山东大学 计算机科学与技术 学院

操作系统 课程实验报告

学号：202200101007	姓名：张祎乾	班级：22.3 班
实验题目：实验 5：扩展 Nachos 的文件系统		
实验学时：2	实验日期：2025/2/23	
实验目的： 为后续实验中实现系统调用 Exec()与 Exit()奠定基础 理解 Nachos 可执行文件的格式与结构； 掌握 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； 掌握如何利用交叉编译生成 Nachos 的可执行程序； 理解系统如何为应用程序创建进程，并启动进程； 理解如何将用户线程映射到核心线程，核心线程执行用户程序的原理与方法； 理解当前进程的页表是如何与 CPU 使用的页表进行关联的；		
实验环境：WSL、Ubuntu		

7.3 Nachos 应用程序与应用程序的加载

总结信息：

$$Nachos\text{应用程序}\xrightarrow[\text{编译}]{gcc\ MIPS\text{交叉编译器}}COFF\text{可执行文件}\xrightarrow[\text{转换}]{../test/coff2noff}NOFF\text{可执行文件}$$

Nachos 应用程序(.c 文件)放于../test 目录
../test$ make
../userprog$ make
./nachos -x halt.noff 可让 Nachos 运行应用程序 halt.noff
运行 Nachos 应用程序的方法：
(1)在../test 目录下运行 make，将该目下的几个现有的 Nachos 应用程序（.c 文件）交叉编译，并转换成 Nachos 可执行的.noff 格式文件。 现有的几个应用程序：halt.c，matmult.c，shell.c，sort.c
(2)在../userprog 目录下运行 make 编译生成 Nachos 系统，键入命令./nachos -x ../test/halt.noff 可让 Nachos 运行应用程序 halt.noff，参数-x 的作用是 Nachos 运行其应用程序。
(3)nachos -d m -x ../test/halt.noff
加上参数 -d m 输出显示 Nachos 模拟的 MIPS CPU 所执行的每条指令
nachos -d m -s -x ../test/halt.noff
可以再加上参数-s，以输出每条指令执行后对应寄存器的状态

7.3.1Nachos 应用程序与可执行程序

修改 test/halt.c

```
15    int
16    main()
17    {
18        /*注释代码 原main函数内容
19        char prompt[2];
20        prompt[0] = '-';
21        prompt[1] = '-';
22
23        Write(prompt, 1, "I will shut down!\n");
24        Halt();*/
25        //新增代码6行 新main函数内容
26        int i,j,k;
27        k=3;
28        i=2;
29        j=i-1;
30        k=i-j+k;
31        Halt();
32        /* not reached */
33    }
```

make

```
zhang@zhang:~/OS/nachos-3.4/code/test$ make
>>> Building dependency file for halt.c <<<
>>> Compiling halt.c <<<
/usr/local/mips/bin/decstation-ultrix-gcc  -G 0 -c -I../userprog -I../threads -c -o arc
h/unknown-i386-linux/objects/halt.o halt.c
>>> Linking arch/unknown-i386-linux/objects/halt.coff <<<
/usr/local/mips/bin/decstation-ultrix-ld -T script -N arch/unknown-i386-linux/objects/s
tart.o arch/unknown-i386-linux/objects/halt.o -o arch/unknown-i386-linux/objects/halt.c
off
>>> Converting to noff file: arch/unknown-i386-linux/bin/halt.noff <<<
../bin/arch/unknown-i386-linux/bin/coff2noff arch/unknown-i386-linux/objects/halt.coff
arch/unknown-i386-linux/bin/halt.noff
numsections 3
Loading 3 sections:
        ".text", filepos 0xd0, mempos 0x0, size 0x140
        ".data", filepos 0x210, mempos 0x140, size 0x0
        ".bss", filepos 0x0, mempos 0x140, size 0x0
ln -sf arch/unknown-i386-linux/bin/halt.noff halt.noff
>>> Converting to flat file: arch/unknown-i386-linux/bin/halt.flat <<<
../bin/arch/unknown-i386-linux/bin/coff2flat arch/unknown-i386-linux/objects/halt.coff
arch/unknown-i386-linux/bin/halt.flat
Loading 3 sections:
        ".text", filepos 0xd0, mempos 0x0, size 0x140
        ".data", filepos 0x210, mempos 0x140, size 0x0
        ".bss", filepos 0x0, mempos 0x140, size 0x0
Adding stack of size: 1024
ln -sf arch/unknown-i386-linux/bin/halt.flat halt.flat
```

/usr/local/mips/bin/decstation-ultrix-gcc -I ../userprog -I ../threads -S halt.c

halt.s 内容

OS > nachos-3.4 > code > test > 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  $2,16($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
```

运行 nachos -x ../test/halt 和 nachos -x ../test/halt.noff 命令得到如下图所示信息，说明../test 中指向../arch/unknown-i386-linux/bin/halt.noff 文件的符号链接文件是 halt.noff

```
zhang@zhang:~/OS/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
Unable to open file ../test/halt
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 10, idle 0, system 10, user 0
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...
zhang@zhang:~/OS/nachos-3.4/code/userprog$ ./nachos -x ../test/halt.noff
Machine halting!

