Lab7 地址空间的扩展

1. 阅读 .../prog/protest.cc，深入理解 Nachos 创建应用程序进程的详细过程

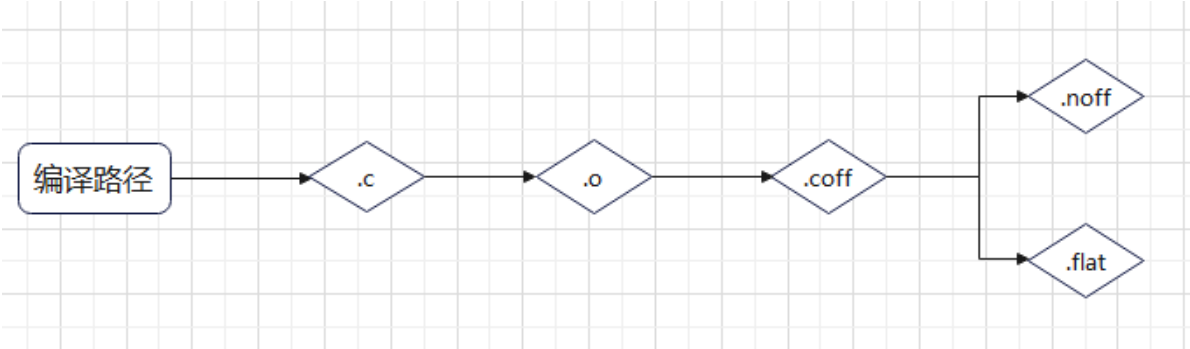
地址空间映射

观察 .../test/Makefile 文件可知，Nachos 利用了交叉编译器提供的 gcc、as、ld 工具用于编译。

```
CC = $(GCCDIR)gcc
AS = $(GCCDIR)as
LD = $(GCCDIR)ld
```

```
coff2noff = ../bin/$(real_bin_dir)/coff2noff
coff2flat = ../bin/$(real_bin_dir)/coff2flat
```

编译的目标文件是 .noff 文件和 .flat 文件，编译过程是先通过 .c 文件编译出 .o 文件，再通过 .o 文件编译出 .coff 文件，再通过 .coff 文件编译出 .noff 文件与 .flat 文件。



整个执行过程的流程图请访问链接查看：（图中黑色线段部分）

https://bucket011.obs.cn-north-4.myhuaweicloud.com:443/lab6-8.tiff?AccessKeyId=EPMCKIK9NRITQHB3EEVR&Expires=1682926038&Signature=S192G8/tf6%2B3OQMTHr3v%2BFihlTw%3D

其中 .noff 文件头结构如下所示：

```
#define NOFFMAGIC   0xbadfad      // Nachos 文件的魔数

typedef struct segment {
  int virtualAddr;              // 段在虚拟地址空间中的位置
  int inFileAddr;               // 段在文件中的位置
  int size;                     // 段大小
} Segment;

typedef struct noffHeader {
   int noffMagic;               // noff 文件的魔数
   Segment code;                // 可执行代码段
   Segment initData;            // 已初始化数据段
   Segment uninitData;          // 未初始化数据段，文件使用之前此数据段应为空
} NoffHeader;
```

我们可以继续观察 Addrspace 类的构造函数，观察 Nachos 是如何创建一个地址空间用于运行用户程序，以及如何打开 noff 文件进行文件空间计算：

```
class AddrSpace {
```

```
  private:
    TranslationEntry *pageTable;    // 线性页表（虚拟页-物理页）
    unsigned int numPages;           // 应用程序页数
};

AddrSpace::AddrSpace(OpenFile *executable) {
    // 可执行文件中包含了目标代码文件
    NoffHeader noffH;               // noff文件头
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);   // 读出noff文件
    if ((noffH.noffMagic != NOFFMAGIC) && (WordToHost(noffH.noffMagic) ==
NOFFMAGIC))
        SwapHeader(&noffH);           // 检查noff文件是否正确
    ASSERT(noffH.noffMagic == NOFFMAGIC);
    // 确定地址空间大小，其中还包括了用户栈大小
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size +
UserStackSize;
    numPages = divRoundUp(size, PageSize);  // 确定页数
    size = numPages * PageSize;             // 计算真实占用大小
    ASSERT(numPages <= NumPhysPages);       // 确认运行文件大小可以运行
    // 第一步，创建页表，并对每一页赋初值
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
      pageTable[i].virtualPage = i;         // 虚拟页
      pageTable[i].physicalPage = i;        // 虚拟页与物理页一一对应
      pageTable[i].valid = TRUE;
      pageTable[i].use = FALSE;
      pageTable[i].dirty = FALSE;
      pageTable[i].readOnly = FALSE;        // 只读选项
    }
    // 第二步，初始化物理内存
    bzero(machine->mainMemory, size);
    // 第三步，将noff文件数据拷贝到物理内存中
    // 拷贝程序段
    if (noffH.code.size > 0) {
      executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
        noffH.code.size, noffH.code.inFileAddr); // ReadAt调用了bcopy函数
    }
    // 拷贝数据段
    if (noffH.initData.size > 0) {
      executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
        noffH.initData.size, noffH.initData.inFileAddr);
    }
}
```

