == 0x10000) {
            opCode = OP_BGEZ;
        } else if (i == 0x100000) {
            opCode = OP_BLTZAL;
        } else if (i == 0x110000) {
            opCode = OP_BGEZAL;
        } else {
            opCode = OP_UNIMP;
        }
    }
}
```

三、Machine 类

这个类相当于定义了一个仿真的主机平台，除了系统调用（这里只实现了 10 种系统调用，而 UNIX 中有 200 种）和浮点指令以外，我们不应该让用户级程序直到它们是运行在虚拟主机上还是真实的硬件上。这个类中有两个私有成员，singleStep 设置为真时，每执行一条指令就会返回一次 debugger；runUtilTime 则表示返回 debugger 的间隔运行长。

```
private:  
    bool singleStep;           // drop back into the debugger after each  
                               // simulated instruction  
    int runUntilTime;          // drop back into the debugger when simulated  
                               // time reaches this value|
```

除此之外，Machine 还有一些可以让内核访问的公共成员，包括：储存用户级程序的主存指针、寄存器数组、TLB 快表、用户级进程的页表和页表大小。用户级进程的虚拟地址翻译由页表或者 TLB 快表控制。

```
char *mainMemory;           // physical memory to store user program,  
                           // code and data, while executing  
int registers[NumTotalRegs]; // CPU registers, for executing user programs  
  
TranslationEntry *tlb;       // this pointer should be considered  
                           // "read-only" to Nachos kernel code  
  
TranslationEntry *pageTable;  
unsigned int pageTableSize;
```

Machine 中剩下的内容便是各种公共方法了。这里我们首先分析 machine.cc 中实现的一些方法。对于其他方法，OneInstruction、DelayedLoad、Run 在 mipssim.cc 中实现，与 MIPS 体系结构有关；Translate、ReadMem、WriteMem 在 translate.cc 中实现，稍后分析。值得注意的是，这些函数虽然都是 public 的，但实际上只有 Run 和读写寄存器的函数允许被内核调用，其他函数 public 化是为了让 Machine 在 simulation 时可以调用。

1. 构造函数和析构函数

构造函数首先会把寄存器清空，并把主存组织成一个字节数组（char 数组），如果选择使用 TLB，回味它分配 TLBSize 个（4）表项，初始有效标记全部设置位 FALSE（valid=FALSE），并且页表为空；否则只使用页表来寻址。构造函数中的参数代表是否开启单步调试模式。在完成上述初始化后，还会调用 CheckEndian 函数检查真实的大小端情况与定义的大小端情况是否一致。

```

Machine::Machine(bool debug)
{
    int i;

    for (i = 0; i < NumTotalRegs; i++)
        registers[i] = 0;
    mainMemory = new char[MemorySize];
    for (i = 0; i < MemorySize; i++)
        mainMemory[i] = 0;
#ifndef USE_TLB
    tlb = new TranslationEntry[TLBSize];
    for (i = 0; i < TLBSize; i++)
        tlb[i].valid = FALSE;
    pageTable = NULL;
#else // use linear page table
    tlb = NULL;
    pageTable = NULL;
#endif
    singleStep = debug;
    CheckEndian();
}

static
void CheckEndian()
{
    union checkit {
        char charword[4];
        unsigned int intword;
    } check;

    check.charword[0] = 1;
    check.charword[1] = 2;
    check.charword[2] = 3;
    check.charword[3] = 4;

#ifdef HOST_IS_BIG_ENDIAN
    ASSERT (check.intword == 0x01020304);
#else
    ASSERT (check.intword == 0x04030201);
#endif
}

```

析构函数做的事情主要就是释放主存空间。此外，如果使用 TLB 寻址机制，那么还会释放 TLB 空间。

```

Machine::~Machine()
{
    delete [] mainMemory;
    if (tlb != NULL)
        delete [] tlb;
}

```

2.ReadRegister 与 WriteRegister

如果输入的寄存器标号在范围内，那么这两个函数会读或者写寄存器的值

```

int Machine::ReadRegister(int num)
{
    ASSERT((num >= 0) && (num < NumTotalRegs));
    return registers[num];
}

void Machine::WriteRegister(int num, int value)
{
    ASSERT((num >= 0) && (num < NumTotalRegs));
    // DEBUG('m', "WriteRegister %d, value %d\n", num, value);
    registers[num] = value;
}

```

3.RaiseException

在程序使用系统调用或者遇到指令异常时，这个函数会保存错误地址（如果有的话），陷入内核并触发异常处理，处理完毕后再返回用户态。

```

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

    // ASSERT(interrupt->getStatus() == UserMode);
    registers[BadAddrReg] = badVAddr;
    DelayedLoad(0, 0); // finish anything
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which); // interrupts are e
    interrupt->setStatus(UserMode);
}

```

4.Debugger

提供给用户级程序的一个调试器。我们不能直接使用 `gdb`，这需要实现更多系统调用。如果输入数字，会运行指定时间片长度后进行中断；如果输入 `c`，会运行至结束；如

果输入 return，会执行单条指令；如果输入？，则会显示提示信息。

```
void Machine::Debugger()
{
    char *buf = new char[80];
    int num;

    interrupt->DumpState();
    DumpState();
    printf("%d> ", stats->totalTicks);
    fflush(stdout);
    fgets(buf, 80, stdin);
    if (sscanf(buf, "%d", &num) == 1)
        runUntilTime = num;
    else {
        runUntilTime = 0;
        switch (*buf) {
        case '\n':
            break;

        case 'c':
            singleStep = FALSE;
            break;

        case '?':
            printf("Machine commands:\n");
            printf("    <return> execute one instruction\n");
            printf("    <number> run until the given timer tick\n");
            printf("    c      run until completion\n");
            printf("    ?      print help message\n");
            break;
        }
    }
    delete [] buf;
}
```

5.DumpState

输出 CPU 状态，主要是各个寄存器的值。对于 8 个 MIPS 之外的寄存器、栈指针、返回地址，输出时会标注名称。

```
void
Machine::DumpState()
{
    int i;

    printf("Machine registers:\n");
    for (i = 0; i < NumGPRegs; i++)
        switch (i) {
        case StackReg:
            printf("\tSP(%d):\t0x%x%s", i, registers[i],
                   ((i % 4) == 3) ? "\n" : "");
            break;

        case RetAddrReg:
            printf("\tRA(%d):\t0x%x%s", i, registers[i],
                   ((i % 4) == 3) ? "\n" : "");
            break;

        default:
            printf("\t%d:\t0x%x%s", i, registers[i],
                   ((i % 4) == 3) ? "\n" : "");
            break;
    }

    printf("\tHi:\t0x%x", registers[HiReg]);
    printf("\tLo:\t0x%x\n", registers[LoReg]);
    printf("\tPC:\t0x%x", registers[PCReg]);
    printf("\tNextPC:\t0x%x", registers[NextPCReg]);
    printf("\tPrevPC:\t0x%x\n", registers[PrevPCReg]);
    printf("\tLoad:\t0x%x", registers[LoadReg]);
    printf("\tLoadV:\t0x%x\n", registers[LoadValueReg]);
    printf("\n");
}
```

四、辅助函数

辅助函数包括异常处理和字节序转换两部分。

1. 异常处理：ExceptionHandler

这个函数在 exception.cc 中实现，在 Machine 类中的 RaiseException 函数中会被调用。

它是进入 Nachos 内核的入口点，当用户级程序进行系统调用，或者生成地址/代码异常时，会执行这个函数。对于系统调用，要把调用号放入 r2，最大支持 4 个参数，依序放在 r4-r7，返回值会被存在 r2 中。目前的异常处理中，只有系统调用会使用 interrupt 的 Halt 函数，根据系统调用号进行处理，其他异常都认为是不期望发生的，会直接终止程序。

```
void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if ((which == SyscallException) && (type == SC_Halt)) {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

2. 字节序转换函数

Nachos 实现了 4 个字节序转换函数，它们可以把 unsigned int 和 unsigned short 进行字节序的转换，它们在 translate.cc 中实现，将在下一部分分析。

(三) translate.h、translate.cc

translate.h 中存放了虚拟页到物理页翻译时使用的数据结构——TranslationEntry。它包含了虚拟页和物理页的页号，还包含了表示有效性、只读性、是否被使用、是否被修改的四个位。这里，valid 标识条目初始化情况；use 当页被引用或者修改时被设置；dirty 当页被修改时设置；readOnly 被设置则表示不允许修改这个页。

```
class TranslationEntry {
public:
    int virtualPage;
    int physicalPage;
    bool valid;
    bool readOnly;
    bool use;
    bool dirty;
};
```

translate.cc 存放着 Machine 类中与地址翻译有关的方法的实现，以及数据大小端转换函数的实现。

一、大小端转换

由于环境机器（Linux）是小端的，而 Nachos 是大端的，所以我们需要进行字节顺序的转换。这里 HOST 和 Machine 都认为是大端序，所以 xxxMachine 的函数就是对 xxxHost 的封装。在实现上，为了加快运行速度，为 result 变量加入 register 关键字，以说明会频繁使用这个变量。

```

unsigned int
WordToHost(unsigned int word) {
#ifndef HOST_IS_BIG_ENDIAN
    register unsigned long result;
    result = (word >> 24) & 0x000000ff;
    result |= (word >> 8) & 0x0000ff00;
    result |= (word << 8) & 0x00ff0000;
    result |= (word << 24) & 0xff000000;
    return result;
#else
    return word;
#endif /* HOST_IS_BIG_ENDIAN */
}

unsigned short
ShortToHost(unsigned short shortword) {
#ifndef HOST_IS_BIG_ENDIAN
    register unsigned short result;
    result = (shortword << 8) & 0xff00;
    result |= (shortword >> 8) & 0x00ff;
    return result;
#else
    return shortword;
#endif /* HOST_IS_BIG_ENDIAN */
}

unsigned int
WordToMachine(unsigned int word) { return WordToHost(word); }

unsigned short
ShortToMachine(unsigned short shortword) { return ShortToHost(shortword); }

