

## Lab6 Nachos 用户程序与系统调用

### 1. Nachos 可执行程序.noff 文件的格式组成

Nachos 的应用程序是作者自己定义的一种文件类型，文件头部分结构相对简单，编程方便。分析析../test/Makefile 可以看出，首先利用交叉编译器提供的 gcc、as、ld 等工具将nachos 应用程序.c 编译链接成 .coff 文件，然后利用../bin/arch/unknown-i386-linux/bin/coff2noff 将 coff 文件转变成 noff 文件：

```
.....
$(all_coff): $(obj_dir)/%.coff: $(obj_dir)/start.o $(obj_dir)/%.o
    @echo ">>> Linking" $(obj_dir)/$(notdir $@) "<<<"
    $(LD) $(LDFLAGS) $^ -o $(obj_dir)/$(notdir $@)

$(all_noff): $(bin_dir)/%.noff: $(obj_dir)/%.coff
    @echo ">>> Converting to noff file:" $@ "<<<"
    $(coff2noff) $^ $@
    ln -sf $@ $(notdir $@)
.....
```

简单的可执行文件头结构便于程序实现（参见 AddrSpace::AddrSpace()中将程序装入主存，并设置 PC 的值部分代码）。

- **coff 文件架构**

COFF——通用对象文件格式（Common Object File Format），是一种很流行的对象文件格式，coff文件本质上就是代码在编译阶段产生的文件。COFF文件一共有8种数据，如下表所示，其中段落头和段落数据可以有多节。默认情况下，COFF文件包含3个段：**.text**：通常包含可执行代码；**.data**：通常包含已初始化的数据；**.bss**：通常为未初始化的数据保留空间。其中，文件头~行号入口表与加载文件密切相关。

COFF文件结构	说明	描述
File header	文件头	用来保存COFF文件的基本信息,如文件标识,各个表的位置等等
Optional Header	可选头	类似于文件头，也是用来保存信息的，主要保存在文件头中没有描述到的信息。
Section Header	段落头	用来描述段落信息的，每个段落都有一个段落头来描述，段落的数目会在文件头中指出。
Section Data	段落数据	段落数据通常是COFF文件中最大的数据段，每个段落真正的数据就保存在这个位置，可以从段落头的描述中看到相关信息。
Relocation Directives	重定位表	这个表通常只存在于目标文件中，他用来描述COFF文件中符号的重定位信息。
Line Numbers	行号表	
Symbol Table	符号表	用来保存COFF文件中所用到的所有符号的信息，链接多个COFF文件时，这个表帮助我们重定位符号。调试程序时也要用到它。
String Table	字符串表	保存字符串的。用于保存一些字符个数超出了符号表和去段头的名称数组最大个数的字符串。

- noff 文件架构

```
#define NOFFMAGIC    0xbadfad

typedef struct segment {
    int virtualAddr;        /* location of segment in virt addr space */
    int inFileAddr;         /* location of segment in this file */
    int size;               /* size of segment */
} Segment;

typedef struct noffHeader {
    int noffMagic;          /* should be NOFFMAGIC */
    Segment code;           /* executable code segment */
    Segment initData;       /* initialized data segment */
    Segment uninitData;     /* uninitialized data segment should be zero'ed
before use */
} NoffHeader;
```

可以看出，文件头给出了每个段的大小，在文件中的开始位置，以及程序的入口地址等信息；AddrSpace 的构造函数在将 NOFF 文件装入内存之前，先打开该文件，读入文件头，然后根据文件头信息确定程序所占用的空间，将相应的段装入内存，将程序的入口地址，给 PC 赋值。