Ticks: total 37, idle 0, system 10, user 27
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...
```

./nachos -d m -x ../test/halt.noff

```
zhang@zhang:~/OS/nachos-3.4/code/userprog$ ./nachos -d m -x ../test/halt.noff
Starting thread "main" at time 10
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!

Ticks: total 37, idle 0, system 10, user 27
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...
```

./nachos -d m -s -x ../test/halt.noff

```
zhang@zhang:~/OS/nachos-3.4/code/userprog$ ./nachos -d m -s -x ../test/halt.noff
Starting thread "main" at time 10
At PC = 0x0: JAL 52
Time: 11, interrupts on
Pending interrupts:
End of pending interrupts
Machine registers:
        0:      0x0     1:      0x0     2:      0x0     3:      0x0
        4:      0x0     5:      0x0     6:      0x0     7:      0x0
        8:      0x0     9:      0x0     10:     0x0     11:     0x0
        12:     0x0     13:     0x0     14:     0x0     15:     0x0
        16:     0x0     17:     0x0     18:     0x0     19:     0x0
        20:     0x0     21:     0x0     22:     0x0     23:     0x0
        24:     0x0     25:     0x0     26:     0x0     27:     0x0
        28:     0x0     SP(29): 0x570   30:     0x0     RA(31): 0x8
        Hi:     0x0     Lo:     0x0
        PC:     0x4     NextPC: 0xd0    PrevPC: 0x0
        Load:   0x0     LoadV:  0x0

11> n
At PC = 0x4: SLL r0,r0,0
Time: 12, interrupts on
Pending interrupts:
End of pending interrupts
Machine registers:
        0:      0x0     1:      0x0     2:      0x0     3:      0x0
        4:      0x0     5:      0x0     6:      0x0     7:      0x0
        8:      0x0     9:      0x0     10:     0x0     11:     0x0
        12:     0x0     13:     0x0     14:     0x0     15:     0x0
        16:     0x0     17:     0x0     18:     0x0     19:     0x0
        20:     0x0     21:     0x0     22:     0x0     23:     0x0
        24:     0x0     25:     0x0     26:     0x0     27:     0x0
        28:     0x0     SP(29): 0x570   30:     0x0     RA(31): 0x8
        Hi:     0x0     Lo:     0x0
        PC:     0xd0    NextPC: 0xd4    PrevPC: 0x4
        Load:   0x0     LoadV:  0x0

12>
At PC = 0xd0: ADDIU r29,r29,-40
Time: 13, interrupts on
Pending interrupts:
End of pending interrupts
Machine registers:
        0:      0x0     1:      0x0     2:      0x0     3:      0x0
        4:      0x0     5:      0x0     6:      0x0     7:      0x0
        8:      0x0     9:      0x0     10:     0x0     11:     0x0
        12:     0x0     13:     0x0     14:     0x0     15:     0x0
        16:     0x0     17:     0x0     18:     0x0     19:     0x0
        20:     0x0     21:     0x0     22:     0x0     23:     0x0
        24:     0x0     25:     0x0     26:     0x0     27:     0x0
```

7.3.2Nachos 可执行程序格式

可以发现 noff 可执行文件主要有两部分组成：头部份 noffHeader 和段部分 segment。
segment 包含虚拟地址 virtualAddr、文件中的地址 inFileAddr、段大小 size
noffHeader 包含 noff 标志 noffMageic、代码段 code、初始化数据段 initData、未初始化数据段 uninitData

```
#define NOFFMAGIC 0xbadfad  /* magic number denoting Nachos
                             * object code file
                             */

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;
```

7.3.3 页表的系统转储

修改 userprog/addrspace.h：

```
class AddrSpace {
  public:
    AddrSpace(OpenFile *executable);  // Create an address space,
                                      // initializing it with the program
                                      // stored in the file "executable"
    ~AddrSpace();     // De-allocate an address space

    void InitRegisters();   // Initialize user-level CPU registers,
                            // before jumping to user code

    void SaveState();     // Save/restore address space-specific
    void RestoreState();    // info on a context switch

    void Print();//新增代码 输出程序的页表

  private:
    TranslationEntry *pageTable;  // Assume linear page table translation
                                  // for now!
    unsigned int numPages;    // Number of pages in the virtual
                              // address space
};
```

修改 userprog/addrspace.cc：