```

二、Machine 类中关于地址翻译的函数

1.Translate

这个函数使用页表或者 TLB 机制，把虚拟地址翻译为物理地址。这个过程包含以下几个步骤：

(1) 首先检查地址对齐和 TLB、页表指针的空缺情况。如果 4 字节对齐，我们不允许地址模 4 余 3；2 字节对齐则不允许地址为奇数。否则会返回 AddressErrorException。此外，TLB 指针和页表指针不可以同时为空，也不可以同时不为空，我们只允许一种寻址机制。

```

if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){
    DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);
    return AddressErrorException;
}

// we must have either a TLB or a page table, but not both!
ASSERT(tlb == NULL || pageTable == NULL);
ASSERT(tlb != NULL || pageTable != NULL);

```

(2) 进行完检查以后，下一步会计算虚拟地址的 VPN 和偏移量。VPN 除以页大小就可以得到，偏移取模就可以得到。获取这两个量以后，对于 TLB 寻址机制，首先找到有效的 TLB 表项中 VPN 相同的那个条目，如果找不到，返回 PageFaultException；对于页表机制，如果 VPN 大于页表尺寸，返回 AddressErrorException，否则，如果所在表项无效，那么返回 PageFaultException。

```

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

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
    }
}

```

(3) 找到条目以后，如果对只读页面做些操作，返回 ReadOnlyException，否则根据条目中的物理页确定 pageFrame。确定好 pageFrame 后，会检查物理页号是否大于最大物理页数量，如果大于，返回 BusErrorException。如果上述异常都没有触发，那么函数会把这个条目中的 use 选项设置为 TRUE。如果是写操作，把 dirty 也设置为 True。最后，再根据物理页基址和偏移量设置指针，如果物理地址正常，那么返回 NoException，同时传入的地址指针被设置为地址值。

```

// if the pageFrame is too big, there is something really wrong!
// An invalid translation was loaded into the page table or TLB.
if (pageFrame >= NumPhysPages) {
    DEBUG('a', "*** frame %d > %d!\n", pageFrame, NumPhysPages);
    return BusErrorException;
}
entry->use = TRUE; // set the use, dirty bits
if (writing)
    entry->dirty = TRUE;
*physAddr = pageFrame * PageSize + offset;
ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
DEBUG('a', "phys addr = 0x%x\n", *physAddr);
return NoException;

```

2.ReadMem 和 WriteMem

这个函数用于读取内存。首先获取异常和地址情况。如果没有异常，那么根据读的大小来获取数值。目前只支持 1、2、4 字节的内存读操作。如果出现异常，调用异常处理程序。

而 WriteMem 则是写内存地址。执行流程与 ReadMem 类似，只是把 Translate 的 writing 参数设置为 True。这个函数也是只支持 1、2、4 字节的写，遇到异常后调用异常处理程序。

```

bool
Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Reading VA 0x%x, size %d\n", addr, size);

    exception = Translate(addr, &physicalAddress, size, FALSE);
    if (exception != NoException) {
        machine->RaiseException(exception, addr);
        return FALSE;
    }
    switch (size) {
        case 1:
            data = machine->mainMemory[physicalAddress];
            *value = data;
            break;

        case 2:
            data = *(unsigned short *) &machine->mainMemory[physicalAddress];
            *value = ShortToHost(data);
            break;

        case 4:
            data = *(unsigned int *) &machine->mainMemory[physicalAddress];
            *value = WordToHost(data);
            break;

        default: ASSERT(FALSE);
    }

    DEBUG('a', "\tvalue read = %8.8x\n", *value);
    return (TRUE);
}

bool
Machine::WriteMem(int addr, int size, int value)
{
    ExceptionType exception;
    int physicalAddress;

    DEBUG('a', "Writing VA 0x%x, size %d, value 0x%x\n", addr, size, value);

    exception = Translate(addr, &physicalAddress, size, TRUE);
    if (exception != NoException) {
        machine->RaiseException(exception, addr);
        return FALSE;
    }
    switch (size) {
        case 1:
            machine->mainMemory[physicalAddress] = (unsigned char) (value & 0xff);
            break;

        case 2:
            (*(unsigned short *) &machine->mainMemory[physicalAddress])
                = ShortToMachine((unsigned short) (value & 0xffff));
            break;

        case 4:
            (*(unsigned int *) &machine->mainMemory[physicalAddress])
                = WordToMachine((unsigned int) value);
            break;

        default: ASSERT(FALSE);
    }

    return TRUE;
}

```

Exercise2

这一部分要求我们处理 TLB Miss 抛出的异常——PageFaultException。在先前的分析中，我们可以看到，除了 NoException 的状态全部会调用 Halt 终止，因此我们要为这个 Exception 添加一个判断的分支。除此之外，这个异常也有可能是由于页表缺失引起的，所以我们还需要在内部再做一层判断。现在的实现中，只要 TLB 全部 miss 就会抛出异常，这个时候为了处理异常，我们应该尝试填补 TLB 表项，这需要我们去页表中再进一步匹配，并更新 TLB。

注意到，如果出现异常，ReadMem 和 WriteMem 会把虚拟地址作为 RaiseException 的参数，RaiseException 会把它放到 BadVAddrReg 寄存器中，因此我们可以在异常处理函数中读取这个寄存器来获得地址。

根据上面的分析，我们可以分两步来执行：

首先，修改 ExceptionHandler 函数，对 PageFaultException 进行特判。这里我们还需要区分是否开启了 TLB 机制：

```
if (which == PageFaultException)
{
    if(machine->tlb == NULL) // PageTable error
    {
        DEBUG('m', "====> Page Table Fault.\n");
        ASSERT(FALSE);
    }
    else // TLB miss
    {
        DEBUG('m', "====> TLB miss arises!\n");
        int badVAddr = machine->ReadRegister(BadVAddrReg);
        TLBMissHandler(badVAddr);
    }
    return;
}
```

然后，实现 TLB Miss 的处理函数。大致流程是：根据地址计算 VPN 和 Offset，利用 VPN 去页表寻找对应条目，并加载进 TLB 中。这里其实涉及到了 TLB 替换算法的设计，将在下一个 Exercise 中进一步完成。除此之外，再观察执行单条指令的 OneInstruction 函数，可以发现，如果取址异常，机器不会更新 PC。也就是说，这条指令会在中断处理完毕后重新执行一次。因此，我们只要保证对应条目被加载进 TLB 表即可。为了简便起见，这里采用轮转的方法来选择被替换的 TLB 表项。实现如下：

```
int Index = 0;

void
TLBMissHandler(int addr)
{
    DEBUG('m', "====> Handle TLB miss caused by VAddr %d!\n", addr);

    unsigned int vpn = (unsigned)addr / PageSize;
    //unsigned int offset = (unsigned)addr % PageSize; // Maybe we'll not use this.

    machine->tlb[Index] = machine->pageTable[vpn];
    Index = (Index + 1) % TLBSize;
}
```

下一步便是测试实现。注意到 test 文件夹中已经实现了一些基础的测试函数。这里选择 halt 的可执行文件来进行测试。但直接测试会发现并没有触发 TLB 机制，由于 TLB 机制需要条件编译，所以可以推断是编译时没有加上 USE_TLB 的宏选项。在查阅资料和检查 Makefile 文件后，可以发现 userprog 中的 Makefile 的 DEFINE 项中确实缺少这个宏，所以需要添加上它。

```

DEFINES = -DUSER_PROGRAM -DFILESYS_NEEDED -DFILESYS_STUB -DUSE_TLB
INCPATH = -I..../bin -I..../filesys -I..../userprog -I..../threads -I..../machine
HFILES = $(THREAD_H) $(USERPROG_H)
CFILES = $(THREAD_C) $(USERPROG_C)
C_OFILES = $(THREAD_O) $(USERPROG_O)

```

此外，由于目前的机制不允许 TLB 与页表同时出现，我们还需要在 translate 中注释掉两条 ASSERT 语句。

```

// we must have either a TLB or a page table, but not both!
//ASSERT(tlb == NULL || pageTable == NULL);
ASSERT(tlb != NULL || pageTable != NULL);