## 2. 阅读../test 目录下的几个 Nachos 应用程序，理解 Nachos 应用程序的编程语法，了解用户进程是如何通过系统调用与操作系统内核进行交互的；

- halt.s

```
.file 1 "halt.c"
gcc2_compiled.:
__gnu_compiled_c:
.text
.align 2
.globl main
.ent main
main:
    .frame $fp,40,$31      # vars= 16, regs= 2/0, args= 16, extra= 0
    .mask 0xc0000000,-4
    .fmask 0x00000000,0
    subu    $sp,$sp,40
    sw      $31,36($sp)
    sw      $fp,32($sp)
    move    $fp,$sp
    jal     __main
    li      $2,3           # 0x00000003
    sw      $2,24($fp)
    li      $2,2           # 0x00000002
    sw      $2,16($fp)
    lw      $2,20($fp)
    addu    $3,$2,-1
    sw      $3,20($fp)
    lw      $2,16($fp)
    lw      $3,20($fp)
    subu    $2,$2,$3
    lw      $3,24($fp)
    addu    $2,$3,$2
    sw      $2,24($fp)
    jal     Halt
$L1:
    move    $sp,$fp
    lw      $31,36($sp)
    lw      $fp,32($sp)
    addu    $sp,$sp,40
    j       $31
.end main
```

- ▶ Include a .frame psuedo-op:

```
.frame framereg, framesize, returnreg
```

Creates a virtual frame pointer (\$fp):  $\$sp + framesize$

- ▶ Save the registers you allocated space for

```
.mask    bitmask, frameoffset
```

```
sw      reg, framesize+frameoffset-N($sp)
```

.mask: used to specify registers to be stored and where they are store

One bit in bitmask for each register saved.

frameoffset: offset from virtual frame pointer. Negative.

$N$  should be 0 for the highest numbered register saved and then incremented by 4 for each lowered numbered register:

```
sw $31, framesize+frameoffset($sp)
```

```
sw $17, framesize+frameoffset-4($sp)
```

```
sw $6, framesize+frameoffset-8($sp)
```

- ▶ Save any floating point register:

```
.fmask    bitmask, frameoffset
```

```
s.[sd]    reg, framesize+frameoffset-N($sp)
```

Use .fmask for saving register

在该汇编代码中，main 函数的帧栈首先通过 \$sp 开辟 40 个大小的空间，保存返回地址，将帧指针 \$sp 指向栈顶，并将返回地址保存到 \$fp 中，接着移交 \$fp 和 \$sp，运行当前程序，在当前程序运行结束后，根据保存的返回地址恢复到之前的运行例程中。

进入目录 `../userprog`，运行以下代码：

```
make clean
make
./nachos -x ../test/halt.noff -d m
```

程序有如下输出：

```
Starting thread "main" at time 120
At PC = 0x0: JAL 52
At PC = 0x4: SLL r0,r0,0
At PC = 0xd0: ADDIU r29,r29,-40
At PC = 0xd4: SW r31,36(r29)
At PC = 0xd8: SW r30,32(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: ADDIU r2,r0,3
At PC = 0xe8: SW r2,24(r30)
At PC = 0xec: ADDIU r2,r0,2
At PC = 0xf0: SW r2,16(r30)
At PC = 0xf4: LW r2,16(r30)
At PC = 0xf8: SLL r0,r0,0
At PC = 0xfc: ADDIU r3,r2,-1
At PC = 0x100: SW r3,20(r30)
At PC = 0x104: LW r2,16(r30)
At PC = 0x108: LW r3,20(r30)
At PC = 0x10c: SLL r0,r0,0
At PC = 0x110: SUBU r2,r2,r3
At PC = 0x114: LW r3,24(r30)
At PC = 0x118: SLL r0,r0,0
```

```
At PC = 0x11c: ADDU r2,r3,r2
At PC = 0x120: JAL 4
At PC = 0x124: SW r2,24(r30)
At PC = 0x10: ADDIU r2,r0,0
At PC = 0x14: SYSCALL
Exception: syscall
Machine halting!
```

可以发现，Nachos 生成 Nachos 可执行程序的过程，实际上是将 c++ 程序编译后的 coff 对象文件再做一次转换，将 coff 文件在特定的格式上转换成 noff 文件，然后生成相应的汇编代码，再在 Nachos 模拟的处理机上运行。实际上，Nachos 的汇编和 g++ 生成的汇编是差不多的。

- **Nachos 对内存、寄存器以及CPU的模拟**

Nachos 机器模拟很重要的部分是内存和寄存器的模拟。**Nachos寄存器组模拟了全部32个MIPS机(R2/3000)的寄存器，同时加上有关 Nachos 系统调试用的 8 个寄存器**，以期让模拟更加真实化并易于调试。

