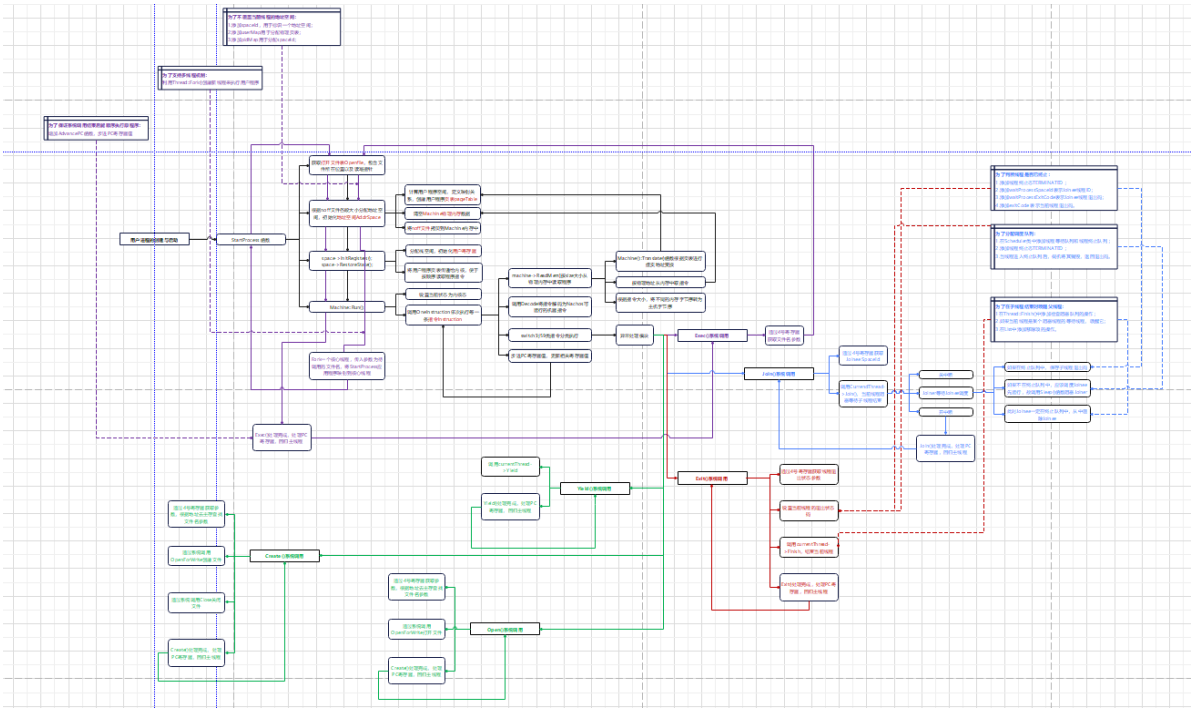
Lab8 系统调用 Exec()与 Exit()

程序总体结构图：



或通过访问链接查看：

https://bucket011.obs.cn-north-4.myhuaweicloud.com:443/lab6-8.tiff?AccessKeyId=EPMCKIK9NRITQHB3EEVR&Expires=1682928392&Signature=RExX9DS4hNPNM%2BSM3TiC8DY2Oz0%3D

1. **阅读 `../userprog/exception.cc`，理解 `Halt` 的实现原理：**

   Nachos 系统调用对应的宏在 `../userprog/syscall.h` 中声明如下：

   ```
   #define SC_Halt     0
   #define SC_Exit     1
   #define SC_Exec     2
   #define SC_Join     3
   #define SC_Create   4
   #define SC_Open     5
   #define SC_Read     6
   #define SC_Write    7
   #define SC_Close    8
   #define SC_Fork     9
   #define SC_Yield    10
   ```

   Nachos 目前仅实现了系统调用 `Halt()`， 其实现代码见 `../userprog/exception.cc` 中的函数 `void ExceptionHandler(ExceptionType which)`，其余的几个系统调用都没有实现。该函数实现如下：

```
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);
    }
}
```

- 系统调用如何执行

  从 `../machine/machine.cc` 及 `mipssim.cc` 中的实现可以看出，每一条用户程序中的指令在虚拟机中被读取后，被包装成一个 `OneInstruction` 对象，然后在 `mipssim.cc` 中调用 `Machine::OneInstruction(Instruction *instr)` 对其解码执行。

  具体而言，在 `../threads/main.c` 中，当读取到系统调用请求后，函数执行的方法如下：

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

  函数首先会调用过程 `StartProcess`，其定义在文件 `../lab7-8/progtest.cc` 中， 如下：

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

      delete executable;

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

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

  它首先为将要运行的用户程序分配地址空间，然后将用户程序的地址空间映射到当前核心线程的地址空间，将核心线程的地址空间映射到寄存器中，然后调用 `machine->Run();` 模拟核心态执行用户程序的操作。其中，`machine-Run()` 的具体实现在 `../machine/mipssim.cc` 中，其具体实现如下：

```
void Machine::Run()
{
    Instruction *instr = new Instruction;

    if(DebugIsEnabled('m')) {
        printf("Starting thread \"%s\" at time %d\n", currentThread-
>getName(), stats->totalTicks);
    }
    interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        interrupt->OneTick();
        if (singleStep && (runUntilTime <= stats->totalTicks))
            Debugger();
    }
}
```

语句 `interrupt->setStatus(UserMode);` 模拟用户态和核心态切换的过程，由于当前是核心态，需要执行用户态的程序，所以把机器寄存器状态设置为用户程序寄存器状态的值。

在 `for` 循环中不断调用 `Machine::OneInstruction(Instruction *instr)` 对用户态的程序进行解码并执行。`Machine::OneInstruction(Instruction *instr)` 中有一个非常大的 `SWITCH` 语句，该语句分析所取出的指令类型执行这条指令。

`switch` 语句对 `Nachos` 系统调用的处理方法是当 `Nachos` 的 `CPU` 检测到该条指令是执行一个 `Nachos` 的系统调用，则抛出一个异常 `SyscallException` 以便从用户态陷入到核心态去处理这个系统调用，代码如下：

```
case OP_SYSCALL:
    RaiseException(SyscallException, 0);
```