在 Nachos 中，页表实现了虚页与实页的对应关系，系统根据页表实现存储保护，页面置换算法根据页表信息进行页面置换。我们查看 .../machine/translate.h 文件得到如下页表项结构：

```
class TranslationEntry {
  public:
    int virtualPage;    // 虚拟内存中的页编号
    int physicalPage;   // 物理内存中的页编号（相对于主存位置）
    bool valid;         // valid = 1, 该转换有效（已经被初始化）
    bool readOnly;      // readOnly = 1, 该页内容不允许修改
    bool use;           // use = 1, 该页被引用或修改（变量由硬件修改）
    bool dirty;         // dirty = 1, 该页被修改（变量由硬件修改）
};
```

通过上述代码，我们可以发现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，将用户程序数据装入所分配的内存空间，并创建相应的页表，建立虚页与实页的映射关系。然后将用户进程映射到一个核心线程。

为了使核心线程能够执行用户进程指令，Nachos 根据用户进程的页表读取用户进程指令，并将用户页表传递给了核心线程的地址变换机构。

用户程序的创建与启动

1. **StartProcess 函数**

   查看 .../threads/main.cc 文件可以发现 Nachos 的参数 -x 调用了 .../userprog/progtest.cc 中的 StartProcess(char *filename); 函数：

   ```
   if (!strcmp(*argv, "-x")) {         // run a user program
       ASSERT(argc > 1);
       StartProcess(*(argv + 1));
       argCount = 2;
   }
   ```

   具体函数内容如下。由于下述文件中出现了打开文件的操作，因此我们查看 .../userprog/Makefile.local 文件可以发现在用户程序中的宏定义为 FILESYS_STUB，即并非使用实验四、五中的文件系统对 DISK 上的文件进行操作，而是直接对 UNIX 文件进行操作。

   ```
   void StartProcess(char *filename) {             // 传入文件名
       OpenFile *executable = fileSystem->Open(filename);  // 打开文件
       AddrSpace *space;                           // 定义地址空间

       if (executable == NULL) {
         printf("Unable to open file %s\n", filename); // 无法打开文件
         return;
       }
       space = new AddrSpace(executable);  // 初始化地址空间
       currentThread->space = space;       // 将用户进程映射到一个核心线程

       delete executable;          // 关闭文件
       space->InitRegisters();     // 设置 Machine 的寄存器初值
       space->RestoreState();      // 将应用程序页表加载到了Machine中
       machine->Run();             // machine->Run()代码中有死循环，不会返回
       ASSERT(FALSE);
   }

   #define StackReg 29             // 用户栈指针
   #define PCReg 34                // 当前存储PC值的寄存器
   #define NextPCReg 35            // 存储下一个PC值的寄存器
   ```

```
#define PrevPCReg 36            // 存储上一次PC值的寄存器
void AddrSpace::InitRegisters() {
    for (int i = 0; i < NumTotalRegs; i++)      // 每个寄存器初值置0
      machine->WriteRegister(i, 0);
    machine->WriteRegister(PCReg, 0);           // PC寄存器初值为0
    machine->WriteRegister(NextPCReg, 4);       // 下一个PC值为4
    // 栈指针赋初值，减去一个数值避免越界
    machine->WriteRegister(StackReg, numPages * PageSize - 16);
}

void AddrSpace::RestoreState() {
    machine->pageTable = pageTable;     // 将应用程序页表赋给Machine
    machine->pageTableSize = numPages;
}
```