Nachos 用宿主机的一块内存模拟自己的内存。为了简便起见，每个内存页的大小同磁盘扇区的大小相同，而整个内存的大小远远小于模拟磁盘的大小。由于 Nachos 是一个教学操作系统，在内存分配上和实际的操作系统是有区别的。事实上，Nachos 的内存只有当需要执行用户程序时用户程序的一个暂存地，而作为操作系统内部使用的数据结构不存放在 Nachos 的模拟内存中，而是申请 Nachos 所在宿主机的内存。所以 Nachos的一些重要的数据结构如线程控制结构等的数目可以是无限的，不受 Nachos 模拟内存大小的限制。

在用户程序运行过程中，会有很多系统陷入核心的情况。系统陷入有两大类原因：**进行系统调用陷入和系统出错陷入**。系统调用陷入在用户程序进行系统调用时发生。系统调用可以看作是软件指令，它们有效地弥补了机器硬件指令不足；系统出错陷入在系统发生错误时发生，比如用户程序使用了非法指令以及用户程序逻辑地址同实际的物理地址映射出错等情况。不同的出错陷入会有不同的处理，比如用户程序逻辑地址映射出错会起页面的重新调入，而用户程序使用了非法指令则需要向用户报告等等。Nachos 处理的陷入有：

```
SyscallException 系统调用陷入
PageFaultException 页面转换出错
ReadOnlyException 试图访问只读页面
BusErrorException 总线错，转换用户程序页面时出错
AddressErrorException 页面访问没有对齐，或者超出了页面的大小
OverflowException 加减法时整数溢出
IllegalInstruException 非法指令访问
```

模拟机的机器指令由操作代码和操作数组成的，其类定义和实现如下所示：

```
class Instruction {
public:
    void Decode(); // 将指令的二进制表示转换成系统方便处理的表示
    unsigned int value; // 指令的二进制表示
    char opCode; // 分析出的操作代码
    char rs, rt, rd; // 分析出的指令的三个寄存器的值
    int extra; // 分析出的指令立即数
};
```

Machine 类的定义如下所示：

```
class Machine {
public:
    Machine(bool debug); // 初始化方法
```

```

~Machine(); // 析构方法
void Run(); // 运行一个用户程序
int ReadRegister(int num); // 读出寄存器中的内容
void WriteRegister(int num, int value); // 向一个寄存器赋值
void OneInstruction(Instruction *instr); // 执行一个用户程序指令
void DelayedLoad(int nextReg, int nextVal); // 执行一次延迟载入
bool ReadMem(int addr, int size, int* value); // 读出内存 addr 地址中的内容
bool WriteMem(int addr, int size, int value); // 向内存 addr 地址中写入内容
ExceptionType Translate(int virtAddr, int* physAddr, int size, bool
writing);
// 将用户程序逻辑转换成物理地址
void RaiseException(ExceptionType which, int badvAddr);
// 执行出错陷入处理程序
void Debugger(); // 调用用户程序调试器
void DumpState(); // 打印机器寄存器和内存状态
char *mainMemory; // 模拟内存
int registers[NumTotalRegs]; // CPU 寄存器模拟
TranslationEntry *tlb; // TLB 页面转换表
TranslationEntry *pageTable; // 线性页面转换表
unsigned int pageTableSize; // 线性页面转换表大小
private:
bool singleStep; // 单步执行标志
int runUntilTime; // 调试时钟
};

```

需要注意的是，虽然这里的很多方法和属性规定为 public 的，但是它们只能在系统核心内被调用。定义 Machine 类的目的是为了执行用户程序，如同许多其它系统一样，用户程序不直接使用内存的物理地址，而是使用自己的逻辑地址，在用户程序逻辑地址和实际物理地址之间，就需要一次转换，系统提供了两种转换方法的接口：**线性页面地址转换方法**和**TLB 页面地址转换方法**