该异常在 `../userprog/exception.cc` 中进行处理，代码如下：

```
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);
    }
}
```

从上述代码中可以看出，系统将系统调用号保存在 `MIPS` 的 `2` 号寄存器 `$2` 中，语句 `type = machine->ReadRegister(2)` 从寄存器 `$2` 中获取系统调用号，如果该条指令要调用 `0` 号系统调用（对应的宏是 `SC_Halt`），则执行 `Halt()` 系统调用的处理程序：

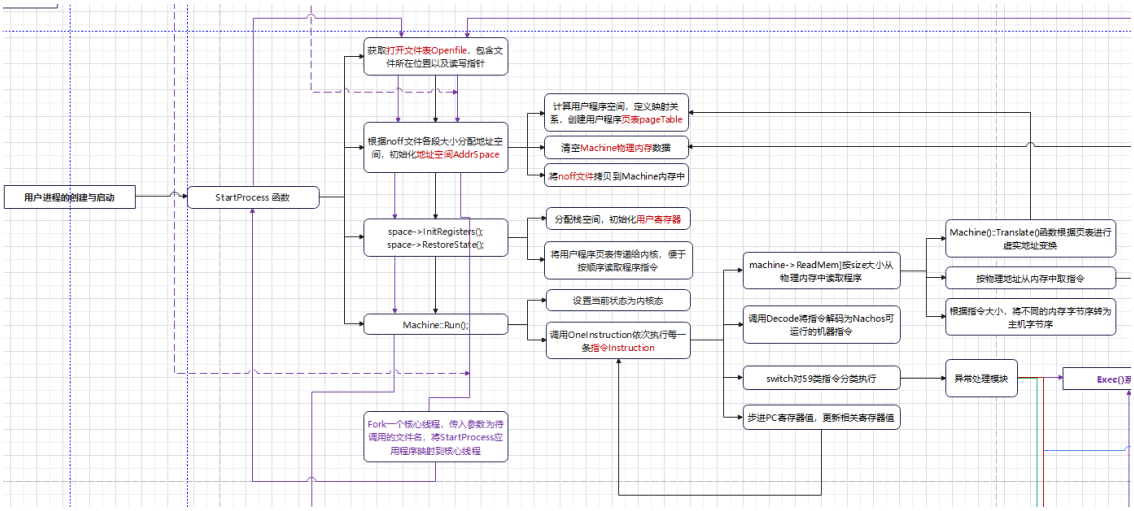
```
void
Interrupt::Halt()
{
    printf("Machine halting!\n\n");
    stats->Print();
    Cleanup();      // Never returns.
}
```

```
void
Cleanup()
{
    printf("\nCleaning up...\n");
#ifdef NETWORK
    delete postOffice;
#endif

#ifdef USER_PROGRAM
    delete machine;
#endif

#ifdef FILESYS_NEEDED
    delete fileSystem;
#endif

#ifdef FILESYS
    delete synchDisk;
#endif

    delete timer;
    delete scheduler;
    delete interrupt;

    Exit(0);
}
```

它清空所有态寄存器的状态，释放所有动态内存并退出程序，至此一个 `Halt` 系统调用被成功执行。`Nachos` 目前除了实现了系统调用 `Halt()`，其它的系统调用均未实现。

2. **基于实验 6、7 中所完成的工作，利用 Nachos 提供的文件管理、内存管理及线程管理等功能，编程实现系统调用 Exec() 与 Exit()（至少实现这两个）。**

- 用户程序的创建与启动过程（图中黑色部分）：



- 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。
  - 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);
}
```

  - 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。
- 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
};
```

- 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
}
```

- Machine()::Translate() 函数

  可以看到在 ReadMem() 函数中调用了 Translate() 函数进行了虚实地址转换，因此我们继续查看 Machine::Translate() 函数，如下所示。

```
/*  该函数主要功能是使用页表或TLB将虚拟地址转换为物理地址，并检查地址是否对齐以及其它错误。
    如果没有错误，则在页表项中设置use、dirty位初值，并将转换后的物理地址保存在physAddr变量中。
    virAddr - 虚拟地址，physAddr - 存储转换结果、size - 写或读的字节数
    writing - 可写标记，需要检查TLB中的"read-only"变量。 */
```

```
ExceptionType Machine::Translate(int virtAddr, int* physAddr, int size,
bool writing) {
    int i;
    unsigned int vpn, offset, pageFrame;
    TranslationEntry *entry;    // 页表项
    // 检查对齐错误，即如果size = 4，则地址为4的倍数，即低两位为0；
    // 若size = 2，则地址为2的倍数，即最低位为0
    if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr &
0x1)))
        return AddressErrorException;
    // TLB与页表必须有一个为空，有一个不为空
    ASSERT(tlb == NULL || pageTable == NULL);   // tlb、pageTable均定义在
Machine类中
    ASSERT(tlb != NULL || pageTable != NULL)    // 通过虚拟地址计算虚拟页编
号以及页内偏移量
    vpn = (unsigned) virtAddr / PageSize;       // 虚拟页编号
    offset = (unsigned) virtAddr % PageSize;    // 页内偏移量

    if (tlb == NULL) {          // TLB为空，则使用页表
      if (vpn >= pageTableSize)     // vpn大于页表大小，即返回地址错误
        return AddressErrorException;
      else if (!pageTable[vpn].valid)   // vpn所在页不可用，即返回页错误
        return PageFaultException;
      entry = &pageTable[vpn];           // 获得页表中该虚拟地址对应页表项
    } else {                                    // TLB不为空，则使用TLB
        for (entry = NULL, i = 0; i < TLBSize; i++)     // 遍历TLB搜索
            if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage ==
vpn)) {
            entry = &tlb[i];            // 找到虚拟地址所在页表项!
            break;
          }
        // 在TLB中没有找到，返回页错误
        if (entry == NULL) return PageFaultException;
    }
    // 想要向只读页写数据，返回只读错误
    if (entry->readOnly && writing) return ReadOnlyException;
    // 由页表项可得到物理页框号
    pageFrame = entry->physicalPage;
    // 物理页框号过大，返回越界错误
    if (pageFrame >= NumPhysPages) return BusErrorException;
    // 设置该页表项正在使用
    entry->use = TRUE;
    // 设置该页表项被修改了，即dirty位为true
    if (writing) entry->dirty = TRUE;
    // 得到物理地址
    *physAddr = pageFrame * PageSize + offset;
    // 物理地址不可越界
    ASSERT((*physAddr >= 0) && ((*physAddr + size) <= MemorySize));
    // 返回没有错误
    return NoException;
}
```

- 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
}
```