```

这时，运行 halt 的指令序列如下：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d am -x ..//test/halt
Initializing address space, num pages 10, size 1280
Initializing code segment, at 0x0, size 256
Initializing stack register to 1264
Starting thread "main" at time 10
Reading VA 0x0, size 4
    Translate 0x0, read: *** no valid TLB entry found for this virtual page!
Exception: page fault/no TLB entry
====> TLB miss arises!
====> Handle TLB miss caused by VAddr 0!
Reading VA 0x0, size 4
    Translate 0x0, read: phys addr = 0x0
    value read = 0c000034
At PC = 0x0: JAL 52
Reading VA 0x4, size 4
    Translate 0x4, read: phys addr = 0x4
    value read = 00000000
At PC = 0x4: SLL r0,r0,0
Reading VA 0xd0, size 4
    Translate 0xd0, read: *** no valid TLB entry found for this virtual page!
Exception: page fault/no TLB entry
====> TLB miss arises!
====> Handle TLB miss caused by VAddr 208!
Reading VA 0xd0, size 4
    Translate 0xd0, read: phys addr = 0xd0
    value read = 27bdfffe
At PC = 0xd0: ADDIU r29,r29,-24
Reading VA 0xd4, size 4
    Translate 0xd4, read: phys addr = 0xd4
    value read = afbf0014
At PC = 0xd4: SW r31,20(r29)
Writing VA 0x4ec, size 4, value 0x8
    Translate 0x4ec, write: *** no valid TLB entry found for this virtual page!
Exception: page fault/no TLB entry
====> TLB miss arises!
====> Handle TLB miss caused by VAddr 1260!
Reading VA 0xd4, size 4
    Translate 0xd4, read: phys addr = 0xd4
    value read = afbf0014
At PC = 0xd4: SW r31,20(r29)
Writing VA 0x4ec, size 4, value 0x8
    Translate 0x4ec, write: phys addr = 0x4ec
Reading VA 0xd8, size 4
    Translate 0xd8, read: phys addr = 0xd8
    value read = afbe0010
At PC = 0xd8: SW r30,16(r29)
Writing VA 0xe48, size 4, value 0x0
    Translate 0xe48, write: phys addr = 0xe48
Reading VA 0xdc, size 4
    Translate 0xdc, read: phys addr = 0xdc
    value read = 0c000030
At PC = 0xdc: JAL 48
Reading VA 0xe0, size 4
    Translate 0xe0, read: phys addr = 0xe0
    value read = 03a0f021
At PC = 0xe0: ADDU r30,r29,r0
Reading VA 0xc0, size 4
    Translate 0xc0, read: phys addr = 0xc0
    value read = 03e00008
At PC = 0xc0: JR r0,r31
Reading VA 0xc4, size 4
    Translate 0xc4, read: phys addr = 0xc4
    value read = 00000000
At PC = 0xc4: SLL r0,r0,0
Reading VA 0xe4, size 4
    Translate 0xe4, read: phys addr = 0xe4
    value read = 0c000004
At PC = 0xe4: JAL 4
Reading VA 0xe8, size 4
    Translate 0xe8, read: phys addr = 0xe8
    value read = 00000000
At PC = 0xe8: SLL r0,r0,0
Reading VA 0x10, size 4
    Translate 0x10, read: phys addr = 0x10
    value read = 24020000
At PC = 0x10: ADDIU r2,r0,0
Reading VA 0x14, size 4
    Translate 0x14, read: phys addr = 0x14
    value read = 0000000c
At PC = 0x14: SYSCALL
Exception: syscall
Shutdown, initiated by user program.
Machine halting!

```

如果我们只使用默认的页表机制来寻址，指令序列如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d am -x ..//test/halt
Initializing address space, num pages 10, size 1280
Initializing code segment, at 0x0, size 256
Initializing stack register to 1264
Starting thread "main" at time 10
Reading VA 0x0, size 4
    Translate 0x0, read: phys addr = 0x0
    value read = 0c000034
At PC = 0x0: JAL 52
Reading VA 0x4, size 4
    Translate 0x4, read: phys addr = 0x4
    value read = 00000000
At PC = 0x4: SLL r0,r0,0
Reading VA 0xd0, size 4
    Translate 0xd0, read: phys addr = 0xd0
    value read = 27bdffe8
At PC = 0xd0: ADDIU r29,r29,-24
Reading VA 0xd4, size 4
    Translate 0xd4, read: phys addr = 0xd4
    value read = afbf0014
At PC = 0xd4: SW r31,20(r29)
Writing VA 0x4ec, size 4, value 0x8
    Translate 0x4ec, write: phys addr = 0x4ec
Reading VA 0xd8, size 4
    Translate 0xd8, read: phys addr = 0xd8
    value read = afbe0010
At PC = 0xd8: SW r30,16(r29)
Writing VA 0x4e8, size 4, value 0x0
    Translate 0x4e8, write: phys addr = 0x4e8
Reading VA 0xdc, size 4
    Translate 0xdc, read: phys addr = 0xdc
    value read = 0c000030
At PC = 0xdc: JAL 48
Reading VA 0xe0, size 4
    Translate 0xe0, read: phys addr = 0xe0
    value read = 03a0f021
At PC = 0xe0: ADDU r30,r29,r0
Reading VA 0xc0, size 4
    Translate 0xc0, read: phys addr = 0xc0
    value read = 03e00008
At PC = 0xc0: JR r0,r31
Reading VA 0xc4, size 4
    Translate 0xc4, read: phys addr = 0xc4
    value read = 00000000
At PC = 0xc4: SLL r0,r0,0
Reading VA 0xe4, size 4
    Translate 0xe4, read: phys addr = 0xe4
    value read = 0c000004
At PC = 0xe4: JAL 4
Reading VA 0xe8, size 4
    Translate 0xe8, read: phys addr = 0xe8
    value read = 00000000
At PC = 0xe8: SLL r0,r0,0
Reading VA 0x10, size 4
    Translate 0x10, read: phys addr = 0x10
    value read = 24020000
At PC = 0x10: ADDIU r2,r0,0
Reading VA 0x14, size 4
    Translate 0x14, read: phys addr = 0x14
    value read = 0000000c
At PC = 0x14: SYSCALL
Exception: syscall
Shutdown, initiated by user program.
Machine halting!

Ticks: total 22, idle 0, system 10, user 12
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...
```

可以看到，两种情况指令执行序列相同（注意到 OneInstruction 的机制是如果发生了寻址错误就会重新执行一次当前指令），因此可以认为 TLB miss 的异常处理已实现完成。

注：这里只使用 **halt** 来测试，是因为目前的 Nachos 地址空间固定，其他代码片段过大，目前的地址空间装载不下。

Exercise3

这一部分要求我们实现至少两种 TLB 置换算法，并比较其优劣。

考察教材上的页面置换算法，我决定首先实现最朴素的 FIFO 算法。再考虑到 **Nachos** 的页表项中已经实现了 **use** 位（对应于现代操作系统上的 R 位），所以我实现的第二种算法为时钟置换算法。

(一) FIFO 算法的实现

这个算法中，如果出现 TLB miss，会把第一个表项换出去。我们只需要把 machine 中的 TLB 数组看作一个队列，模拟这一过程，于是 FIFO 算法实现如下：

```
#ifdef USE_FIFO
void
TLBFIFO(TranslationEntry page)
{
    DEBUG('m', "====> Now use FIFO algorithm to exchange TLB item.\n");
    int idx = -1;

    for(int i=0; i<TLBSize; ++i)
    {
        if(machine->tlb[i].valid == FALSE) // find a n unused item
        {
            idx = i;
            DEBUG('m', "====> Find a free item in TLB.\n");
            break;
        }
    }

    if(idx == -1) // all items are used
    {
        DEBUG('m', "====> We need to remove the first page in TLB.\n");
        for(int i=0; i<TLBSize-1; ++i)
        {
            machine->tlb[i] = machine->tlb[i+1];
        }
        idx = TLBSize - 1;
    }

    machine->tlb[idx] = page;
}
#endif
```

(二) CLOCK 算法的实现

这个算法中，应该对表项进行轮盘式查询。找到空位则停止，否则对已使用的表项设置为未使用，在 TLB 中保留一次使用的机会，因此实现如下：

```
#ifdef USE_CLOCK
void
TLBClock(TranslationEntry page)
{
    DEBUG('m', "====> Now use CLOCK algorithm to exchange TLB item.\n");
    while(true)
    {
        if(machine->tlb[Index].valid == FALSE) // there is a free item
        {
            DEBUG('m', "====> Find a free item in TLB.\n");
            break;
        }
        else // this item is occupied
        {
            if(machine->tlb[Index].use) // just used once
            {
                DEBUG('m', "====> Item %d can be given another chance.\n", Index);
                machine->tlb[Index].use = FALSE;
                Index = (Index +1) % TLBSize;
            }
            else // have given one chance
            {
                DEBUG('m', "====> Item %d have been given extra chance.\n", Index);
                break;
            }
        }
    }

    // initialize new entry
    machine->tlb[Index] = page;
    machine->tlb[Index].use = TRUE;
}
#endif
```

(三) 置换次数比较

为了比较置换次数和 miss rate，我们还需要引入计数变量 misscnt（计数缺失次数）和 totalcnt（总访存次数）。这两个变量在 translate.h 中声明，在 translate.cc 中初始化。当调用 translate.cc 时，totalcnt 增加；当 ReadMem 和 WriteMem 出现 PageFaultError 时，misscnt 增加。miss rate 使用 misscnt/totalcnt 计算即可。为了便于查看，在 ExceptionHandler 中，对于程序终止的处理中加入了输出 miss rate 的函数。注意在 Nachos 中，每次 TLB 项缺失都会重新执行一次访存，所以计算总数时应该减去 miss 数。

```
void
TLBMissRate()
{
#ifndef USE_TLB
    printf("Total translation count is %d, translation missing count is %d, miss rate is %.6f.\n",
           totalcnt-misscnt, misscnt, misscnt*1.0 / (totalcnt-misscnt));
#endif
}
```

除此之外，观察并测试目前给出的示例程序，只有 halt 能加载运行，其他的代码都太大了，为了更多地访存，我又编写了一个示例程序 arrayAdd，内容如下：

```
#include "syscall.h"

#define T 2
#define N 40

int
main()
{
    int t, i;
    int A[N], B[N];
    for(t=0; t<T; ++t)
    {
        A[0] = 1; B[0] = 1;
        A[1] = 2; B[1] = 2;
        for(i=2; i<N; ++i)
        {
            A[i] = A[i-1] + A[i-2];
            B[i] = B[i-1] + B[i-2];
        }
    }
    Halt();
}
```

观察 test 文件夹的 Makefile，仿照其他测试文件，为这个文件添加编译选项如下：

```
arrayAdd.o: arrayAdd.c
    $(CC) $(CFLAGS) -c arrayAdd.c
arrayAdd: arrayAdd.o start.o
    $(LD) $(LDFLAGS) start.o arrayAdd.o -o arrayAdd.coff
    ..../bin/coff2noff arrayAdd.coff arrayAdd
```

编译出可执行文件后，便可以进行两种算法的测试了。

对 FIFO 算法的测试结果如下（由于指令运行比较多，所以只展示 miss rate）：

```
Total translation count is 5991, translation missing count is 428, miss rate is 0.071440.
```

对 Clock 算法的测试结果如下：

```
Total translation count is 5886, translation missing count is 316, miss rate is 0.053687.
```

可以看到，在运行测试程序时，CLOCK 算法的表现比 FIFO 要好一些。我觉得可以认为是第二次机会的机制让循环带来的重复访问可以有更多的命中次数。

Exercise4

这一部分要求我们实现空闲链表或者位图等空闲内存管理结构。我选择实现位图的管理模式。

位图管理本质上是把内存划分为多个分配单元，再用一定长度的二进制位来标记每个单元的使用情况。注意到目前在 Nachos 中，内存本身被分成了 32 个物理页，因此为了简化起见，我们可以直接把内存按物理页大小来分成 32 个单元，这样我们可以直接用一个无符号整数来作为位图。