2. **Instruction 类**

   Instruction 类封装了一条 Nachos 机器指令，具体信息如下：

```
class Instruction {
  public:
    void Decode();          // 解码二进制表示的指令
    unsigned int value;     // 指令的二进制表达形式
    char opCode;            // 指令类型
    char rs, rt, rd;        // 指令的 3 个寄存器
    int extra;              // 立即数（带符号） 或 目标 或 偏移量
};
```

3. **Machine::ReadMem() 函数**

   继续查看 Machine 类，可以看到将虚拟地址数据读取到实际地址的函数，Machine::ReadMem();，如下所示：

```
// 此函数将虚拟内存addr处的size字节的数据读取到value所指的物理内存中，读取错误则返回false。
bool Machine::ReadMem(int addr, int size, int *value) {     // 虚拟地址、读取字节数、物理地址
    int data, physicalAddress;
    ExceptionType exception;        // 异常类型
    // 进行虚实地址转换
    exception = Translate(addr, &physicalAddress, size, FALSE);
    if (exception != NoException) {
      machine->RaiseException(exception, addr);     // 抛出异常，返回false
      return FALSE;
    }
    switch (size) {         // 对字节大小进行分类处理
      case 1:               // 读取一个字节，放入value所指地址
        data = machine->mainMemory[physicalAddress];
        *value = data;
        break;
      case 2:               // 读取两个字节，即一个short类型
        data = *(unsigned short *) &machine->mainMemory[physicalAddress];
        *value = ShortToHost(data);         // 短字转为主机格式
        break;
      case 4:               // 读取四个字节，即一个int类型
        data = *(unsigned int *) &machine->mainMemory[physicalAddress];
```

```
        *value = WordToHost(data);          // 字转为主机格式
        break;
      default: ASSERT(FALSE);
    }
    return (TRUE);      // 读取正确
}
```

4. **Machine::OneInstruction() 函数**

   Nachos 将虚拟地址转化为物理地址后，从物理地址取出指令放入 Machine::OneInstruction(Instruction *instr) 函数进行执行，该函数具体代码如下所示：

```
#define PCReg 34                    // 当前存储PC值的寄存器
#define NextPCReg 35                // 存储下一个PC值的寄存器
#define PrevPCReg 36                // 存储上一次PC值的寄存器

// 执行一条用户态的指令。如果执行指令过程中有异常或中断发生，则调出异常处理装置，待其处理完成后再继续运行。
void Machine::OneInstruction(Instruction *instr) {
    int raw, nextLoadReg = 0, nextLoadValue = 0;    // nextLoadValue记录延迟的载入操作，用于之后执行
    //  读取指令数据到raw中
    if (!machine->ReadMem(registers[PCReg], 4, &raw)) return;     // 发生异常
    instr->value = raw; // 指令数据赋值
    instr->Decode();    // 指令解码
    int pcAfter = registers[NextPCReg] + 4; // 计算下下个PC指令地址
    int sum, diff, tmp, value;
    unsigned int rs, rt, imm;
    // 59条指令分类执行
    switch (instr->opCode) {
      case:     // 59个case
        ...
      default:
        ASSERT(FALSE);
    }
    // 执行被延迟的载入操作
    DelayedLoad(nextLoadReg, nextLoadValue);
    // 增加程序计数器（PC）
    registers[PrevPCReg] = registers[PCReg];    // 记录上一个PC值，用于之后调试
    registers[PCReg] = registers[NextPCReg];    // 将下一个PC值赋给NOW_PC寄存器
    registers[NextPCReg] = pcAfter;         // 将下下个PC值赋给NEXT_PC寄存器
}
```

5. **Machine::Run() 函数**

   Nachos 中调用了 Machine::Run() 函数循环调用上述 Machine::OneInstruction(Instruction *instr) 函数执行程序指令，具体函数代码如下所示：

```
// 模拟用户程序的执行，该函数不会返回
void Machine::Run() {
    Instruction *instr = new Instruction;   // 用于存储解码后的指令
    interrupt->setStatus(UserMode);          // 将中断状态设为用户模式
    for (;;) {
      OneInstruction(instr);                // 执行指令
      interrupt->OneTick();              // 用户模式下执行一条指令，时钟数为1
      // 单步调试
      if (singleStep && (runUntilTime <= stats->totalTicks))    Debugger();
    }
}
```