- 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();
    }
}
```

- AddrSpace::RestoreState() 函数

由上述执行过程中调用的函数代码可知，我们需要将用户进程的页表传递给 Machine 类维护的页表，才能执行用户程序指令。该过程由函数 AddrSpace::RestoreState() 实现，将用户进程的页表传递给 Machine 类，而用户进程的页表再为用户进程分配地址空间时就创建了。AddrSpace::RestoreState() 函数如下所示。

```
// 通过一次上下文切换，保存machine的状态使得地址空间得以运行
void AddrSpace::RestoreState() {
    machine->pageTable = pageTable;     // 页表项
    machine->pageTableSize = numPages;  // 页表大小
}
```

为了便于上下文切换时保存与恢复寄存器状态，Nachos 设置了两组寄存器，一组是 CPU 使用的寄存器 int registers[NumTotalRegs]，用于保存执行完一条机器指令时该指令的执行状态；另一组是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。

- Machine 类

  接下来我们通过查看 Machine 类来了解 CPU 使用的寄存器的定义，具体定义代码如下。

```
/*
    模拟主机工作硬件，包括CPU寄存器、主存等。
    用户程序无法分辨他们运行在模拟器上还是真实的硬件上，除非他们发现了该模拟器不支持浮点运算。
    模拟器有10条系统调用，但UNIX有200条系统调用。
*/
class Machine {
  public:
    Machine(bool debug);            // 模拟硬件的构造函数，用于运行用户程序
    ~Machine();                     // 析构函数
    // 运行用户程序的函数
    void Run();                     // 运行用户程序
    int ReadRegister(int num);  // 读取CPU寄存器中的内容
    void WriteRegister(int num, int value); // 保存value到num编号的CPU寄存器中
    // 模拟硬件的实现过程
    void OneInstruction(Instruction *instr);    // 执行一条用户程序指令
    void DelayedLoad(int nextReg, int nextVal); // 延迟加载
    bool ReadMem(int addr, int size, int* value); // 读取虚拟地址处的size个字节到value所指物理地址处
    bool WriteMem(int addr, int size, int value); // 将size个字节的value数据写入addr的虚拟地址处
    // 将虚拟地址转换为物理地址，并检查是否有异常
    ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);
    // 抛出异常，陷入系统态
    void RaiseException(ExceptionType which, int badVAddr);
    void Debugger();                // 调出用户程序调试器
    void DumpState();               // 打印出用户CPU和主存状态
    // 模拟硬件的数据结构
    char *mainMemory;               // 物理内存，用于存储用户程序、代码与数据
    int registers[NumTotalRegs];    // CPU寄存器，用于保存执行完机器指令时该指令执行状态
    // 虚、实地址转换（mainMemory首地址为0号地址）
    TranslationEntry *tlb;          // 快表，存在唯一，因此指针不可修改，类似于只读指针
    TranslationEntry *pageTable;    // 传统线性页表，可存在多个。
```

```
    unsigned int pageTableSize;     // 页表大小

  private:
    bool singleStep;                // 单步调试开关，即每次用户指令执行结束是否
进入调试器
    int runUntilTime;               // 当运行时间到达该值时，进入调试器
};
```

- Thread 类

  我们继续查看 Thread.h，来查看运行用户程序时使用的用户寄存器。

```
/*
    该类定义了线程控制块。
    每个线程都拥有（1）线程执行栈（2）数组存储CPU寄存器状态（3）线程状态（运行、可运
行、阻塞）
    用户进程拥有用户地址空间，仅运行在内核态的线程没有地址空间
*/
class Thread {
  private:
    // 下述两个变量用于上下文切换，位置不可更改
    int* stackTop;                          // 当前栈指针
    _int machineState[MachineStateSize];    // 所有CPU寄存器状态

  public:
    Thread(char* debugName);                // 构造函数
    ~Thread();                              // 析构函数，运行态线程不可析构
        // 基础线程操作
    void Fork(VoidFunctionPtr func, _int arg); // 使线程运行在 (*func)
(arg) 函数位置
    void Yield();           // 当前线程，运行态 => 可运行态
    void Sleep();           // 当前线程，运行态 => 阻塞态
    void Finish();          // 线程运行结束
    void CheckOverflow();   // 检查线程栈是否溢出
    void setStatus(ThreadStatus st) { status = st; }    // 设置线程状态
    char* getName() { return (name); }      // 获取线程名
    void Print() { printf("%s, ", name); }  // 输出线程名

  private:
    int* stack;             // 栈底指针，主线程栈底指针为NULL
    ThreadStatus status;    // 线程状态（运行、可运行、阻塞）
    char* name;             // 线程名
    void StackAllocate(VoidFunctionPtr func, _int arg); // 为线程分配栈空
间，用于Fork函数内部实现

