# 系统调用实习报告

姓名张颜 学号 2001210541
日期 2020/12/24

# 目录

内容一：总体概述.....3
内容二：任务完成情况.....3
任务完成列表（Y/N）.....3
具体 Exercise 的完成情况.....3
内容三：遇到的困难以及解决方法.....10
内容四：收获及感想.....15
内容五：对课程的意见和建议.....15
内容六：参考文献.....15

# 内容一：总体概述

本次实验主要是阅读 nachos 关于系统调用的源代码，理解用户程序调用系统调用的整个过程，这个过程中涉及的操作和各个寄存器中的值和表达含义。并且通过实现几个系统调用加深对这些过程的理解。

# 内容二：任务完成情况

## 任务完成列表（Y/N）

| | Exercise1 | Exercise2 | Exercise3 | Exercise4 | Exercise5 |
|---|---|---|---|---|---|
| | Y | Y | Y | Y | Y |

## 具体 Exercise 的完成情况

### 第一部分

**Exercise1** 源代码阅读

阅读与系统调用相关的源代码，理解系统调用的实现原理。

code/userprog/syscall.h

code/userprog/exception.cc

code/test/start.s

系统调用是操作系统提供给用户的接口，为了保护操作系统，很多操作系统的内部函数不能直接提供给用户，所以采用系统调用的方式，用户程序进行的某些操作，需要在操作系统的内核态完成，这个时候就需要设计系统调用，让操作系统陷入内核态执行相关操作，最后再返回到用户态，继续陷入之前执行的代码，继续执行。

在 nachos 中，用户程序是无法直接调用 nachos 操作系统中定义的一些函数的，但是如果需要用到这些函数的功能，就可以通过设计操作系统对外的接口——系统调用来完成使用这些函数的功能。

在 syscall.h 头文件中定义了不同功能的系统调用一共十种，具体的十种系统调用如下：

```
#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
```

每种系统调用一个全局唯一的编号，nachos 操作系统通过这些指令的编号来识别系统调用的类型。在 start.s 文件中，描述了系统调用执行的过程。选择 Yield 系统调用作为例子看一看。

```
Yield:
addiu $2,$0,SC_Yield
syscall
j      $31
.end Yield
```

这段代码定义定义了一个系统调用 Yield，表示在用户程序使用 Yield 系统调用的时候，计算机将会把 SC_Yield(Yield 系统调用的全局唯一代码，具体的值是 10)放入寄存器 r2 中。之后生成一个系统调用异常，该异常的类型是枚举类型 ExceptionType，具体的值是 SyscallException(1)。根据 Mips 的编译决定了参数的传递满足规则：

参数 1： r4 寄存器
参数 2： r5 寄存器
参数 3： r6 寄存器
参数 4： r7 寄存器

如果系统调用有返回值，那么，根据 MIPS 的标准 C 调用习惯，返回值存放在 r2 寄存器中。

异常的处理函数在 exception.cc 中，通过传入的 SyscallException 参数来判断当前的异常是系统调用异常，通过读取 2 号寄存器的内容判断当前系统调用的类型。根据对这两个参数的判断，填充具体的每个系统调用的处理函数。

## 第二部分

**Exercise2** 系统调用实现

类比 Halt 的实现，完成与文件系统相关的系统调用：Create, Open，Close，Write，Read。Syscall.h 文件中有这些系统调用基本说明。

**Exercise 3** 编写用户程序

编写并运行用户程序，调用练习 2 中所写系统调用，测试其正确性。

- Create 系统调用主要是需要实现功能创建文件。根据系统调用头文件的描述，该函数的要根据给定的文件名创建文件。可以看到该系统调用有一个参数，没有返回值。那么根据 nachos 的规则，可以从 r2 寄存器中读取传入的参数，传入的参数为一个文件名的字符型指针。根据这个指针，从内存中一个字符一个字符地将文件名读出来。

```
char name[20];
int offset = 0;
int data;

while(true)
{
    machine->ReadMem(nameAddr + offset , 1 , &data);
    if (data == 0)
    {
        name[offset] = '\0';
        break;
    }
    name[offset] = char(data);
    offset+=1;
}
```

再调用操作系统文件系统提供的接口，fileSystem->Create(name , 128);以该文件名创建一个大小为 128 字节的文件。

如果此时在用户程序中直接调用该系统调用，那么程序将会陷入系统调用中没有办法出来不断循环。那是因为，系统调用结束后，应该执行用户程序中系统调用下一条指令。需要将 PC 寄存器向前推进一格，否则，从系统调用返回用户程序，下一条依然需要执行地指令还是系统调用指令，这样就陷入不断的循环。为了解决这个问题，在 Machine 类中定义了一个 PCOneTick()函数，让 PC 寄存器向后走一步。具体的：

```
void
Machine::PCOneTick()
{
    WriteRegister(PrevPCReg,registers[PCReg]);
    WriteRegister(PCReg,registers[PCReg]+sizeof(int));
    WriteRegister(NextPCReg,registers[NextPCReg]+sizeof(int));
}
```

PrevPCReg 寄存器记录了之前一条指令的地址，现在讲当前 PC 寄存器中的值放入这个寄存器。PCReg 寄存器中存储下一个要执行的指令地址，将系统调用指令（即当前 PC 寄存器中的地址）加上一个指令长度(sizeof(int))，表示用户程序中，系统调用下一条指令的地址，放入 PC 寄存器中。最后，将原先的 NextPCReg 内容加上一个指令长度，写回 NextPCReg 寄存器中。这样实现了跳出循环，用户程序从系统调用出来后会执行系统调用的下一个指令。

编写测试函数测试功能：在 test/halt.c 文件中加入 Create("testfile");编译执行。

测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
This is Create Syscall
Read filename Addr from Register4 : NameAddr = 272
Read filename from the given Addr : testfile
Then Create the file
this is halt syscall
Machine halting!