2. 阅读理解类 AddrSpace，然后对其进行修改，使 Nachos 能够支持多进程机制，允许 Nachos 同时运行多个用户进程

1. **在类 AddrSpace 中添加完善的 Print() 函数（在实验六中已给出）**

```
void AddrSpace::Print()
{
    printf("page table dump: %d pages in total\n", numPages);
    printf("============================\n");
    printf("\tVirtPage, \tPhysPage\n");
    for (int i=0; i < numPages; i++)
    {
        printf("\t %d, \t\t%d\n", pageTable[i].virtualPage,
pageTable[i].physicalPage);
    }
    printf("========================================\n\n");
}
```

2. **在类 AddrSpace 中实例化类 BitMap 的一个全局对象，用于管理空闲帧**

```
private:
    static BitMap *userMap;// 全局位图
```

3. **如果将 SpaceId 直接作为进程号 Pid 是否合适？如果不合适，应该如何为进程分配相应的 Pid？**

   不合适，应该动态地分配和管理进程Pid，在 AddrSpace 类中添加 spaceId，用于标识一个地址空间；userMap 用于分配物理页表；pidMap 用于分配 spaceId；Print() 用于输出该地址空间的页表。

```
private:
    static BitMap *userMap, *pidMap;// 全局位图
    TranslationEntry *pageTable;    // 线性页表
    unsigned int numPages, spaceId; // 页表中的页表项以及地址编号
```

4. **为了实现 Join(pid)，考虑如何在该进程相关联的核心线程中保存进程号**

将用户线程映射到核心线程的过程中，调用AddrSpace构造函数为其动态管理进程号，AddrSpace类的具体定义如下：

```
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();

    unsigned int getSpaceId() { return spaceId; }

#ifdef FILESYS
    OpenFile *fileDescriptor[UserProgramNum];   // 文件描述符，0、1、2分别为
stdin、stdout、stderr
    int getFileDescriptor(OpenFile *openfile);
    OpenFile *getFileId(int fd);
    void releaseFileDescriptor(int fd);
#endif

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

其构造函数的具体实现如下：

```
AddrSpace::AddrSpace(OpenFile *executable) {
    ASSERT(pidMap->NumClear() >= 1);  // 保证还有线程号可以分配
    spaceId = pidMap->Find() + 100;     // 0-99留给内核线程

    // 可执行文件中包含了目标代码文件

    NoffHeader noffH;               // noff文件头
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);   // 读出noff文件
    if ((noffH.noffMagic != NOFFMAGIC) && (WordToHost(noffH.noffMagic) ==
NOFFMAGIC))
        SwapHeader(&noffH);           // 检查noff文件是否正确
    ASSERT(noffH.noffMagic == NOFFMAGIC);
    // 确定地址空间大小，其中还包括了用户栈大小
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size +
UserStackSize;
    numPages = divRoundUp(size, PageSize);  // 确定页数
    size = numPages * PageSize;             // 计算真实占用大小
```

```
    ASSERT(numPages <= NumPhysPages);        // 确认运行文件大小可以运行

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
numPages, size);
    // 第一步，创建页表，并对每一页赋初值
    pageTable = new TranslationEntry[numPages];

    ASSERT(userMap->NumClear() >= numPages);    // 确认页面足够分配
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;         // 虚拟页
        pageTable[i].physicalPage = userMap->Find();  // 在位图找空闲页进行分配
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;        // 只读选项
    }
    // 第二步，将noff文件数据拷贝到物理内存中
    if (noffH.code.size > 0) {
        int pagePos =
pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize;
        int offSet = noffH.code.virtualAddr % PageSize;

        executable->ReadAt(&(machine->mainMemory[pagePos+offSet]),
                           noffH.code.size, noffH.code.inFileAddr); //
ReadAt调用了bcopy函数
    }
    if (noffH.initData.size > 0) {
        int pagePos =
pageTable[noffH.initData.virtualAddr/PageSize].physicalPage * PageSize;
        int offSet = noffH.initData.virtualAddr % PageSize;