为了使用这样的机制，我们应该在 Machine 类中加入以下内容：

```
#ifdef USE_BITMAP
    unsigned bitmap;
    int allocateMem();
    void freeMem();
#endif
```

其中，无符号整数 bitmap 表示位图，在 Machine 的构造函数中初始化为 0。

allocateMem 和 freeMem 用于分配和释放内存，具体实现如下：

```
int
Machine::allocateMem()
{
    for(int pos=0; pos<NumPhysPages; ++pos)
    {
        if((bitmap & (1<<pos)) == 0)
        {
            bitmap |= (1<<pos);
            DEBUG('B', "Allocate frame %d, and bitmap is %08X\n", pos, bitmap);
            return pos;
        }
    }
    DEBUG('B', "All frames have been used!\n");
    return -1;
}
```

现在我们只是实现了位图的管理，为了使用位图，我们还应该在分配和释放内存的时候加入位图的管理机制。由于目前还是单线程的状态，因此，我们首先需要修改的是 AddrSpace 的构造函数：

```
void
Machine::freeMem()
{
    for(int i=0; i<pageTableSize; ++i)
    {
        if(pageTable[i].valid)
        {
            int pos = pageTable[i].physicalPage;
            bitmap &= (~(1<<pos));
            DEBUG('B', "Free frame %d, and bitmap is %08X\n", pos, bitmap);
        }
    }
    DEBUG('B', "After freeing, bitmap is %08X\n", bitmap);
}
```

除此之外，注意到执行程序退出时，执行的 **Cleanup** 函数中，最后会执行 **Exit** 函数退出运行，但对于 **Halt** 函数，是先处理 **Halt** 系统调用，再调用 **sysdep** 中的 **Exit** 函数直接退出。因此我们要在异常处理函数中，调用 **interrupt->Halt()** 之前释放空间。注意此时 **machine** 上保存的地址空间即为 **currentThread** 上的地址空间，因此可以这样实现：

```
#ifndef USER_PROGRAM
    if(currentThread->space != NULL)
    {
        #ifdef USE_BITMAP
            machine->freeMem();
        #endif
        delete currentThread->space;
        currentThread->space = NULL;
    }
#endif
```

运行测试程序，可查看位图管理结果如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d B -x ../test/halt
Allocate frame 0, and bitmap is 00000001
Allocate frame 1, and bitmap is 00000003
Allocate frame 2, and bitmap is 00000007
Allocate frame 3, and bitmap is 0000000F
Allocate frame 4, and bitmap is 0000001F
Allocate frame 5, and bitmap is 0000003F
Allocate frame 6, and bitmap is 0000007F
Allocate frame 7, and bitmap is 000000FF
Allocate frame 8, and bitmap is 000001FF
Allocate frame 9, and bitmap is 000003FF
Free frame 0, and bitmap is 000003FE
Free frame 1, and bitmap is 000003FC
Free frame 2, and bitmap is 000003F8
Free frame 3, and bitmap is 000003F0
Free frame 4, and bitmap is 000003E0
Free frame 5, and bitmap is 000003C0
Free frame 6, and bitmap is 00000380
Free frame 7, and bitmap is 00000300
Free frame 8, and bitmap is 00000200
Free frame 9, and bitmap is 00000000
After freeing, bitmap is 00000000
Machine halting!

Ticks: total 22, idle 0, system 10, user 12
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...
```

可以看到，由于前面对内存的分配是从低位到高位依次寻找空闲单元，所以内存会被连续分配；最后释放以后 **machine** 中的位图又重新清零，因此上述实现可以认为是正确的。

事实上，可以观察到，在 **userprog** 中，**Nachos** 也提供了 **bitmap** 位图类。如果后续实验中需要更复杂的管理机制，把单元进一步缩小，那么我们可能需要使用封装好的这个位图类来实现。

Exercise5

这一部分要求我们实现对多线程的支持。

要支持多线程，首先要完善对于单线程的支持。注意到，先前使用的测试函数都是在

执行完主体程序后调用 Halt 进行停机。但如果我们不以 Halt 结尾，我们会发现，在现有的修改下，程序运行结束后会触发 **ASSERT** 错误，进一步分析可知，实际上是触发了 Exit 系统调用异常。因此，我们应该先实现 Exit 异常的处理。注意到，为了实现可以运行多个线程，在 Exit 的处理中，我们不应该调用 Halt 停机，而是应该对当前线程调用 Finish，进行新一轮的线程调度。因此，Exit 的处理如下：

```

if (which == SyscallException) {
    if(type == SC_Halt)
    {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        TLBMissRate();
    #ifdef USER_PROGRAM
        if(currentThread->space != NULL)
        {
    #ifdef USE_BITMAP
            machine->freeMem();
    #endif
            delete currentThread->space;
            currentThread->space = NULL;
        }
    #endif
        interrupt->Halt();
    }
    else if(type == SC_Exit)
    {
        ExitHandler();
    }
}

void
ExitHandler()
{
    // deal with exit status
    int status = machine->ReadRegister(4); // get the first argument ----- exit status
    if(status == 0)
    {
        DEBUG('E', "*****User program exit normally*****\n");
    }
    else
    {
        DEBUG('E', "*****User program exit with status %d*****\n", status);
    }

    // release space
    #ifdef USER_PROGRAM
        if(currentThread->space != NULL)
        {
    #ifdef USE_BITMAP
            machine->freeMem();
    #endif
            currentThread->space->SaveState(); // clear tlb when current thread exits
            delete currentThread->space;
            currentThread->space = NULL;
        }
    #endif

    // finish this user thread
    currentThread->Finish();
}

```

除此之外，注意到，如果出现了运行中的线程切换，由于 TLB 是共享的，因此，如果我们开启了 TLB 机制，我们还需要刷新 TLB，因此需要修改 AddrSpace 的 SaveState 方法：

```

void AddrSpace::SaveState()
{
    #ifdef USE_TLB
        DEBUG('T', "Clean up TLB when context switch occurs!\n");
        for(int i=0; i<TLBSize; ++i)
        {
            machine->tlb[i].valid = FALSE;
        }
    #endif
}

```

进一步观察 AddrSpace 的构造方法，我们可以看到，**初始的实现中，总是在对内存中从 0 开始的地址进行初始化和赋值。因此，为了支持多线程，我们需要基于 Exercise4 中实现的位图，修改对线程地址空间的初始化和分配。**否则，对于后续的线程，由于物理地址并不是从 0 开始的，会出现指令的错误执行（典型的错误是 SLL r0 r0 0，这条指令的位表示全部为 0），修改如下：

```
#ifdef USE_BITMAP
    // clear main memory used by current program
    for(int i=0; i<numPages; ++i)
    {
        for(int j=0; j<PageSize; ++j)
        {
            machine->mainMemory[pageTable[i].physicalPage*PageSize+j] = 0;
        }
    }

    // allocate data and code sections, we need to translate sections' virtualAddr to physicalAddr
    // one byte per loop
    for(int VA=noffH.code.virtualAddr, cnt=0; VA<noffH.code.size; ++VA, ++cnt)
    {
        unsigned int VPN = VA / PageSize;
        unsigned int PPN = pageTable[VPN].physicalPage;
        unsigned int PPO = VA % PageSize;
        unsigned int PA = PPN*PageSize + PPO;
        //printf("%d 0x%llx %d 0x%llx\n", PA, PA, VA, VA);
        executable->ReadAt(&(machine->mainMemory[PA]),
                            1, noffH.code.inFileAddr+cnt);
        //printf("%d 0x%llx\n", machine->mainMemory[PA], machine->mainMemory[PA]);
    }

    for(int VA=noffH initData.virtualAddr, cnt=0; VA<noffH initData.size; ++VA, ++cnt)
    {
        unsigned int VPN = VA / PageSize;
        unsigned int PPN = pageTable[VPN].physicalPage;
        unsigned int PPO = VA % PageSize;
        unsigned int PA = PPN*PageSize + PPO;
        //printf("%d 0x%llx %d 0x%llx\n", PA, PA, VA, VA);
        executable->ReadAt(&(machine->mainMemory[PA]),
                            1, noffH initData.inFileAddr+cnt);
        //printf("%d 0x%llx\n", machine->mainMemory[PA], machine->mainMemory[PA]);
    }
}

#else
```

如果再去分析用户程序的加载和执行过程，我们可以发现，事实上，我们可以同时加载多个程序进入 Nachos 的“内存”，但是现在的用户级程序执行入口（progtest.cc 中的 StartProcess）只能运行一个线程，因为这个函数只加载一次可执行文件，而且 machine->Run() 是一个执行指令的死循环。所以，为了检验上面的支持多线程的实现，我们还需要编写线程启动函数。实现如下：

```
#define N 3
void
StartNProcesses(char *filename)
{
    // open executable file
    OpenFile *executable = fileSystem->Open(filename);

    if (executable == NULL) {
        printf("Unable to open file %s\n", filename);
        return;
    }

    // create N threads
    Thread *threads[N];
    for(int i=0; i<N; ++i)
    {
        threads[i] = CreateThread(executable, i+1);
    }

    // close file
    delete executable;

    // fork new threads
    for(int i=0; i<N; ++i)
    {
        threads[i]->Fork(UserThread, (void*)(i+1));
    }

    // main thread give up CPU for new User Programs
    currentThread->Yield();
}
```

其中，还用到了以下两个辅助函数（UserThread 中如果不用 strdup 会输出乱码）：

```
Thread*  
CreateThread(OpenFile* executable, int num)  
{  
    printf("Thread %d is being created\n", num);  
  
    // allocate structre of new thread  
    char ThreadName[20];  
    sprintf(ThreadName, "User Program %d", num);  
    Thread *thread = new Thread(strdup(ThreadName), 0);  
  
    // generate address space for new thread  
    AddrSpace *space = new AddrSpace(executable);  
    thread->space = space;  
  
    return thread;  
}  
  
void  
UserThread(int which)  
{  
    printf("User Program %d starts to run!\n", which);  
  