#ifdef USER_PROGRAM         // 如果为用户程序
    // 运行用户程序的线程有两组CPU寄存器，一个存储运行用户代码的线程状态，一个存储运
行内核代码的线程状态
    int userRegisters[NumTotalRegs];    // 用户态下CPU寄存器状态

  public:
    void SaveUserState();       // 保存用户态下寄存器状态
    void RestoreUserState();    // 恢复用户态下寄存器状态
    AddrSpace *space;             // 运行用户程序时的地址空间
#endif
};
```

由于 CPU 只有一个，因此 CPU 寄存器也只有一组。但每个用户程序至少需要映射到一个核心线程，因此每个核心线程都可能执行用户程序，所以每个核心线程都需要维护一组用户寄存器 userRegisters[]，用于保存与恢复相应的用户程序指令的执行状态。

- Scheduler::Run() 函数

  当用户进程进行上下文切换时，即执行用户进程的核心线程发生了上下文切换时，Nachos 就会将老进程的 CPU 寄存器状态保存到用户寄存器 userRegisters[] 中，并将新用户进程的寄存器状态恢复到 CPU 寄存器中，使得 CPU 能够继续执行上次被中断的用户程序。

  在 Scheduler::Run() 中，我们可以看到核心进程切换时对 CPU 寄存器与用户寄存器的保存与恢复，具体代码如下所示。

```
// 给CPU分配下一个线程，即进行上下文切换，需要保存旧线程状态并加载新线程状态。
void Scheduler::Run (Thread *nextThread) {
    Thread *oldThread = currentThread;      // 旧线程

#ifdef USER_PROGRAM                         // 运行用户程序
    if (currentThread->space != NULL) {
        currentThread->SaveUserState();     // 保存用户态下寄存器状态
        currentThread->space->SaveState();  // 保存地址空间状态
    }
#endif

    oldThread->CheckOverflow();             // 检查旧线程是否有栈溢出
    currentThread = nextThread;             // 当前线程切换到下一个线程
    currentThread->setStatus(RUNNING);      // 设置当前线程为运行态
    SWITCH(oldThread, nextThread);          // 新旧线程上下文切换
    // 一个线程运行结束时不可以直接删除，因为当前仍然运行在其栈空间中
    if (threadToBeDestroyed != NULL) {      // 如果之前设置了需要被删除的线程
        delete threadToBeDestroyed;         // 该变量在 Finish() 函数中设置
        threadToBeDestroyed = NULL;
    }

#ifdef USER_PROGRAM
    if (currentThread->space != NULL) {     // 如果新线程运行用户程序
        currentThread->RestoreUserState();  // 恢复运行用户程序时CPU寄存器状
态
        currentThread->space->RestoreState();// 恢复运行用户程序时地址空间
    }
#endif
}
```

- **源代码及注释**

  修改内容概述

  实验六是读懂代码，实验七是扩展 AddrSpace 类，实验八是实现各系统调用并且继续修改了 AddrSpace、Thread、Scheduler、List、OpenFile、BitMap、FileHeader、FileSystem 类以及 exception.cc 文件，接下来依次列出各文件的修改内容。

  此处需要注意 BitMap、FileHeader、FileSystem 的修改均为实验五中修改的内容，因此下面代码中不再重复列出。

  - AddrSpace
    - AddrSpace 类

      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。

FILESYS 部分的内容用于实现基于 FILESYS 的文件系统调用，主要是分配和释放文件 Id。

```
class AddrSpace {
  public:
    AddrSpace(OpenFile *executable);    // 创建地址空间
    ~AddrSpace();                       // 析构函数

    void InitRegisters();       // 初始化CPU寄存器
    void SaveState();           // 保存、储存地址空间
    void RestoreState();        // 恢复地址空间
    void Print();               // 打印页表
    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;// 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
};
```

- AddrSpace::AddrSpace()

```
#define MAX_USERPROCESSES 256

BitMap *AddrSpace::userMap = new BitMap(NumPhysPages);
BitMap *AddrSpace::pidMap = new BitMap(MAX_USERPROCESSES);

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
}
```

- AddrSpace::~AddrSpace()

  该函数需要将地址空间所分配到的物理空间、spaceId 等释放。

```
AddrSpace::~AddrSpace() {
  pidMap->Clear(spaceId-100);
  for(int i = 0; i < numPages; i++)
    userMap->Clear(pageTable[i].physicalPage);
  delete [] pageTable;
}
```

- AddrSpace::Print()

  该函数用于打印地址空间分配的页表。

```
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].physica
lPage);
    printf("============================================\n\n");
}
```

- AddrSpace 中寄存器保存与恢复

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

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

- AddrSpace 中基于 FILESYS 实现的函数

```
#ifdef FILESYS
int AddrSpace::getFileDescriptor(OpenFile *openfile) {
  for(int i = 3; i < 10; i++)
    if(fileDescriptor[i] == NULL){
      fileDescriptor[i] = openfile;
      return i;
    }
  return -1;
}

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

void AddrSpace::releaseFileDescriptor(int fd) {
  ASSERT((fd >= 0) && (fd < UserProgramNum));
  fileDescriptor[fd] = NULL;
}
#endif
```

- Thread
  - Thread 类

在 Thread 类中，首先增加了 TERMINATED 进程状态，并定义了 waitProcessSpaceId、 waitProcessExitCode、exitCode 等变量用于 Join() 系统调用的实现，以及一系列函数关于这三个变量的访问与设定。

```
// 线程状态
enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED,
TERMINATED };

class Thread {
  private:
    // 下述两个变量用于上下文切换，位置不可更改
    int* stackTop;            // 当前栈指针
    _int machineState[MachineStateSize];  // 所有CPU寄存器状态

    int* stack;              // 栈底指针，主线程栈底指针为NULL
    char *name;
    ThreadStatus status;    // 线程状态（运行、可运行、阻塞）
    // 为线程分配栈空间，用于Fork函数内部实现
    void StackAllocate(VoidFunctionPtr func, _int arg);

#ifdef USER_PROGRAM
    // 运行用户程序的线程有两组CPU寄存器，一个存储运行用户代码的线程状态，一个存
储运行内核代码的线程状态

    int userRegisters[NumTotalRegs];    // 用户态下CPU寄存器状态
    int waitProcessSpaceId, waitProcessExitCode, exitCode;
#endif

  public:
    AddrSpace *space;           // 运行用户程序时的地址空间
    Thread(char* debugName);         // 构造函数
    ~Thread();               // 析构函数，运行态线程不可析构