```
185  //新增代码10行
186  void AddrSpace::Print()
187  {
188      printf("page table dump:  %d pages  in total\n", numPages);
189      printf("============================\n");
190      printf("\tVirtPage, \tPhysPage\n");
191      for (int i=0; i < numPages;  i++) {
192      printf("\t %d, \t\t%d\n", pageTable[i].virtualPage, pageTable[i].physicalPage);
193      }
194      printf("=========================================\n\n");
195  }
```

修改 userprog/progtest.cc

```
23   void
24   StartProcess(char *filename)
25   {
26       OpenFile *executable = fileSystem->Open(filename);
27       AddrSpace *space;
28
29       if (executable == NULL) {
30       printf("Unable to open file %s\n", filename);
31       return;
32       }
33       space = new AddrSpace(executable);
34       currentThread->space = space;
35       space->Print();//新增代码 输出该作业的页表信息
36       delete executable;          // close file
37
38       space->InitRegisters();     // set the initial register values
39       space->RestoreState();      // load page table register
40
41       machine->Run();         // jump to the user progam
42       ASSERT(FALSE);          // machine->Run never returns;
43                                // the address space exits
44                                // by doing the syscall "exit"
45   }
```

make clean

make

./nachos -x ../test/halt.noff

```
zhang@zhang:~/OS/nachos-3.4/code/userprog$ ./nachos -x ../test/halt.noff
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
========================================

Machine halting!

Ticks: total 37, idle 0, system 10, user 27
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...
```

7.3.4 应用程序进程的创建与启动

（1）理解 Nachos 为应用程序创建进程的过程;

在 Nachos 中，StartProcess 函数负责为应用程序创建一个进程。其主要步骤包括:

打开可执行文件：通过 fileSystem->Open(filename) 打开指定文件，加载可执行文件内容。

创建地址空间：通过 new AddrSpace(executable) 为进程分配地址空间，并初始化页表。

关联地址空间与线程：将地址空间绑定到当前线程（currentThread->space = space）。

初始化寄存器和页表：调用 space->InitRegisters() 初始化寄存器，调用 space->RestoreState() 加载页表到 CPU。

启动进程：调用 machine->Run() 开始执行用户程序。

（2）理解系统为用户进程分配内存空间、建立页表的过程，分析目前的处理方 法存在的问题?

内存分配与页表建立:

在 AddrSpace 的构造函数中，系统会为进程分配内存空间，并根据可执行文件的内容建立页表。

页表将虚拟地址映射到物理地址，确保进程可以访问自己的内存空间。

存在的问题:

内存管理简单：Nachos 的内存管理较为简单，可能缺乏高效的内存分配和回收机制。

页表大小固定：页表的大小可能是固定的，无法动态扩展，可能导致内存浪费或不足。

缺乏共享内存支持：没有实现共享内存机制，进程间无法高效共享数据。

缺乏保护机制：可能没有实现内存保护机制，进程可能意外访问其他进程的内存。

（3）理解应用进程如何映射到一个核心线程;

在 Nachos 中，应用进程通过 currentThread->space = space 将地址空间与核心线程绑定。

核心线程（Thread 类）是 Nachos 中调度的基本单位，每个线程可以关联一个地址空间（AddrSpace）。

当线程运行时，CPU 会使用该线程关联的地址空间的页表进行地址转换。
（4）如何启动主进程（父进程）；
主进程（父进程）通常是由 Nachos 内核直接启动的。
在 StartProcess 函数中，通过 machine->Run() 启动用户程序，这相当于启动主进程。
主进程的地址空间和页表会在 StartProcess 中初始化并加载到 CPU。
（5）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 space->RestoreState() 中，页表会被加载到 CPU 的页表寄存器（Page Table Register, PTR）。
CPU 在执行指令时，会根据 PTR 中的页表进行虚拟地址到物理地址的转换。
每次切换进程时，都需要调用 RestoreState() 更新 CPU 的页表。
（6）思考如何在父进程中创建子进程，实现多进程机制；
实现子进程创建：
复制地址空间：通过复制父进程的地址空间（AddrSpace）为子进程创建独立的地址空间。
创建新线程：创建一个新的核心线程（Thread），并将其地址空间设置为复制的地址空间。
初始化子进程：调用 space->InitRegisters() 初始化子进程的寄存器，设置程序计数器（PC）为子进程的入口点。
启动子进程：将子线程加入调度队列，等待调度执行。
多进程机制：
通过线程调度器（Scheduler）实现多进程的并发执行。
需要实现进程间通信（IPC）机制，如管道、共享内存等。
（7）思考进程退出要完成的工作有哪些？
进程退出时需要完成以下工作：
释放内存空间：释放进程的地址空间（AddrSpace）和页表。
关闭文件：关闭进程打开的文件，释放文件描述符。
释放线程资源：释放与进程关联的线程资源。
通知父进程：如果存在父进程，通知父进程子进程已退出（如通过 wait 机制）。
更新进程状态：将进程状态设置为“终止”，并从进程管理表中移除。
清理寄存器状态：清理 CPU 寄存器，确保不会影响其他进程的执行。
回收其他资源：如信号量、锁等资源。

7.3.5 分配更大的地址空间

修改 test/halt.c