        executable->ReadAt(&(machine->mainMemory[pagePos+offSet]),
                           noffH.initData.size, noffH.initData.inFileAddr);
    }
#ifdef FILESYS
    for(int i = 3; i < 10; i++) fileDescriptor[i] = NULL;
    OpenFile *StdinFile = new OpenFile("stdin");
    OpenFile *StdoutFile = new OpenFile("stdout");
    OpenFile *StderrFile = new OpenFile("stderr");
    /* 输出、输入、错误 */
    fileDescriptor[0] = StdinFile;
    fileDescriptor[1] = StdoutFile;
    fileDescriptor[2] = StderrFile;
#endif
}
```

获取地址空间ID的方法如下：

```
OpenFile* AddrSpace::getFileId(int fd) {
    ASSERT((fd >= 0) && (fd < UserProgramNum));
    return fileDescriptor[fd];
}
```

在将用户线程映射到核心线程的过程中，应该为用户线程记录好分配的地址空间的ID号，具体方法如下：

```
......

AddrSpace *space = new AddrSpace(executable);

delete executable;   // close file

// 创建一个核心线程，将该应用程序映射到该核心线程
char *forkedThreadName = filename;
Thread *thread = new Thread(filename);
thread->Fork(StartProcess, space->getSpaceId());
// 用户线程映射到核心线程
thread->space = space;
DEBUG('q', "filenale = %s, which address space is:\n", filename);
thread->space->Print();

......
```

5. **根据进程创建时系统为其所做的工作，考虑进程退出时应该做哪些工作**

   线程退出时，应该将线程状态设置为终止态，然后将其放入终止队列中，以便之后Join系统调用的实现。同时，应该将子线程相关的寄存器清零，然后恢复寄存器上下文，处理完系统调用后，应该步进PC寄存器，然后将父线程加入到调度队列中进行调度。

6. **考虑系统调用 Exec() 与 Exit() 的设计实现方案**

   Exec()系统调用实现方案：

   1、修改exception.cc，根据系统调用类型对各系统调用进行处理；
   (1) 从2号寄存器中获取当前的系统调用号(type=machine->ReadRegister(2)), 根据type对系统调用分别处理；

   (2) 获取系统调用参数（寄存器4、5、6、7，可以携带4个参数）
   在Exec(char *filename)的处理代码中：
   (a) 从第4号寄存器中获取Exec()的参数filename在内存中的地址（addr=machine->ReadRegister(4)）；
   (b) 利用Machine::ReadMem()从该地址读取应用程序文件名filename；
   (c) 打开该应用程序（OpenFile *executable = fileSystem->Open(filename)）；
   (d) 为其分配内存空间、创建页表、分配pid（space = new AddrSpace(executable)），至此为应用程序创建了一个进程；

   - 需要修改AddrSpace:: AddrSpace()实现上述功能；同时，需要修改AddrSpace:: ~AddrSpace()，进程退出时释放pid，为其所分配的帧也应释放（修改空闲帧位示图）；
   - 因此需要修改AddrSpace:: AddrSpace()与AddrSpace:: ~AddrSpace()；

   (e) 创建一个核心线程，并将该进程与新建的核心线程关联（thread = new Thread(forkedThreadName)，thread->Fork(StartProcess, space->getSpaceID())）；

   - 需要特别指出的是，通过Thread::Fork()创建的线程需要指明该线程要执行的代码（函数）及函数所需的参数；
   - 我们可以重载函数StartProcess(int spaceId)，作为新建线程执行的代码，并将进程的pid传递给系统，供其它系统调用（如Join()）使用；
   - 当调度到该线程时，就启动应用程序进程的执行；

   2、修改progtest.cc，重载函数StatProcess(char *filename)

将该函数作为应用程序进程所关联的核心线程的执行代码，当调度到该线程时，Exec(filename)中filename所对应的应用程序进程随即执行；

```
void StartProcess(int spaceId)
{
    space->InitRegisters(); // set the initial register values
    space->RestoreState();      // load page table register

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

3、修改AddrSpace::AddrSpace()及AddrSpace::~AddrSpace()