- 无论是线性页面地址转换还是 TLB 页面地址转换，都需要一张地址转换表，地址转换表是由若干个表项（Entry）组成的。每个表项记录程序逻辑页到内存实页的映射关系，和实页的使用状态、访问权限信息。类 TranslationEntry 描述了表项的结构：

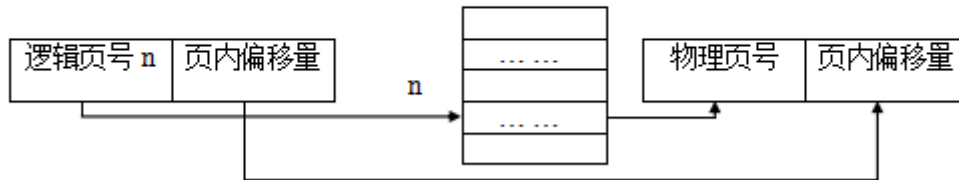
```

class TranslationEntry {
public :
    int virtualPage; // 逻辑页号
    int physicalPage; // 内存物理页号
    bool valid; // 该 Entry 是否使用，TRUE 表示在使用
    bool readOnly; // 对应页的访问属性，TRUE 示只读，否则为读写
    bool use; // 该 Entry 是否被使用过，每次访问后置为 TRUE
    bool dirty; // 对应的物理页使用情况，TRUE 表示被写过
}

```

**线性页面地址转换**是一种较为简单的方式，即用户程序的逻辑地址同实际物理地址之间的关系是线性的。在作转换时，给出逻辑地址，计算出其所在的逻辑页号和页中偏移量，通过查询转换表（实际上在使用线性页面地址转换时，TranslationEntry 结构中的 virtualPage 是多余的，线性页面转换表的下标就是逻辑页号），即可以得到实际物理页号和其页中偏移量。在模拟机上保存有线性页面转换表，它记录的是当前运行用户程序的页面转换关系；在用户进程空间中，也需要保存线性页面转换表，保存有自己运行用户程序的页面转换关系。当其被切换上模拟处理机上运行时，需要将进程的线性页面转换表覆盖模拟处理机的线性页面转换表。

线性页面转换表



**TLB 页面转换**则不同，TLB 转换页表是硬件来实现的，表的大小一般较实际的用户程序所占的页数要小，所以一般 TLB 表中只存放一部分逻辑页到物理页的转换关系。这样就可能出现逻辑地址转换失败的现象，会发生 PageFaultException 异常。在该异常处理程序中，就需要借助用户进程空间的线性页面转换表来计算出物理页，同时 TLB 表中增加一项。如果 TLB 表已满，就需要对 TLB 表项做 LRU 替换。使用 TLB 页面转换表处理起来逻辑较线性表为复杂，但是速度要快得多。由于 TLB 转换页表是硬件实现的，所以指向 TLB 转换页表的指针应该是只读的，所以 Machine 类一旦实例化，TLB 指针值不能改动。

在实际的系统中，线性页面地址转换和 TLB 页面地址转换只能二者取一，为简便起见，**Nachos 选择了前者**。另外，由于 Nachos 可以在多种平台上运行，各种运行平台的数据表达方式不完全一样，有的是高位在前，有的是高位在后。为了解决跨平台的问题，特地给出了四个函数作数据转换，它们是 WordToHost、ShortToHost、WordToMachine 和 ShortToMachine。

### 3. 阅读../test/Makefile，掌握如何利用交叉编译生成 Nachos 的可执行程序；

- 分析析../test/Makefile 可以看出，首先利用交叉编译器提供的 gcc、as、ld 等工具将 nachos 应用程序 .c 编译链接成 .coff 文件，然后利用 ../bin/arch/unknown-i386-linux/bin/coff2noff 将 coff 文件转变成 noff 文件：

```
.....
$(all_coff): $(obj_dir)/%.coff: $(obj_dir)/start.o $(obj_dir)/%.o
    @echo ">>> Linking" $(obj_dir)/$(notdir $@) "<<<"
    $(LD) $(LDFLAGS) $^ -o $(obj_dir)/$(notdir $@)

$(all_noff): $(bin_dir)/%.noff: $(obj_dir)/%.coff
    @echo ">>> Converting to noff file:" $@ "<<<"
    $(coff2noff) $^ $@
    ln -sf $@ $(notdir $@)
.....
```