```
13  #include "syscall.h"
14  static int a[40];//新增代码，分配更大的地址空间
15  int
16  main()
17  {
18      /*注释代码 原main函数内容
```

7.4

7.4.1Nachos 可执行程序的结构

可以发现 noff 可执行文件主要有两部分组成：头部份 noffHeader 和段部分 segment。
segment 包含虚拟地址 virtualAddr、文件中的地址 inFileAddr、段大小 size
noffHeader 包含 noff 标志 noffMageic、代码段 code、初始化数据段 initData、未初始化数据段 uninitData

```
#define NOFFMAGIC 0xbadfad  /* magic number denoting Nachos
                             * object code file
                             */

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;
```

7.4.2 页表

```
class TranslationEntry {
  public:
    int virtualPage;    // The page number in virtual memory.
    int physicalPage;   // The page number in real memory (relative to the
      //  start of "mainMemory"
    bool valid;         // If this bit is set, the translation is ignored.
      // (In other words, the entry hasn't been initialized.)
    bool readOnly;  // If this bit is set, the user program is not allowed
      // to modify the contents of the page.
    bool use;           // This bit is set by the hardware every time the
      // page is referenced or modified.
    bool dirty;         // This bit is set by the hardware every time the
      // page is modified.
};
```

可以看到变量如下：
虚拟页 virtualPage
物理页 physicalPage
有效位 valid

只读 readOnly
是否在使用 use
脏位 dirty

7.4.3 用户进程的创建与启动

1、Nachos 的参数-x（nachos -x filename）调用../userprog/progtest.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。

2、Instruction 类封装了一条 Nachos 机器指令

3、machine::ReadMem(registers[PCReg], 4, &raw))（在 translate.cc 中实现），实现读内存操作。

4、Machine::OneInstruction(Instruction *instr)（../machine/mipssim.cc 中实现），对通过 machine::ReadMem(registers[PCReg], 4, &raw))读出的指令进行译码并根据指令规定的操作执行该条指令；

5、machine::Run()（./machine/mipssim.cc 中实现）循环调用 Machine::OneInstruction(Instruction *instr)执行程序指令，直到程序退出或遇到一个异常；

6、AddrSpace::RestoreState() 将用户进程的页表传递给 Machine 类，该页表在为用户进程分配地址空间时创建

7、为便于上下文切换时保存与恢复寄存器状态，Nachos 设置了两种寄存器组：

①CPU 使用的寄存器 int registers[NumTotalRegs]（参见 Machine 类 in Machine.h），用于保存执行完一条机器指令时该指令的执行状态；

②核心线程运行用户程序时使用的用户寄存器 int userRegisters[NumTotalRegs]，用户保存执行完一条用户程序指令后的寄存器状态

8、Scheduler::Run()中核心进程切换时对 CPU 寄存器与用户寄存器的保存与恢复

修改 userprog/progetst.cc 中的 StartProcsss()(只加了注释):

```
23  void
24  StartProcess(char *filename)
25  {
26      OpenFile *executable = fileSystem->Open(filename);
27      AddrSpace *space;
28
29      if (executable == NULL) {
30      printf("Unable to open file %s\n", filename);
31      return;
32      }
33      space = new AddrSpace(executable);
34  // 为应用程序filename分配内存空间并将其装入所分配的内存空间中，然后建立页表，并建立虚页与实页（帧）的映射关系
35  // space 就是该进程的标识，而不是用pid来当进程标识
36      currentThread->space = space;// 将该进程映射到一个核心进程
37      space->Print();//新增代码 输出该作业的页表信息
38      delete executable;          // close file
39
40      space->InitRegisters();     // set the initial register values
41      // 初始化 CPU 的寄存器
42      space->RestoreState();      // load page table register
43
44      machine->Run();         // jump to the user progam
45      // 从程序入口开始，完成取指令、译码、执行的过程，直到进程遇到Exit()语句或者异常才退出
46      ASSERT(FALSE);          // machine->Run never returns;
47                              // the address space exits
48                              // by doing the syscall "exit"
49  }
```