    // 下述为基础线程操作
    void Fork(VoidFunctionPtr func, _int arg);  // 使线程运行在
(*func)(arg) 函数位置
    void Yield();        // 当前线程，运行态 => 可运行态，调度其它线程
    void Sleep();        // 当前线程，运行态 => 阻塞态，调度其它线程
    void Finish();       // 线程运行结束

    void CheckOverflow();        // 检查线程栈是否溢出
    void setStatus(ThreadStatus st) { status = st; }    // 设置线程状
态
    char* getName() { return (name); }       // 获取线程名
    void Print() { printf("%s\n", name); }  // 输出线程名

#ifdef USER_PROGRAM
    void SaveUserState();        // 保存用户态下寄存器状态
    void RestoreUserState();     // 恢复用户态下寄存器状态
    void Join(int SpaceId);
    void Terminated();
    int userProgramId() { return space->getSpaceId(); }
    int ExitCode() { return exitCode; }
    int waitExitCode() { return waitProcessExitCode; }
    int setWaitExitCode(int tmpCode) { waitProcessExitCode =
tmpCode; }
    int setExitCode(int tmpCode) { exitCode = tmpCode; }
```

```
#endif
};
```

- Thread::Thread()

针对 Thread 类成员的增加，构造函数也需要作出一定的修改。

```
Thread::Thread(char* threadName) {
    name = new char[50];
    strcpy(name,threadName);
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif
}
```

- Thread::Finish()

该函数用于结束一个进程，并将其对应的 Joiner 从等待队列中唤醒。

```
void Thread::Finish () {
    (void) interrupt->SetLevel(IntOff);
    ASSERT(this == currentThread);
#ifdef USER_PROGRAM
    // 运行结束，执行Exit()命令时已获取退出码
    // Joinee 运行结束，唤醒 Joiner
    List *waitingList = scheduler->getWaitingList();
    // 检查 Joiner 是否在等待队列中
    ListElement *first = waitingList->listFirst(); // 队列首
    while(first != NULL){
        Thread *thread = (Thread *)first->item;     // 强转成Thread指
针
        if(thread->waitProcessSpaceId == userProgramId()){       //
在队列中
            // printf("yes\n");
            // 将子线程退出码赋给父进程的等待退出码
            thread->setWaitExitCode(exitCode);
            scheduler->ReadyToRun((Thread *)thread);
            waitingList->RemoveItem(first);
            break;
        }
        first = first->next;
    }
    Terminated();
#else
    DEBUG('t', "Finishing thread \"%s\"\n", getName());
    threadToBeDestroyed = currentThread;
    Sleep();
#endif
}
```

- Thread::Join()

该函数也属于 Join() 系统调用实现的一部分。

```
void Thread::Join(int SpaceId) {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);       // 关中断
    waitProcessSpaceId = SpaceId;                          // 设置当前
线程所等待进程的spaceId
    List *terminatedList = scheduler->getTerminatedList();  // 终止队
列
    List *waitingList = scheduler->getWaitingList();        // 等待队
列
    // 确定Joinee在不在终止队列中
    bool interminatedList = FALSE;
    ListElement *first = terminatedList->listFirst(); // 队列首
    while(first != NULL){
        Thread *thread = (Thread *)first->item;     // 强转成Thread指
针
        if(thread->userProgramId() == SpaceId){       // 在队列中
            interminatedList = TRUE;
            waitProcessExitCode = thread->ExitCode();  // 设置父线程等
待子线程退出码
            break;
        }
        first = first->next;
    }
    // Joinee不在终止队列中，可运行态或阻塞态
    if(!interminatedList){
        waitingList->Append((void *)this);  // 阻塞Joiner
        currentThread->Sleep();             // Joiner阻塞
    }
    // 被唤醒且Joinee在终止队列中，在终止队列中删除Joinee
    scheduler->deleteTerminatedThread(SpaceId);
    (void) interrupt->SetLevel(oldLevel);   // 开中断
}
```

- Thread::Terminated()

该函数为将一个进程终止并加入终止队列的具体代码。

```
void Thread::Terminated() {
    List *terminatedList = scheduler->getTerminatedList();
    ASSERT(this == currentThread);
    ASSERT(interrupt->getLevel() == IntOff);
    status = TERMINATED;
    terminatedList->Append((void *)this);
    Thread *nextThread = scheduler->FindNextToRun();
    while(nextThread == NULL){
        // printf("yes\n");
        interrupt->Idle();
        nextThread = scheduler->FindNextToRun();
    }
    scheduler->Run(nextThread);
}
```

- **Scheduler**
  - Scheduler 类

    在 Scheduler 类中，增加了进程等待、终止队列并添加了这两个队列所对应的函数，具体代码如下所示：

```
class Scheduler {
  public:
    Scheduler();             // 初始化调度队列
    ~Scheduler();            // 析构函数

    void ReadyToRun(Thread* thread);    // 将线程放入可运行队列
    Thread* FindNextToRun();           // 找到第一个可运行态线程
    void Run(Thread* nextThread);   // 运行线程
    void Print();            // 打印可运行线程队列

  private:
    List *readyList;       // 可运行态线程的队列

#ifdef USER_PROGRAM
  public:
    List *getReadyList() { return readyList; }
    List *getWaitingList() { return waitingList; }
    List *getTerminatedList() { return terminatedList; }
    void deleteTerminatedThread(int deleteSpaceId);
    void emptyList(List *tmpList) { delete tmpList; }
  private:
    List *waitingList;    // 等待运行线程的队列
    List *terminatedList; // 终止运行但未释放线程的队列
#endif
};
```

- Scheduler::Scheduler()

由于添加了新的类成员，因此需要在构造函数对类成员进行初始化。

```
Scheduler::Scheduler() {
    readyList = new List;
#ifdef USER_PROGRAM
    // 如果 Joinee 没有退出，Joiner 进入等待
    waitingList = new List;
    // 线程调用 Finish() 进入该队列，Joiner 通过检查该队列确定 Joinee 是否
已经退出
    terminatedList = new List;
#endif
}
```

- Scheduler::~Scheduler()