### 4. 阅读../threads/main.cc, ../userprog/progtest.cc，根据对命令行参数-x 的处理过程，理解系统如何为应用程序创建进程，并启动进程的

- 在 main.cc 中，检测到命令行参数 -x 后，程序会调用 StartProcess(\*(argv + 1)) 过程：

```
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

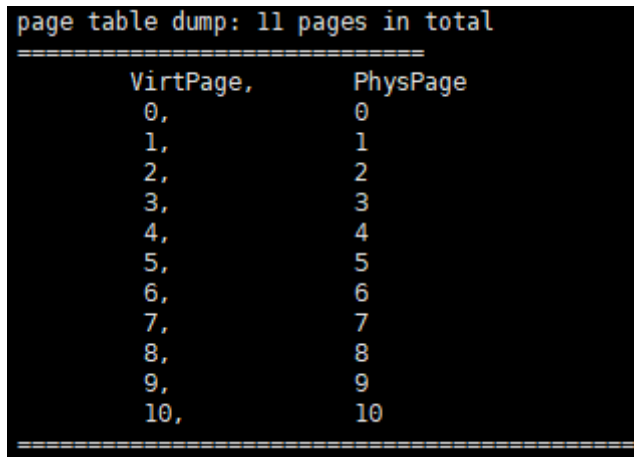
space->InitRegisters();      // set the initial register values
space->RestoreState();       // load page table register

machine->Run();              // jump to the user program
ASSERT(FALSE);              // machine->Run never returns;
}

```

在本函数中，Nachos 会根据输入的文件名以打开文件结构的方式打开该文件，然后调用 `space = new AddrSpace(executable);` 为其分配地址空间，把当前线程的空间赋值给该地址空间，然后关闭打开文件，初始化相应的地址空间和寄存器空间后，调用处理机模拟例程 `machine` 来执行该用户程序。

在类 `AddrSpace` 中添加成员函数 `Print()`，在为一个应用程序新建一个地址空间后调用该函数，输出该程序的页表（页面与帧的映射关系），显示信息有助于后续程序的调试与开发。



```

page table dump: 11 pages in total
=====
VirtPage,    PhysPage
0,           0
1,           1
2,           2
3,           3
4,           4
5,           5
6,           6
7,           7
8,           8
9,           9
10,          10
=====

```

- 在类 `AddrSpace` 中，实际上是做了用户程序地址空间到内核地址空间的映射过程，包括页表的映射等，实际上并不保存用户进程的信息，只需要在运行用户程序的线程中保存相应的映射结果然后运行即可。Nachos 的参数 `-x` (`nachos -x filename`) 调用 `../userprog/ proptest.cc` 的 `StartProcess(char *filename)` 函数，为用户程序创建 `filename` 创建相应的进程，并启动该进程的执行。
- 系统要运行一个应用程序，需要为该程序创建一个用户进程，为程序分配内存空间，将用户程序（代码段与数据段，数据段包括初始化的全局变量与未初始化的全局变量，以及静态变量）装入所分配的内存空间，创建相应的页表，建立虚页与实页（帧）的映射关系；（`AddressSpace::AddressSpace()`）。然后将用户进程映射到一个核心线程；（`StartProcess()` in `proptest.cc`）
- 为使该核心线程能够执行用户进程的代码，需要核心在调度执行该线程时，根据用户进程的页表读取用户进程指令；因此需要将用户页表首地址传递给核心的地址变换机构；（`machine.h` 中维护一个 `pageTable` 指针，指向当前正在运行的 Nachos 应用进程的页表）