    // initialization  
    currentThread->space->InitRegisters();  
    currentThread->space->RestoreState();  
    currentThread->space->PrintAddrState();  
  
    // run program  
    machine->Run();  
  
    //should not reach  
    printf("error occurs\n");  
    ASSERT(FALSE);  
}
```

然后，我们需要在 thread 文件夹中的 main 文件中添加对上述测试代码的命令行支持：

```
| extern void StartNProcesses(char *filename);  
  
else if(!strcmp(*argv, "-X"))           // run user programs, multithreading  
{  
    ASSERT(argc > 1);  
    StartNProcesses(*(argv + 1));  
    argCount = 2;  
}
```

为了查看地址空间情况，我在 AddrSpace 中添加了一个打印线程地址空间的函数，用

于测试：

```
void  
AddrSpace::PrintAddrState()  
{  
    printf("===== PageTable Info =====\n");  
    printf("numPages = %d\n", numPages);  
    printf("VPN\TPN\tvalid\tused\tdirty\n");  
    for(int i=0; i<numPages; ++i)  
    {  
        printf("%d\t%d\t%d\t", pageTable[i].virtualPage, pageTable[i].physicalPage, pageTable[i].valid);  
        printf("%d\t%d\t%d\n", pageTable[i].readonly, pageTable[i].use, pageTable[i].dirty);  
    }  
    printf("===== ======\n");  
}
```

接下来，我们便可以开始撰写用户程序了。由于现在的内存空间有限，为避免内存不足，我的用户级程序书写如下：

```
#include "syscall.h"  
  
int  
main()  
{  
    Exit(0);  
    /* not reached */  
}
```

上述实现的测试结果如下（E 代表 ExitHandler 中的输出，t 为线程测试的输出，m 为指令访存输出）：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d Etm -X .../test/exit
Thread 1 is being created
Thread 2 is being created
Thread 3 is being created
Forking thread "User Program 1" with func = 0x5660a51c, arg = 1
Putting thread User Program 1 on ready list.
Forking thread "User Program 2" with func = 0x5660a51c, arg = 2
Putting thread User Program 2 on ready list.
Forking thread "User Program 3" with func = 0x5660a51c, arg = 3
Putting thread User Program 3 on ready list.
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "User Program 1"
User Program 1 starts to run!
===== PageTable Info =====
numpages = 10
VPN    PPN    valid   RD     Used   Dirty
0      0       1        0      0       0
1      1       1        0      0       0
2      2       1        0      0       0
3      3       1        0      0       0
4      4       1        0      0       0
5      5       1        0      0       0
6      6       1        0      0       0
7      7       1        0      0       0
8      8       1        0      0       0
9      9       1        0      0       0
=====
Starting thread "User Program 1" at time 50
At PC = 0x0: JAL 52
At PC = 0x4: SLL r0,r0,0
At PC = 0xd0: ADDIU r29,r29,-24
At PC = 0xd4: SW r31,20(r29)
At PC = 0xd8: SW r30,16(r29)
At PC = 0xdc: JAL 48
At PC = 0xe0: ADDU r30,r29,r0
At PC = 0xc0: JR r0,r31
At PC = 0xc4: SLL r0,r0,0
At PC = 0xe4: ADDU r29,r30,r0
At PC = 0xe8: LW r31,20(r29)
At PC = 0xec: LW r30,16(r29)
At PC = 0xf0: JR r0,r31
At PC = 0xf4: ADDIU r29,r29,24

At PC = 0x8: JAL 8
At PC = 0xc: ADDU r4,r0,r0
At PC = 0x20: ADDIU r2,r0,1
At PC = 0x24: SYSCALL
Exception: syscall
***User program exit normally***
Finishing thread "User Program 1"
Sleeping thread "User Program 1"
Switching from thread "User Program 1" to thread "User Program 2"
User Program 2 starts to run!
===== PageTable Info =====
numpages = 10
VPN    PPN    valid   RD     Used   Dirty
0      10      1        0      0       0
1      11      1        0      0       0
2      12      1        0      0       0
3      13      1        0      0       0
4      14      1        0      0       0
5      15      1        0      0       0
6      16      1        0      0       0
7      17      1        0      0       0
8      18      1        0      0       0
9      19      1        0      0       0
=====
Starting thread "User Program 2" at time 77
At PC = 0x0: JAL 52
At PC = 0x4: SLL r0,r0,0
At PC = 0xd0: ADDIU r29,r29,-24
At PC = 0xd4: SW r31,20(r29)
At PC = 0xd8: SW r30,16(r29)
At PC = 0xdc: JAL 48
At PC = 0xe0: ADDU r30,r29,r0
At PC = 0xc0: JR r0,r31
At PC = 0xc4: SLL r0,r0,0
At PC = 0xe4: ADDU r29,r30,r0
At PC = 0xe8: LW r31,20(r29)
At PC = 0xec: LW r30,16(r29)
At PC = 0xf0: JR r0,r31
At PC = 0xf4: ADDIU r29,r29,24
At PC = 0x8: JAL 8
At PC = 0xc: ADDU r4,r0,r0
At PC = 0x20: ADDIU r2,r0,1
At PC = 0x24: SYSCALL
```

```

Exception: syscall
***User program exit normally***
Finishing thread "User Program 2"
Sleeping thread "User Program 2"
Switching from thread "User Program 2" to thread "User Program 3"
User Program 3 starts to run!
===== PageTable Info =====
numpages = 10
VPN     PPN    valid   RD    Used   Dirty
0       20      1        0      0      0
1       21      1        0      0      0
2       22      1        0      0      0
3       23      1        0      0      0
4       24      1        0      0      0
5       25      1        0      0      0
6       26      1        0      0      0
7       27      1        0      0      0
8       28      1        0      0      0
9       29      1        0      0      0
=====
Starting thread "User Program 3" at time 104
At PC = 0x0: JAL 52
At PC = 0x4: SLL r0,r0,0
At PC = 0xd0: ADDIU r29,r29,-24
At PC = 0xd4: SW r31,20(r29)
At PC = 0xd8: SW r30,16(r29)
At PC = 0xdc: JAL 48
At PC = 0xe0: ADDU r30,r29,r0
At PC = 0xc0: JR r0,r31
At PC = 0xc4: SLL r0,r0,0
At PC = 0xe4: ADDU r29,r30,r0
At PC = 0xe8: LW r31,20(r29)
At PC = 0xec: LW r30,16(r29)
At PC = 0xf0: JR r0,r31
At PC = 0xf4: ADDIU r29,r29,24
At PC = 0x8: JAL 8
At PC = 0xc: ADDU r4,r0,r0
At PC = 0x20: ADDIU r2,r0,1
At PC = 0x24: SYSCALL
Exception: syscall
***User program exit normally***
Finishing thread "User Program 3"
Sleeping thread "User Program 3"

```

```

Switching from thread "User Program 3" to thread "main"
Now in thread "main"
Deleting thread "User Program 3"
Finishing thread "main"
Sleeping thread "main"
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 131, idle 0, system 80, user 51
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...

```

上述代码无法检验对数据段的初始化，因此我修改测试代码如下：

```

#include "syscall.h"

int
main()
{
    int i=514, A[1];
    for(i=0; i<514; ++i)
    {
        A[0] = i;
    }
    Exit(A[0]);
    /* not reached */
}

```

运行结果如下（由于涉及到循环，控制流过长，故省略）：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d Et -X ..//test/exit
Thread 1 is being created
Thread 2 is being created
Forking thread "User Program 1" with func = 0x565f2b99, arg = 1
Putting thread User Program 1 on ready list.
Forking thread "User Program 2" with func = 0x565f2b99, arg = 2
Putting thread User Program 2 on ready list.
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "User Program 1"
User Program 1 starts to run!
===== PageTable Info =====
numpages = 11
VPN      PPN      valid    RD    Used    Dirty
0        0        0        0     0       0
1        0        0        0     0       0
2        0        0        0     0       0
3        0        0        0     0       0
4        0        0        0     0       0
5        0        0        0     0       0
6        0        0        0     0       0
7        0        0        0     0       0
8        0        0        0     0       0
9        0        0        0     0       0
10       0        0        0     0       0
=====
*****User program exit with status 513*****
Finishing thread "User Program 1"
Sleeping thread "User Program 1"
Switching from thread "User Program 1" to thread "User Program 2"
User Program 2 starts to run!
===== PageTable Info =====
numpages = 11
VPN      PPN      valid    RD    Used    Dirty
0        0        0        0     0       0
1        0        0        0     0       0
2        0        0        0     0       0
3        0        0        0     0       0
4        0        0        0     0       0
5        0        0        0     0       0
6        0        0        0     0       0
7        0        0        0     0       0
8        0        0        0     0       0
9        0        0        0     0       0
10       0        0        0     0       0
=====
*****User program exit with status 513*****
Finishing thread "User Program 2"
Sleeping thread "User Program 2"
Switching from thread "User Program 2" to thread "main"
Now in thread "main"
Deleting thread "User Program 2"
Finishing thread "main"
Sleeping thread "main"
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 13490, idle 0, system 60, user 13430
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...

```

可以看到，代码段和数据段的初始化都没有问题，程序可以正常运行。且支持了多个线程同时存在于内存中，因此上述多线程支持可以认为是正确的。

Exercise6

这一部分要求我们实现缺页异常。手册中指明，先前我们实现的只是 TLB 缺失而带来的异常，现在我们要实现的是，当页表条目中的物理页无效时，如何进行异常处理来从磁盘中调入物理页。

注意到 Nahcos 中没有对物理磁盘的模拟，所以我选择用**外部文件来模拟 Nahcos 中的线程所对应的磁盘空间**。在为每个线程进行地址空间初始化时，我们应该为地址空间分配一个对应的文件，并且把代码和数据段储存到这个文件中，作为放在外存的虚拟内存，等待后面缺页异常时调用。因此，对应地，我们需要修改 AddrSpace 的构造方法。这里，我

添加了 USE_DISK 宏以表示开启缺页中断机制。我们大致需要修改的有以下几部分内容：

(1) 为每个地址空间指定一份磁盘文件。这里我使用 VirtualMemory 接线程计数的命名方式来区分每个地址空间的磁盘文件。与此同时，AddrSpace 中页添加了 VMName 成员以记录虚拟内存名称，并引入全局变量 SpaceCnt 来记录线程个数（生成的地址空间个数）。