```
Scheduler::~Scheduler() {
    delete readyList;
    delete waitingList;
    delete terminatedList;
}
```

- Scheduler::deleteTerminatedThread()

该函数用于将一个线程从终止队列中移除，依然用于 Join() 系统调用的实现。

```
#ifdef USER_PROGRAM
void Scheduler::deleteTerminatedThread(int deleteSpaceId) {
    ListElement *first = terminatedList->listFirst();
    while(first != NULL){
        Thread *thread = (Thread *)first->item;
        if(thread->userProgramId() == deleteSpaceId){
            terminatedList->RemoveItem(first);
            break;
        }
        first = first->next;
    }
}
#endif
```

- **List**

  该部分内容的修改主要是辅助 Scheduler 中针对队列操作，具体修改部分如下所示。

  - List 类

    在该类中主要添加了两个函数，分别是 `void RemoveItem(ListElement *tmp)` 与 `ListElement *listFirst()`，均用于 Scheduler 中的队列操作。

```
class List {
  public:
    List();     // 初始化 List
    ~List();    // 析构函数

    void Prepend(void *item);   // 将 item 放到 List 首
    void Append(void *item);    // 将 item 放到 List 尾
    void *Remove();      // 将 item 从 List 首移除
    void Mapcar(VoidFunctionPtr func);  // 对 List 中每个元素应用
"func"
    bool IsEmpty();     // List 是否为空
    void RemoveItem(ListElement *tmp);  // 移除 List 中一个元素

    // Routines to put/get items on/off list in order (sorted by
key)
    void SortedInsert(void *item, int sortKey); // Put item into
list
    void *SortedRemove(int *keyPtr);           // Remove first item
from list
    ListElement *listFirst() { return first; }

  private:
    ListElement *first;     // List 首，NULL 则为空
    ListElement *last;      // List 尾
};
```

  - List::RemoveItem()

    该函数用于从 List 中删除一个 ListElement，用于实现 Scheduler 类中从终止队列中移除一个元素的功能。

```
void List::RemoveItem(ListElement *tmp) {
    bool isFind = FALSE;
    ListElement *now = first, *pre = NULL;
    while(now != NULL){
```

```
        if(now->item == tmp->item) { isFind = TRUE; break;}
        pre = now;
        now = now->next;
    }
    if(isFind){
        if(first == last) first = last = NULL;  // 队里只有一个元素
        else{
            if(pre == NULL) first = first->next;    // 删队首
            else if(now == last) {last = pre; last->next = NULL;}
// 删队尾
            else{   // 删中间
                pre->next = now->next;
            }
        }
    }
}
```

- **OpenFile**

  该部分内容的修改主要是用于基于 FILESYS 的文件系统调用的实现，具体修改部分如下所示。

  - OpenFile 类

    该类的修改主要针对于后续基于 FILESYS 实现的文件系统调用的实现。

```
#ifdef FILESYS
class FileHeader;

class OpenFile {
  public:
    OpenFile(int sector);   // 打开一个文件头在 sector 扇区的文件（DISK）
    ~OpenFile();            // 关闭文件

    void Seek(int position);    // 定位文件读写位置
    // 读取 numBytes 字节数据到 into 中，返回实际读取字节
    int Read(char *into, int numBytes);
    // 将 from 中 numByters 数据写入 OpenFile 中
    int Write(char *from, int numBytes);
    // 从 OpenFile 的 pos 位置读取字节到 into 中
    int ReadAt(char *into, int numBytes, int position);
    // 从 from 中的 pos 位置读取字节到 OpenFile 中
    int WriteAt(char *from, int numBytes, int position);
    int Length();       // 返回文件字节数
    void WriteBack();   // 将文件头写回 DISK 中

#ifdef FILESYS
    OpenFile(char *type) {}
    int WriteStdout(char *from, int numBytes);
    int ReadStdin(char *into, int numBytes);
#endif

  private:
    FileHeader *hdr;    // 文件头
    int seekPosition, hdrSector; // 文件当前读取位置，文件头所在扇区号
};
#endif
```

- OpenFile::WriteStdout()

  将数据写入输出对应的 OpenFile 中。

```
int OpenFile::WriteStdout(char *from, int numBytes) {
    int file = 1;
    WriteFile(file,from,numBytes);  // 将from文件数据写入file中
    return numBytes;
}
```

- OpenFile::ReadStdin()

  将 OpenFile 中的数据写入对应的 into 文件中。

```
int OpenFile::ReadStdin(char *into, int numBytes) {
    int file = 0;
    return ReadPartial(file,into,numBytes); // 将file文件数据写入into
中
}
```

- **exception.cc**

  该代码主要实现了实验中要求实现的各个系统调用，包括 Exec()、Exit()、Join()、Yield() 基础系统调用以及基于文件系统的 Create()、Open()、Write()、Read()、Close() 系统调用，此处分别实现了基于 FILESYS_STUB 与 FILESYS 两套文件系统的文件系统调用。

  - Exec()

    该系统调用主要用于执行一个新的 Nachos 文件，在 FILESYS 中从 DISK 中寻找该文件，在 FILESYS_STUB 中则在 UNIX 系统中寻找文件。

```
void AdvancePC(){
    machine->WriteRegister(PrevPCReg,machine->ReadRegister(PCReg));
// 前一个PC
    machine->WriteRegister(PCReg,machine->ReadRegister(NextPCReg));
// 当前PC
    machine->WriteRegister(NextPCReg,machine-
>ReadRegister(NextPCReg)+4);   // 下一个PC
}

void StartProcess(int spaceId) {
    // printf("spaceId:%d\n",spaceId);
    ASSERT(currentThread->userProgramId() == spaceId);
    currentThread->space->InitRegisters();     // 设置寄存器初值
    currentThread->space->RestoreState();      // 加载页表寄存器
    machine->Run();             // 运行
    ASSERT(FALSE);
}

case SC_Exec:{
    printf("This is SC_Exec, CurrentThreadId: %d\n",(currentThread-
>space)->getSpaceId());
    int addr = machine->ReadRegister(4);
    char filename[50];
    for(int i = 0; ; i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
```