##### 5. 阅读 `../userprog/ proptest.cc`, `../threads/scheduler.cc` (`Run()`)，理解如何将用户线程映射到核心线程，以及核心线程执行用户程序的原理与方法

- Nachos 的用户进程由两部分组成：核心部分和用户程序部分。核心部分同一般的系统线程没有区别，它共用了 Nachos 的正文段和数据段，运行在宿主主机上；而用户程序部分则有自己的正文段、数据段和栈段，它存储在 Nachos 的模拟内存中，运行在 Nachos 的模拟机上。在控制结构上，Nachos 的用户进程比系统线程多了以下内容：



```

int userRegisters[NumTotalRegs];    // 虚拟机的寄存器组
void SaveUserState();               // 线程切换时保存虚拟机寄存器组
void RestoreUserState();            // 线程切换时恢复虚拟机寄存器组
AddrSpace *space;                   // 线程运行的用户程序

```

其中，用户程序空间有AddrSpace类来描述：

```

class AddrSpace {
public:
    AddrSpace(OpenFile *executable);    // 根据可执行文件构成用户程序空间
    ~AddrSpace();                       // 析构方法
    void InitRegisters();               // 初始化模拟机的寄存器组
    void SaveState();                   // 保存当前机器页表状态
    void RestoreState();                // 恢复机器页表状态
private:
    TranslationEntry *pageTable;        // 用户程序页表
    unsigned int numPages;              // 用户程序的虚页数
};

```

在Linux系统中，使用gcc交叉编译技术将C程序编译成R2/3000可以执行的目标代码，通过Nachos提供的**coff2noff**工具将其转换成Nachos可以识别的可执行代码格式，拷贝到Nachos的文件系统中才能执行。

- 生成方法：

语 法	<b>AddrSpace (OpenFile *executable)</b>
参 数	Executable: 需要执行代码的打开文件结构
功 能	初始化用户程序空间。
实 现	1. 判断打开文件是否符合可执行代码的格式，如果不符合，出错返回 2. 将用户程序的正文段、数据段以及栈段一起考虑，计算需要空间大小。如果大于整个模拟的物理内存空间，出错返回。 3. 生成用户程序线性页表。 4. 将用户程序的正文段和数据段依次调入内存，栈段记录的是用户程序的运行状态，它的位置紧接于数据段之后。
返 回	无。

- InitRegisters方法

语法:	<b>Void InitRegisters ()</b>
参数:	无。
功能:	初始化寄存器，让用户程序处于可以运行状态。
实现:	设置PC指针、栈指针的初值，并将其它寄存器的值设置为0。同时，设置NextPCReg寄存器的值为4，方便PC得到下地址运行。
返回:	无。

- SaveState方法

语法:	<b>Void SaveState ()</b>
参数:	无。
功能:	存储用户程序空间的状态。
实现:	目前为空。
返回:	无。

- RestoreState方法

语法:	<b>Void RestoreState ()</b>
参数:	无。
功能:	恢复处理机用户程序空间的状态。
实现:	赋值实现。
返回:	无。

目前Nachos在运行用户程序时，有如下的限制：

- (1) 系统一次只能有运行一个用户程序，所以目前的线性转换页表比较简单，虚拟页号同物理页号完全一样。
- (2) 系统能够运行的用户程序大小是有限制的，必须小于模拟的物理内存空间大小，否则出错。在虚拟内存实现以后，这部分内容也将做改动。

**当用户程序空间初始化之后（设由space指针指向），真正的启动运行过程如下：**

```
space -> InitRegisters();           // 初始化模拟机寄存器组
space -> RestoreState();           // 恢复处理机用户程序空间的状态，实际上是将用户程序空间的
转换页表覆盖                               // 模拟机的转换页表
machine -> Run();                   // 运行用户程序
```

6. 阅读../userprog/ progtest.cc, ../machine/translate.cc, 理解当前进程的页表是如何与 CPU 使用的页表进行关联的;

- 在 AddrSpace 类的构造函数中, 有如下操作:

```
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;
}
```

它为用户程序的每一页都分配了相应的虚拟页号到物理页号的映射。在文件 translate.cc 中, 类 TranslationEntry 定义如下:

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

二者的关系就是如此构造起来的。Nachos 为用户程序分配地址空间, 地址空间中调用 TranslationEntry 为程序分配页表。再看程序 Machine::Translate(int virtAddr, int\* physAddr, int size, bool writing), 它涉及到物理页和虚拟页的转换。