```
#ifndef USE_DISK
    ASSERT(numPages <= NumPhysPages); // check we're not trying
                                    // to run anything too big --
                                    // at least until we have
                                    // virtual memory
#else
    char str[20];
    sprintf(str, "VirtualMemory%d", SpaceCnt++);
    VMName = strdup(str);
    bool succeed_creating_file = fileSystem->Create(VMName, MemorySize);
    ASSERT(succeed_creating_file);
#endif
```

(2) 修改页表项的初始化方式。先前的实现中都是直接分配好了页表，但这里我选择修改为初始的页表项都是无效的，等到需要使用时再加载（为了测试缺页中断）：

```
// first, set up the translation
pageTable = new TranslationEntry[numPages];
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
#ifndef USE_DISK
#if USE_BITMAP
    pageTable[i].physicalPage = machine->allocateMem();
    ASSERT(pageTable[i].physicalPage != -1);
#else
    pageTable[i].physicalPage = i;
#endif
    pageTable[i].valid = TRUE;
#else
    pageTable[i].valid = FALSE;
#endif
    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) 把代码和数据按照虚拟地址的布局方式来分配到磁盘中的虚拟内存空间。注意到最开始的单线程实现中，本质上相当于把代码和数据按照程序虚拟地址空间的布局从 0 开始写入内存，因此我们可以仿照上面的实现来完成到磁盘虚拟内存的写入：

```
#else
    DEBUG('P', "Demand paging: copy executable to virtual memory %s!\n", VMName);
    OpenFile *vm = fileSystem->Open(VMName);
    char *VM_tmp = new char[size];
    bzero(VM_tmp, size);
    if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%08x, size %d\n",
              noffH.code.virtualAddr, noffH.code.size);
        executable->ReadAt(&(VM_tmp[noffH.code.virtualAddr]),
                            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%08x, size %d\n",
              noffH.initData.virtualAddr, noffH.initData.size);
        executable->ReadAt(&(VM_tmp[noffH.initData.virtualAddr]),
                            noffH.initData.size, noffH.initData.inFileAddr);
    }
    vm->WriteAt(VM_tmp, size, 0);
    delete vm;
#endif
```

注意到原有的地址翻译实现中，只使用页表时，如果页表项无效，认为是地址太大了，因此我们需要开启 **TLB** 选项。除此之外，为了支持多线程，我们还需要开启 **Exercise4** 中实现的位图选项，以管理空闲内存。

在之前的异常处理中，我们只处理了 TLB 的缺失，现在我们要进一步实现缺页异常的处理：

(1) **修改原有的 TLB 缺失的异常处理函数**，对与开启缺页中断处理的情况，当获取的页表项无效时，进行进一步的异常处理：

```
#ifndef USE_DISK
    ASSERT(page.valid);
#else
    if(!page.valid)
    {
        DEBUG('P', "====> Page Miss Found, vpn is %d 0x%llx\n", vpn, vpn);
        PageFaultHandler(vpn);
    }
#endif
```

(2) **实现缺页中断处理函数**。主要思想是：首先从内存中选取一个空闲的物理页，然后从磁盘上储存的虚拟内存中拿出虚拟地址对应的内容，放到刚才找到的空闲物理页中。

为了检查实现，我还一并打印了当前线程的页表情况：

```
void
PageFaultHandler(int vpn)
{
    int physicalPage = -1;
#ifdef USE_BITMAP
    physicalPage = machine->allocateMem();
#else
    ASSERT(FALSE);
#endif

    machine->pageTable[vpn].physicalPage = physicalPage;

    DEBUG('P', "Load page from virtual memory %s\n", currentThread->space->VMName);
    OpenFile *vm = fileSystem->Open(currentThread->space->VMName);
    ASSERT(vm!=NULL);
    vm->ReadAt(&(machine->mainMemory[PageSize*physicalPage]), PageSize, vpn*PageSize);
    delete vm;

    machine->pageTable[vpn].valid = TRUE;
    machine->pageTable[vpn].use = FALSE;
    machine->pageTable[vpn].dirty = FALSE;
    machine->pageTable[vpn].readOnly = FALSE;

    currentThread->space->PrintAddrState();
}
```

接下来我们便可以开始测试了。测试使用的用户级程序如下：

```
#include "syscall.h"

int
main()
{
    int i=514, A[1];
    for(i=0; i<514; ++i)
    {
        A[0] = 1;
    }
    Exit(114);
    /* not reached */
}
```

运行结果如下（其中，debug 的-P 选项表示在实现缺页中断时添加的 debug 输出）：

```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d P -X ../test/exit
Thread 1 is being created
Demand paging: copy executable to virtual memory VirtualMemory0!
Thread 2 is being created
Demand paging: copy executable to virtual memory VirtualMemory1!
User Program 1 starts to run!
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      0       0     0      0
1      0      0       0     0      0
2      0      0       0     0      0
3      0      0       0     0      0
4      0      0       0     0      0
5      0      0       0     0      0
6      0      0       0     0      0
7      0      0       0     0      0
8      0      0       0     0      0
9      0      0       0     0      0
10     0      0       0     0      0
=====
==> Page Miss Found, vpn is 0 0x0!
Load page from virtual memory VirtualMemory0
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      1       0     0      0
1      0      0       0     0      0
2      0      0       0     0      0
3      0      0       0     0      0
4      0      0       0     0      0
5      0      0       0     0      0
6      0      0       0     0      0
7      0      0       0     0      0
8      0      0       0     0      0
9      0      0       0     0      0
10     0      0       0     0      0
=====
==> Page Miss Found, vpn is 1 0x1!
Load page from virtual memory VirtualMemory0
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      0       0     0      0
1      0      0       0     0      0
2      0      0       0     0      0
3      0      0       0     0      0
4      0      0       0     0      0
5      0      0       0     0      0
6      0      0       0     0      0
7      0      0       0     0      0
8      0      0       0     0      0
9      0      0       0     0      0
10     0      0       0     0      0
=====
==> Page Miss Found, vpn is 0 0x0!
Load page from virtual memory VirtualMemory0
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      1       0     0      0
1      0      0       0     0      0
2      0      0       0     0      0
3      0      0       0     0      0
4      0      0       0     0      0
5      0      0       0     0      0
6      0      0       0     0      0
7      0      0       0     0      0
8      0      0       0     0      0

```

```

9      0      0      0      0      0
10     0      0      0      0      0
=====
====> Page Miss Found, vpn is 10 0xa!
Load page from virtual memory VirtualMemory0
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      1      0      0      0
1      1      1      0      0      0
2      0      0      0      0      0
3      0      0      0      0      0
4      0      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
7      0      0      0      0      0
8      0      0      0      0      0
9      0      0      0      0      0
10     2      1      0      0      0
=====
====> Page Miss Found, vpn is 2 0x2!
Load page from virtual memory VirtualMemory0
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      1      0      0      0
1      1      1      0      0      0
2      3      1      0      0      0
3      0      0      0      0      0
4      0      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
7      0      0      0      0      0
8      0      0      0      0      0
9      0      0      0      0      0
10     2      1      0      0      0
=====
User Program 2 starts to run!
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      0      0      0      0
1      0      0      0      0      0
2      0      0      0      0      0
3      0      0      0      0      0
4      0      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
7      0      0      0      0      0
8      0      0      0      0      0
9      0      0      0      0      0
10     0      0      0      0      0
=====
====> Page Miss Found, vpn is 10 0xa!
Load page from virtual memory VirtualMemory1
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      0      0      0      0
1      0      0      0      0      0
2      0      0      0      0      0
3      0      0      0      0      0
4      0      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
7      0      0      0      0      0
8      0      0      0      0      0
9      0      0      0      0      0
10     0      1      0      0      0
=====
====> Page Miss Found, vpn is 2 0x2!
Load page from virtual memory VirtualMemory1
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      0      0      0      0
1      0      0      0      0      0
2      1      1      0      0      0
3      0      0      0      0      0
4      0      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
7      0      0      0      0      0
8      0      0      0      0      0
9      0      0      0      0      0

```

```

10      0      1      0      0      0
=====
====> Page Miss Found, vpn is 1 0x1!
Load page from virtual memory VirtualMemory1
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      0      0       0     0       0
1      2      1       0     0       0
2      1      1       0     0       0
3      0      0       0     0       0
4      0      0       0     0       0
5      0      0       0     0       0
6      0      0       0     0       0
7      0      0       0     0       0
8      0      0       0     0       0
9      0      0       0     0       0
10     0      1       0     0       0
=====
====> Page Miss Found, vpn is 0 0x0!
Load page from virtual memory VirtualMemory1
===== PageTable Info =====
numpages = 11
VPN    PPN    valid   RD    Used   Dirty
0      3      1       0     0       0
1      2      1       0     0       0
2      1      1       0     0       0
3      0      0       0     0       0
4      0      0       0     0       0
5      0      0       0     0       0
6      0      0       0     0       0
7      0      0       0     0       0
8      0      0       0     0       0
9      0      0       0     0       0
10     0      1       0     0       0
=====
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

```

可以看到，添加了缺页中断机制以后，多线程仍能正常运行，因此可以认为在限制内存大小的情况下，上面的实现是正确的。

实际上，为了进一步完善内存管理机制，我们还应该支持物理页全部占用时的物理页面替换算法。这一部分在下一个 exercise 中一并完成。

Exercise7

这一部分要求我们实现 lazy-loading 的物理页面调度机制。也就是说，只有在用到这个页面时，才进行页面的调换。

注意到，在上一个 exercise 中，我们对地址空间初始化时，已经让所有的页表项在最开始都是无效的，这样便可以实现在初始化时的 lazy-loading。

此外，为了支持运行更大的程序，由于现在添加了磁盘机制的模拟，所以我们也可以在进行物理页面调度的时候使用 lazy-loading。当处理缺页中断时，如果发现所有的页面都被占用了，我们首先替换当前线程页表项中被占用但是未被修改过的页面。如果页面都被修改过，那么就根据遍历顺序来选择当前线程页表项中的一个页面，把它写回对应的磁盘文件中，并让这个页面无效。实现如下：