```
    OpenFile *executable = fileSystem->Open(filename);
    if(executable == NULL) {
        printf("Unable to open file %s\n",filename);
        return;
    }
    // 建立新地址空间
    AddrSpace *space = new AddrSpace(executable);
    // space->Print();   // 输出新分配的地址空间
    delete executable;  // 关闭文件
    // 建立新核心线程
    Thread *thread = new Thread(filename);
    printf("new Thread, SpaceId: %d, Name: %s\n",space-
>getSpaceId(),filename);
    // 将用户进程映射到核心线程上
    thread->space = space;
    thread->Fork(StartProcess,(int)space->getSpaceId());
    machine->WriteRegister(2,space->getSpaceId());
    AdvancePC();
    break;
}
```

- Exit()

  该系统调用用于一个用户程序的退出，具体代码如下所示。

```
case SC_Exit:{
    printf("This is SC_Exit, CurrentThreadId: %d\n",(currentThread-
>space)->getSpaceId());
    int exitCode = machine->ReadRegister(4);
    machine->WriteRegister(2,exitCode);
    currentThread->setExitCode(exitCode);
    // 父进程的退出码特殊标记，由 Join 的实现方式决定
    if(exitCode == 99)
        scheduler->emptyList(scheduler->getTerminatedList());
    delete currentThread->space;
    currentThread->Finish();
    AdvancePC();
    break;
}
```

- Join()

  该系统调用用于一个父线程（Joiner）等待一个子线程（Joinee）运行结束后再继续运行，常用于同步设计中。

```
case SC_Join:{
    printf("This is SC_Join, CurrentThreadId: %d\n",(currentThread-
>space)->getSpaceId());
    int SpaceId = machine->ReadRegister(4);
    currentThread->Join(SpaceId);
    // waitProcessExitCode —— 返回 Joinee 的退出码
    machine->WriteRegister(2, currentThread->waitExitCode());
    AdvancePC();
    break;
}
```

- Yield()

```
case SC_Yield:{
    printf("This is SC_Yield, CurrentThreadId: %d\n",(currentThread-
>space)->getSpaceId());
    currentThread->Yield();
    AdvancePC();
    break;
}
```

- FILESYS_STUB - Create()

  该系统调用基于 FILESYS_STUB 实现了文件系统调用 Create()，即在 UNIX 系统中创建一个新文件，具体代码如下所示。

```
case SC_Create:{
    int addr = machine->ReadRegister(4);
    char filename[128];
    for(int i = 0; i < 128; i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    int fileDescriptor = OpenForWrite(filename);
    if(fileDescriptor == -1) printf("create file %s
failed!\n",filename);
    else printf("create file %s succeed, the file id is
%d\n",filename,fileDescriptor);
    Close(fileDescriptor);
    // machine->WriteRegister(2,fileDescriptor);
    AdvancePC();
    break;
}
```

- FILESYS_STUB - Open()

  该系统调用基于 FILESYS_STUB 实现了文件系统调用 Open()，即在 UNIX 系统中打开一个已经存在的文件，具体代码如下所示。

```
case SC_Open:{
    int addr = machine->ReadRegister(4);
    char filename[128];
    for(int i = 0; i < 128; i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    int fileDescriptor = OpenForWrite(filename);
    if(fileDescriptor == -1) printf("Open file %s
failed!\n",filename);
    else printf("Open file %s succeed, the file id is
%d\n",filename,fileDescriptor);
    machine->WriteRegister(2,fileDescriptor);
    AdvancePC();
    break;
}
```

- FILESYS_STUB - Write()

  该系统调用基于 FILESYS_STUB 实现了文件系统调用 Write()，即在 UNIX 系统中将数据写入一个已经存在的文件中，具体代码如下所示。

```
case SC_Write:{
    // 读取寄存器信息
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);       // 字节数
    int fileId = machine->ReadRegister(6);      // fd

    // 打开文件
    OpenFile *openfile = new OpenFile(fileId);
    ASSERT(openfile != NULL);

    // 读取具体数据
    char buffer[128];
    for(int i = 0; i < size; i++){
        machine->ReadMem(addr+i,1,(int *)&buffer[i]);
        if(buffer[i] == '\0') break;
    }
    buffer[size] = '\0';

    // 写入数据
    int writePos;
    if(fileId == 1) writePos = 0;
    else writePos = openfile->Length();
    // 在 writePos 后面进行数据添加
    int writtenBytes = openfile->WriteAt(buffer,size,writePos);
    if(writtenBytes == 0) printf("write file failed!\n");
    else printf("\"%s\" has wrote in file %d
succeed!\n",buffer,fileId);
    AdvancePC();
    break;
}
```

- FILESYS_STUB - Read()

  该系统调用基于 FILESYS_STUB 实现了文件系统调用 Read()，即在 UNIX 系统中将数据从一个已经存在的文件中读出，具体代码如下所示。

```
case SC_Read:{
    // 读取寄存器信息
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);       // 字节数
    int fileId = machine->ReadRegister(6);      // fd

    // 打开文件读取信息
    char buffer[size+1];
    OpenFile *openfile = new OpenFile(fileId);
    int readnum = openfile->Read(buffer,size);

    for(int i = 0; i < size; i++)
        if(!machine->WriteMem(addr,1,buffer[i])) printf("This is
something Wrong.\n");
    buffer[size] = '\0';
    printf("read succeed, the content is \"%s\", the length is
%d\n",buffer,size);
    machine->WriteRegister(2,readnum);
    AdvancePC();
    break;
}
```

- FILESYS_STUB - Close()

  该系统调用基于 FILESYS_STUB 实现了文件系统调用 Close()，即将一个已经分配的文件 Id 关闭，但不是删除这个文件。再次使用该文件需要重新打开，具体代码如下所示。