- 为管理空闲帧，建立一个全局的空闲帧管理位示图；
- 为管理pid（spaceId），建立一个全局的pid数组；
- 从内存的第一个空闲帧为Exec(filename)中的filename分配内存空间，创建该进程页表，建立虚实页表的映射关系，分配pid；
- 在释放应用程序内存空间时，应该清除空闲帧位示图相应的标志，释放pid，释放页表。
- AddrSpace::AddrSpace()及AddrSpace::~AddrSpace()的代码参考如下：

```
AddrSpace::AddrSpace(OpenFile *executable) {
    ASSERT(pidMap->NumClear() >= 1);  // 保证还有线程号可以分配
    spaceId = pidMap->Find() + 100;     // 0-99留给内核线程

    // 可执行文件中包含了目标代码文件

    NoffHeader noffH;               // noff文件头
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);   // 读出noff文
件
    if ((noffH.noffMagic != NOFFMAGIC) && (WordToHost(noffH.noffMagic)
== NOFFMAGIC))
        SwapHeader(&noffH);           // 检查noff文件是否正确
    ASSERT(noffH.noffMagic == NOFFMAGIC);
    // 确定地址空间大小，其中还包括了用户栈大小
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
+ UserStackSize;
    numPages = divRoundUp(size, PageSize);  // 确定页数
    size = numPages * PageSize;             // 计算真实占用大小
    ASSERT(numPages <= NumPhysPages);       // 确认运行文件大小可以运行

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
numPages, size);
    // 第一步，创建页表，并对每一页赋初值
    pageTable = new TranslationEntry[numPages];

    ASSERT(userMap->NumClear() >= numPages);    // 确认页面足够分配
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;         // 虚拟页
        pageTable[i].physicalPage = userMap->Find();  // 在位图找空闲页进行
分配
```

```
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;        // 只读选项
    }
    // 第二步，将noff文件数据拷贝到物理内存中
    if (noffH.code.size > 0) {
        int pagePos =
pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize;
        int offSet = noffH.code.virtualAddr % PageSize;

        executable->ReadAt(&(machine->mainMemory[pagePos+offSet]),
                           noffH.code.size, noffH.code.inFileAddr); //
ReadAt调用了bcopy函数
    }
    if (noffH.initData.size > 0) {
        int pagePos =
pageTable[noffH.initData.virtualAddr/PageSize].physicalPage * PageSize;
        int offSet = noffH.initData.virtualAddr % PageSize;

        executable->ReadAt(&(machine->mainMemory[pagePos+offSet]),
                           noffH.initData.size,
noffH.initData.inFileAddr);
    }
#ifdef FILESYS
    for(int i = 3; i < 10; i++) fileDescriptor[i] = NULL;
    OpenFile *StdinFile = new OpenFile("stdin");
    OpenFile *StdoutFile = new OpenFile("stdout");
    OpenFile *StderrFile = new OpenFile("stderr");
    /* 输出、输入、错误 */
    fileDescriptor[0] = StdinFile;
    fileDescriptor[1] = StdoutFile;
    fileDescriptor[2] = StderrFile;
#endif
}
```

Exit()系统调用实现方案：

- 系统调用void Exit(int status)的参数status是用户程序的退出状态。系统调用int Join(SpaceId id)需要返回该退出状态status。由于可能在id结束之后，其它程序（如parent）才调用Join(SpaceId id)，因此在id执行Exit(status)退出时需要将id的退出码ststus保存起来，以备Join()使用。
- 关于系统调用Exit()的实现，首先从4号寄存器读出退出码，然后释放该进程的内存空间及其其表，释放分配给该进程的实页（帧），释放其pid（参见AddrSpace::~AddrSpace()），调用currentThread->Finish结束该进程对应的线程。
- 管理空闲帧的位示图以及pid结构不能释放，因为它们是全局的。
- 实现代码形如：

```
case SC_Exit:{
   int ExitStatus=machine->ReadRegister(4);
   machine->WriteRegister(2,ExitStatus);
   currentThread->setExitStatus(ExitStatus);
if (ExitStatus == 99)  //parent process exit, delele all terminated
threads
```

```
{
   List *terminatedList = scheduler->getTerminatedList();
   scheduler->emptyList(terminatedList);
}
delete currentThread->space;
   currentThread->Finish();
   AdvancePC();
   break;
}
```

- 同时，要借助Thread类实现Join系统调用。