(1) 首先在 PageFaultHandler 中添加置换页表项的处理：

```

void
PageFaultHandler(int vpn)
{
    int physicalPage = -1;
#ifdef USE_BITMAP
    physicalPage = machine->allocateMem();
#else
    ASSERT(FALSE);
#endif
    if(physicalPage == -1)
    {
        physicalPage = ReplacePage();
    }
    machine->pageTable[vpn].physicalPage = physicalPage;

    DEBUG('P', "Load page from virtual memory %s\n", currentThread->space->VMName);
    OpenFile *vm = fileSystem->Open(currentThread->space->VMName);
    ASSERT(vm!=NULL);
    vm->ReadAt(&(machine->mainMemory[PageSize*physicalPage]), PageSize, vpn*PageSize);
    delete vm;

    machine->pageTable[vpn].valid = TRUE;
    machine->pageTable[vpn].use = FALSE;
    machine->pageTable[vpn].dirty = FALSE;
    machine->pageTable[vpn].readOnly = FALSE;

    currentThread->space->PrintAddrState();
}

```

(2) 接下来，实现这个替换函数：

```

int
ReplacePage()
{
    DEBUG('P', "==> Try to find a physical page in current pagetable.\n");
    int physicalPage = -1;
    for(int i=0; i<machine->pageTableSize; ++i)
    {
        if(machine->pageTable[i].valid)
        {
            if(!machine->pageTable[i].dirty)
            {
                DEBUG('P', "==> Find an unmodified physical page.\n");
                machine->pageTable[i].valid = FALSE;
                physicalPage = machine->pageTable[i].physicalPage;
                break;
            }
        }
    }
    if(physicalPage == -1) // all has been modified
    {
        for(int i=0; i<machine->pageTableSize; ++i)
        {
            if(machine->pageTable[i].valid)
            {
                DEBUG('P', "==> Find a modified physical page.\n");
                machine->pageTable[i].valid = FALSE;
                physicalPage = machine->pageTable[i].physicalPage;

                OpenFile *vm = fileSystem->Open(currentThread->space->VMName);
                ASSERT(vm!=NULL);
                vm->WriteAt(&(machine->mainMemory[PageSize*machine->pageTable[i].physicalPage]), PageSize, i*PageSize);
                delete vm;
                break;
            }
        }
    }
    return physicalPage;
}

```

添加上述修改后，如果我们装载更多的用户级线程，试验结果如下（matmult 中的 N 修改为 16）：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -d E -X ./test/matmult
Thread 1 is being created
Thread 2 is being created
Thread 3 is being created
Thread 4 is being created
Thread 5 is being created
Thread 6 is being created
Thread 7 is being created
Thread 8 is being created
Thread 9 is being created
Thread 10 is being created
Thread 11 is being created
Thread 12 is being created
Thread 13 is being created
Thread 14 is being created
Thread 15 is being created
Thread 16 is being created
Thread 17 is being created
Thread 18 is being created
Thread 19 is being created
Thread 20 is being created
Thread 21 is being created
Thread 22 is being created
Thread 23 is being created
Thread 24 is being created
Thread 25 is being created
Thread 26 is being created
Thread 27 is being created
Thread 28 is being created
Thread 29 is being created
Thread 30 is being created
Thread 31 is being created
Thread 32 is being created
Thread 33 is being created
Thread 34 is being created
Thread 35 is being created
Thread 36 is being created
Thread 37 is being created
Thread 38 is being created
Thread 39 is being created
Thread 40 is being created
Thread 41 is being created
Thread 42 is being created
Thread 43 is being created
Thread 44 is being created
Thread 45 is being created
Thread 46 is being created
Thread 47 is being created
Thread 48 is being created
Thread 49 is being created
Thread 50 is being created
Thread 51 is being created
Thread 52 is being created
Thread 53 is being created
Thread 54 is being created
Thread 55 is being created
Thread 56 is being created
Thread 57 is being created
Thread 58 is being created
Thread 59 is being created
Thread 60 is being created
Thread 61 is being created
Thread 62 is being created
Thread 63 is being created
Thread 64 is being created
Thread 65 is being created
Thread 66 is being created
Thread 67 is being created
Thread 68 is being created
Thread 69 is being created
Thread 70 is being created
Thread 71 is being created
Thread 72 is being created
Thread 73 is being created
Thread 74 is being created
Thread 75 is being created
Thread 76 is being created
Thread 77 is being created
Thread 78 is being created
Thread 79 is being created
Thread 80 is being created
Thread 81 is being created
Thread 82 is being created
Thread 83 is being created
Thread 84 is being created
Thread 85 is being created
```



```
*****User program exit with status 3600*****
User Program 80 starts to run!
*****User program exit with status 3600*****
User Program 81 starts to run!
*****User program exit with status 3600*****
User Program 82 starts to run!
*****User program exit with status 3600*****
User Program 83 starts to run!
*****User program exit with status 3600*****
User Program 84 starts to run!
*****User program exit with status 3600*****
User Program 85 starts to run!
*****User program exit with status 3600*****
User Program 86 starts to run!
*****User program exit with status 3600*****
User Program 87 starts to run!
*****User program exit with status 3600*****
User Program 88 starts to run!
*****User program exit with status 3600*****
User Program 89 starts to run!
*****User program exit with status 3600*****
User Program 90 starts to run!
*****User program exit with status 3600*****
User Program 91 starts to run!
*****User program exit with status 3600*****
User Program 92 starts to run!
*****User program exit with status 3600*****
User Program 93 starts to run!
*****User program exit with status 3600*****
User Program 94 starts to run!
*****User program exit with status 3600*****
User Program 95 starts to run!
*****User program exit with status 3600*****
User Program 96 starts to run!
*****User program exit with status 3600*****
User Program 97 starts to run!
*****User program exit with status 3600*****
User Program 98 starts to run!
*****User program exit with status 3600*****
User Program 99 starts to run!
*****User program exit with status 3600*****
User Program 100 starts to run!
*****User program exit with status 3600*****
```

```
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 32596922, idle 0, system 2020, user 32594902
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

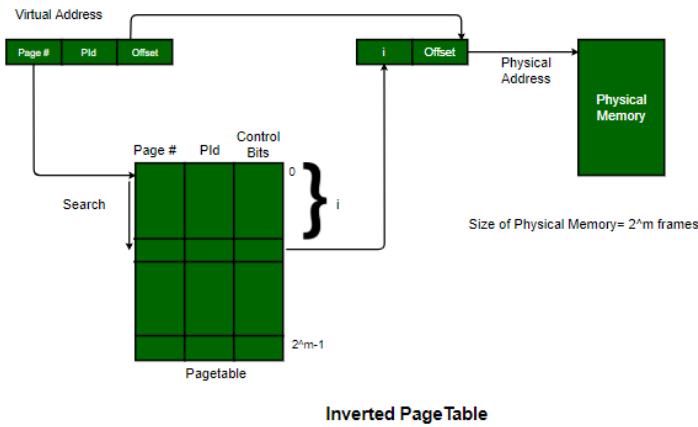
Cleaning up...
```

可以看到，原来即便是只加载一个都无法运行的线程，在支持了虚拟地址空间和 lazy-loading 以后，即便是加载多个线程也可以全部成功运行。可以认为上述实现是正确的。

Challenge

这一部分中，我完成了 Challenge2——实现倒排页表。

倒排页表通过把物理页面的每个页框对应表项，来组织和管理内存。一种实现方法如下图所示（图片来源 <https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/>）：



可以看到，与原来的页表不同，倒排页表还要存储 PID（Nachos 中应该为线程 id）。

而且考虑到倒排页表也是一种管理全局内存的机制，所以倒排页表应该和前面几个练习中的实现不兼容。因此，我们应该做出以下修改。这里我们使用 INVERTED_PAGETABLE 来对倒排页表进行条件编译：

(1) 在 TranslationEntry 这一页表类中加入线程 ID 成员。这样，我们可以用 physicalPage 指明这个表项对应的物理页框（实际上，应该等于数组下标，虽然有冗余，但是原来的结构中提供了这一成员，因此也进行标记）， virtualPage 指明对应线程中的哪个虚拟页， TID 指明所属的线程：

```
#ifdef INVERTED_PAGETABLE
    int TID;
#endif
```

(2) 由于倒排页表是所有线程共用的，因此 AddrSpace ::RestoreState 中的恢复页表在倒排页表机制中不再需要，所以要用条件编译去除：

```
void AddrSpace::RestoreState()
{
#ifndef INVERTED_PAGETABLE
    machine->pageTable = pageTable;
    machine->pageTableSize = numPages;
#endif
}
```

(3) 由于倒排页表是内存管理成员，因此与前面实现的位图类似，我们应该把它放在 Machine 类中声明，并且在构造方法中初始化。注意到我们只需要位图和倒排页表中的一个，所以保险起见，我加上了 ASSERT 语句。我们为机器的倒排页表分配与物理页框数相同个数的页表条目，每个条目用 physicalPage 来标识对应的物理页，用 TID 标识归属：

```
#if USE_BITMAP && INVERTED_PAGETABLE
    ASSERT(FALSE);
#endif
```

```

#ifndef INVERTED_PAGETABLE

#ifdef USE_TLB
    tlb = new TranslationEntry[TLBSize];
    for (i = 0; i < TLBSize; i++)
        tlb[i].valid = FALSE;
    pageTable = NULL;
#else // use linear page table
    tlb = NULL;
    pageTable = NULL;
#endif