```
case SC_Close:{
    int fileId = machine->ReadRegister(4);
    Close(fileId);
    printf("File %d closed succeed!\n",fileId);
    AdvancePC();
    break;
}
```

- FILESYS - Create()

  该系统调用基于 FILESYS 实现了文件系统调用 Create()，即在 Nachos 系统中创建一个新文件，具体代码如下所示。

```
case SC_Create:{
    int addr = machine->ReadRegister(4);
    char filename[128];
    for(int i = 0; i < 128; i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    if(!fileSystem->Create(filename,0)) printf("create file %s
failed!\n",filename);
    else printf("create file %s succeed!\n",filename);
    AdvancePC();
    break;
}
```

- FILESYS - Open()

  该系统调用基于 FILESYS 实现了文件系统调用 Open()，即在 Nachos 系统中打开一个已经存在于 模拟硬盘 DISK 中的文件，具体代码如下所示。

```
case SC_Open:{
    int addr = machine->ReadRegister(4), fileId;
    char filename[128];
    for(int i = 0; i < 128; i++){
        machine->ReadMem(addr+i,1,(int *)&filename[i]);
        if(filename[i] == '\0') break;
    }
    OpenFile *openfile = fileSystem->Open(filename);
    if(openfile == NULL) {
        printf("File \"%s\" not Exists, could not open
it.\n",filename);
        fileId = -1;
    }
    else{
        fileId = currentThread->space->getFileDescriptor(openfile);
        if(fileId < 0) printf("Too many files opened!\n");
        else printf("file:\"%s\" open succeed, the file id is
%d\n",filename,fileId);
    }
```

```
    machine->WriteRegister(2,fileId);
    AdvancePC();
    break;
}
```

- FILESYS - Write()

  该系统调用基于 FILESYS 实现了文件系统调用 Write()，即在 Nachos 系统中将数据写入模拟硬盘 DISK 中已经存在的文件中，具体代码如下所示。

```
case SC_Write:{
    // 读取寄存器信息
    int addr = machine->ReadRegister(4);        // 写入数据
    int size = machine->ReadRegister(5);        // 字节数
    int fileId = machine->ReadRegister(6);       // fd

    // 创建文件
    OpenFile *openfile = new OpenFile(fileId);
    ASSERT(openfile != NULL);

    // 读取具体写入的数据
    char buffer[128];
    for(int i = 0; i < size; i++){
        machine->ReadMem(addr+i,1,(int *)&buffer[i]);
        if(buffer[i] == '\0') break;
    }
    buffer[size] = '\0';


    // 打开文件
    openfile = currentThread->space->getFileId(fileId);
    if(openfile == NULL) {
        printf("Failed to Open file \"%d\".\n",fileId);
        AdvancePC();
        break;
    }
    if(fileId == 1 || fileId == 2){
        openfile->WriteStdout(buffer,size);
        delete []buffer;
        AdvancePC();
        break;
    }

    // 写入数据
    int writePos = openfile->Length();
    openfile->Seek(writePos);

    // 在 writePos 后面进行数据添加
    int writtenBytes = openfile->Write(buffer,size);
    if(writtenBytes == 0) printf("Write file failed!\n");
    else if(fileId != 1 & fileId != 2)
        printf("\"%s\" has wrote in file %d
succeed!\n",buffer,fileId);
    AdvancePC();
    break;
}
```

- FILESYS - Read()

该系统调用基于 FILESYS 实现了文件系统调用 Read()，即在 Nachos 系统中将数据从模拟硬盘 DISK 中一个已经存在的文件中读出，具体代码如下所示。

```
case SC_Read:{
    // 读取寄存器信息
    int addr = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);       // 字节数
    int fileId = machine->ReadRegister(6);      // fd

    // 打开文件
    OpenFile *openfile = currentThread->space->getFileId(fileId);

    // 打开文件读取信息
    char buffer[size+1];
    int readnum = 0;
    if(fileId == 0) readnum = openfile->ReadStdin(buffer,size);
    else readnum = openfile->Read(buffer,size);

    // printf("readnum:%d,fileId:%d,size:%d\n",readnum,fileId,size);
    for(int i = 0; i < readnum; i++)
        machine->WriteMem(addr,1,buffer[i]);
    buffer[readnum] = '\0';

    for(int i = 0; i < readnum; i++)
        if(buffer[i] >= 0 && buffer[i] <= 9) buffer[i] = buffer[i]+0x30;
    char *buf = buffer;
    if(readnum > 0){
        if(fileId != 0)
            printf("Read file (%d) succeed! the content is \"%s\", the
length is %d\n",fileId,buf,readnum);
    }
    else printf("\nRead file failed!\n");
    machine->WriteRegister(2,readnum);
    AdvancePC();
    break;
}
```

- FILESYS - Close()

  该系统调用基于 FILESYS 实现了文件系统调用 Close()，即将一个已经分配的文件 Id 清楚，但不是删除这个文件。再次使用该文件需要重新打开，具体代码如下所示。

```
case SC_Close:{
    int fileId = machine->ReadRegister(4);
    OpenFile *openfile = currentThread->space->getFileId(fileId);
    if(openfile != NULL) {
        openfile->WriteBack(); // 将文件写入DISK
        delete openfile;
        currentThread->space->releaseFileDescriptor(fileId);
        printf("File %d closed succeed!\n",fileId);
    }
    else printf("Failed to Close File %d.\n",fileId);
    AdvancePC();
    break;
}
```

- **实验测试结果：**
  - Exec() 系统调用：

```
filename = ../test/exec.noff, which address space is:
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
==========================================

filenale = ../test/halt.noff, which address space is:
page table dump: 14 pages in total
==============================
        VirtPage,       PhysPage
         0,             11
         1,             12
         2,             13
         3,             14
         4,             15
         5,             16
         6,             17
         7,             18
         8,             19
         9,             20
         10,            21
         11,            22
         12,            23
         13,            24
==========================================

Machine halting!
```

  - Exit() 系统调用：

```
filename = ../test/exit.noff, which address space is:
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
==========================================

This is SC_Exit, CurrentThreadId: 100
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 142, idle 0, system 130, 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
```