#ifdef USE_BITMAP
    bitmap = 0;
#endif

#else
    pageTable = new TranslationEntry[NumPhysPages];
    pageTableSize = NumPhysPages;
    for(int i=0; i<NumPhysPages; ++i)
    {
        pageTable[i].virtualPage = -1;
        pageTable[i].physicalPage = -1;
        pageTable[i].valid = FALSE;
        pageTable[i].use = FALSE;
        pageTable[i].readonly = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].TID = -1;
    }
#endif

```

(4) 为了节约空间，我们可以复用先前实现的 allocateMem 与 freeMem，使用条件编译来区分我们采用位图机制还是倒排页表机制。allocateMem 中，我们只需要为线程（地址空间）提供一个未使用的物理页（PPN），后续的处理可以在 AddrSpace 的初始化中实现；而 freeMem 中，我们可以利用 currentThread 这一全局变量来寻找我们要释放的页表项：

```

#ifndef INVERTED_PAGETABLE
    for(int pos=0; pos<NumPhysPages; pos++)
    {
        if(!pageTable[pos].valid)
        {
            return pos;
        }
    }
    printf("All physical pages have been used!!!\n");
    ASSERT(FALSE);
    return -1;
#endif

#ifndef INVERTED_PAGETABLE
    for(int pos=0; pos<NumPhysPages; ++pos)
    {
        if(pageTable[pos].valid && pageTable[pos].TID==currentThread->getTID())
        {
            pageTable[pos].valid = FALSE;
            DEBUG('I', "Thread %d has freed physical page %d\n", currentThread->getTID(), pos);
        }
    }
    DEBUG('I', "Thread %d has freed all physical pages it occupied\n", currentThread->getTID());
#endif

```

(5) 如 (4) 中所描述，我们也要修改 AddrSpace 的构造方法，以利用倒排页表。我们选取用户级程序所需个数的倒排页表项，并初始化 Mahcine 中的倒排页表项。：

```

-->
    ASSERT(numPages <= NumPhysPages);
    for(int i=0; i<numPages; ++i)
    {
        int physicalPage = machine->allocateMem();
        //printf("%d\n", physicalPage);
        machine->pageTable[physicalPage].virtualPage = i;
        machine->pageTable[physicalPage].physicalPage = physicalPage;
        machine->pageTable[physicalPage].valid = TRUE;
        machine->pageTable[physicalPage].use = FALSE;
        machine->pageTable[physicalPage].readOnly = FALSE;
        machine->pageTable[physicalPage].dirty = FALSE;
        machine->pageTable[physicalPage].TID = currentThread->getTID();
        for(int j=0; j<PageSize; ++j)
        {
            machine->mainMemory[physicalPage*PageSize+j] = 0;
        }
    }

    printf("=====PAGE TABLE=====\\n");
    for(int i=0; i<NumPhysPages; ++i)
    {
        printf("%d\t%d\t%d\\n", machine->pageTable[i].virtualPage, machine->pageTable[i].physicalPage,
               machine->pageTable[i].valid, machine->pageTable[i].TID);
    }
    printf("=====\\n");

    // allocate data and code sections, we need to translate sections' virtualAddr to physicalAddr
    // one byte per loop
    for(int VA=noffH.code.virtualAddr, cnt=0; VA<noffH.code.size; ++VA, ++cnt)
    {
        unsigned int VPN = VA / PageSize;
        unsigned int PPN = 0;
        for(PPN=0; PPN<NumPhysPages; ++PPN)
        {
            if(machine->pageTable[PPN].valid && machine->pageTable[PPN].virtualPage==VPN
               && machine->pageTable[PPN].TID == currentThread->getTID())
                break;
        }
        unsigned int PPO = VA % PageSize;
        unsigned int PA = PPN*PageSize + PPO;
        //printf("%d 0x%08x %d 0x%08x\\n", PA, PA, VA, VA);
        executable->ReadAt(&(machine->mainMemory[PA]),
                            1, noffH.code.inFileAddr+cnt);
        //printf("%d 0x%08x\\n", machine->mainMemory[PA], machine->mainMemory[PA]);
    }

    for(int VA=noffH.initData.virtualAddr, cnt=0; VA<noffH.initData.size; ++VA, ++cnt)
    {
        unsigned int VPN = VA / PageSize;
        unsigned int PPN = 0;
        for(PPN=0; PPN<NumPhysPages; ++PPN)
        {
            if(machine->pageTable[PPN].valid && machine->pageTable[PPN].virtualPage==VPN
               && machine->pageTable[PPN].TID == currentThread->getTID())
                break;
        }
        unsigned int PPO = VA % PageSize;
        unsigned int PA = PPN*PageSize + PPO;
        //printf("%d 0x%08x %d 0x%08x\\n", PA, PA, VA, VA);
        executable->ReadAt(&(machine->mainMemory[PA]),
                            1, noffH.initData.inFileAddr+cnt);
        //printf("%d 0x%08x\\n", machine->mainMemory[PA], machine->mainMemory[PA]);
    }
}
#endif

```

(6) 除了以上的实现，我们还需要修改地址翻译的机制。与先前的实现类似，我们需要检查地址是否越界、页表项是否有效。除此之外，我们还需要检查，这个虚拟地址对应条目的线程与当前运行线程是否相同。只有满足以上条件，我们才能进行地址翻译：

(7) 还有一个细节需要注意。在 Exit 的处理函数中，我们也需要加上倒排页表的条件编译选项，以保证退出时当前线程占用的页面全部被释放：

```
#ifdef USE_BITMAP || INVERTED_PAGETABLE  
    machine->freeMem();  
#endif
```

以上的修改实现了倒排页表的基本功能，现在我们可以用先前编写的函数来做简单的测试：

```
#include "syscall.h"

int
main()
{
    int i=514, A[1];
    for(i=0; i<514; ++i)
    {
        A[0] = i;
    }
    Exit(A[0]);
    /* not reached */
}
```

```

./nachos -d EI -x ../test/exit
=====PAGE TABLE=====
0      0      1      0
1      1      1      0
2      2      1      0
3      3      1      0
4      4      1      0
5      5      1      0
6      6      1      0
7      7      1      0
8      8      1      0
9      9      1      0
10     10     1      0
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
=====PAGE TABLE=====

```

```

*****User program exit with status 513*****
Thread 0 has freed physical page 0
Thread 0 has freed physical page 1
Thread 0 has freed physical page 2
Thread 0 has freed physical page 3
Thread 0 has freed physical page 4
Thread 0 has freed physical page 5
Thread 0 has freed physical page 6
Thread 0 has freed physical page 7
Thread 0 has freed physical page 8
Thread 0 has freed physical page 9
Thread 0 has freed physical page 10
Thread 0 has freed all physical pages it occupied
=====PAGE TABLE=====
0      0      0      0
1      1      0      0
2      2      0      0
3      3      0      0
4      4      0      0
5      5      0      0
6      6      0      0
7      7      0      0
8      8      0      0
9      9      0      0
10     10     0      0
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
-1     -1     0      -1
=====PAGE TABLE=====

```

```
-1      -1      0      -1
-1      -1      0      -1
-1      -1      0      -1
=====
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 6715, idle 0, system 10, user 6705
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...
```

可以看到，用户级程序可以被正常地分配到页表，退出状态也与预期相同，且结束后使用过的页表也都被释放了，因此上述实现可以认为是正确的。

内容二：遇到的困难以及收获

首先，我在条件编译的实现上遇到了一些问题。由于 Exercise 和 Challenge 中给出了很多情境，但某些情境又是冲突的（比如 TLB 的两种算法，我们每次运行时只能选择一种），观察到代码中的`#ifdef...#endif`语句块后，我觉得条件编译是实现不同条件下均能运行的好方法。查阅资料后，我发现，条件编译的宏选项可以在 Makefile 文件中添加并开启，因此对于不同的 Exercise，我在 Makefile 中选取了不同的条件编译选项，同时我也为不同 Exercise 开启了不同的条件编译选项，这样便可以实现互斥。

其次，我在多线程实现时遇到了一些问题。Nachos 原有的加载代码、数据段机制只适用于单个线程，因为它直接按照虚拟地址布局，从内存的地址为 0 的位置开始加载。但为了支持多线程，我们需要支持位图等全局数据结构的分配，这样便需要修改加载机制。

除此之外，由于 Nachos 不支持真实的磁盘模拟，因此在 Exercise7 中，我只能退而求其次，采取使用磁盘文件来模拟虚拟内存放在磁盘上的情形。此外，为了支持多线程，目前想到的做法是开启多个磁盘文件，按照其地址空间布局来存储。是否还有更好的做法留待进一步探索。

最后，倒排页表的实现中，目前仍然只实现了单道程序。对于多道程序和 TLB 的支持，本质上与 Exercise2-7 的实现相似。

内容三：对课程或 Lab 的意见和建议

希望可以把引导手册进一步完善，现在每次阅读代码的时候需要我们递归阅读和寻找其他额外的代码，但我们往往不知道应该去哪里能找到，这样浪费了许多不必要的时间。我觉得可以把某些常用函数（或者重要辅助函数）的位置标注出来，这样可以节省遍历搜索的时间。

也希望可以为调研类的作业提供更多的提示和说明。现在我们只能漫无目的地去遍历搜索百度、google 的结果，我们无法确认博客等网站表述内容的正确性和严谨性，而且有的调研题目很难找到相关博客。让我们在短时间内查询所有体系结构/操作系统的手册，对于还有许多其他课程的本科生来说在时间上不是很现实。所以希望课程可以提供更多相关资源，或者提示我们应该去看手册的哪些章节，去找什么样的权威资料。这样既可以让我们学到准确的知识，也可以提升我们的学习效率。

内容四：参考文献

1. MIPS 寄存器标号：

<http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html>

2. C++中的 register 关键字：

<https://blog.csdn.net/liu537192/article/details/50194515>

3. Makefile 中用宏定义进行条件编译

<https://www.cnblogs.com/welzh/p/5607824.html>

4. 位图的知识：《现代操作系统》相关章节（3.2.3 的第 1 节）

5. 倒排页表相关：

<https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/>

<https://web.eecs.umich.edu/~akamil/teaching/sp04/040104.pdf>

《现代操作系统》3.3.4 的第 2 节