Ticks: total 29, idle 0, system 10, user 19
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...
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ll
total 525
drwxrwxrwx 1 vagrant vagrant   8192 Dec 23 05:43 ./
drwxrwxrwx 1 vagrant vagrant   4096 Dec 23 03:23 ../
-rwxrwxrwx 1 vagrant vagrant   6830 Sep  6  2015 addrspace.cc*
-rwxrwxrwx 1 vagrant vagrant   1293 Sep  6  2015 addrspace.h*
-rwxrwxrwx 1 vagrant vagrant  13328 Dec 23 03:20 addrspace.o*
-rwxrwxrwx 1 vagrant vagrant   4569 Sep  6  2015 bitmap.cc*
-rwxrwxrwx 1 vagrant vagrant   2202 Sep  6  2015 bitmap.h*
-rwxrwxrwx 1 vagrant vagrant  12300 Dec 23 03:17 bitmap.o*
-rwxrwxrwx 1 vagrant vagrant  11932 Dec 23 03:20 console.o*
-rwxrwxrwx 1 vagrant vagrant   3942 Dec 23 05:43 exception.cc*
-rwxrwxrwx 1 vagrant vagrant  16220 Dec 23 05:43 exception.o*
-rwxrwxrwx 1 vagrant vagrant  21532 Dec 23 03:20 interrupt.o*
-rwxrwxrwx 1 vagrant vagrant   7332 Dec 23 03:17 list.o*
-rwxrwxrwx 1 vagrant vagrant  17640 Dec 23 03:20 machine.o*
-rwxrwxrwx 1 vagrant vagrant  11424 Dec 23 03:20 main.o*
-rwxrwxrwx 1 vagrant vagrant  26224 Sep  6  2015 Makefile*
-rwxrwxrwx 1 vagrant vagrant  29516 Dec 23 03:20 mipssim.o*
-rwxrwxrwx 1 vagrant vagrant 165206 Dec 23 05:43 nachos*
-rwxrwxrwx 1 vagrant vagrant   2650 Dec 23 03:35 progtest.cc*
-rwxrwxrwx 1 vagrant vagrant  17436 Dec 23 05:38 progtest.o*
-rwxrwxrwx 1 vagrant vagrant  12068 Dec 23 03:20 scheduler.o*
-rwxrwxrwx 1 vagrant vagrant   3896 Dec 23 03:17 stats.o*
-rwxrwxrwx 1 vagrant vagrant    700 Dec 23 03:20 switch.o*
-rwxrwxrwx 1 vagrant vagrant   1402 Dec 23 03:20 swtch.s*
-rwxrwxrwx 1 vagrant vagrant   9604 Dec 23 03:20 synchlist.o*
-rwxrwxrwx 1 vagrant vagrant  16532 Dec 23 03:20 synch.o*
-rwxrwxrwx 1 vagrant vagrant   3865 Sep  6  2015 syscall.h*
-rwxrwxrwx 1 vagrant vagrant  22144 Dec 23 03:20 sysdep.o*
-rwxrwxrwx 1 vagrant vagrant  20652 Dec 23 03:20 system.o*
-rwxrwxrwx 1 vagrant vagrant      0 Dec 23 05:43 testfile*
-rwxrwxrwx 1 vagrant vagrant  21700 Dec 23 03:20 thread.o*
-rwxrwxrwx 1 vagrant vagrant   7336 Dec 23 03:20 threadtest.o*
-rwxrwxrwx 1 vagrant vagrant   7268 Dec 23 03:20 timer.o*
```

- Open 系统调用主要是实现打开文件功能。根据系统调用头文件描述，Open 系统调用具体的功能是根据给定的文件名打开文件，并且返回一个文件描述符。同样从 r4 寄存器中获取参数的值，表示文件名字符串所在的地址，通过这个地址得到文件名，得到文件名后，调用 nachos 文件系统提供的 open 函数将文件打开，获得文件描述符，并且将最后的结果存入 r2 寄存器中，置 PC 的值前进一步。关键代码：

```
OpenFile* openfile = fileSystem->Open(name);
printf("Now Get the FileID(Handler) is : %d\n" , int(openfile));
printf("Write this value into rg2 as the retrurn value\n" , openfile);
machine->WriteRegister(2 , int(openfile));
machine->PCOneTick();
```

编写测试函数测试：
在 halt.c 中调用 Open("testfile");
测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
Successfully Add Thread main to GlobalList!
This is Open Syscall
Read filename Addr from Register4 : NameAddr = 336
Read filename from the given Addr : testfile
Then Open the file
Now Get the FileID(Handler) is : 150419112
Write this value into rg2 as the retrurn value
this is halt syscall  current thread is : main
Machine halting!

Ticks: total 29, idle 0, system 10, user 19
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...
```

- Close 系统调用主要的功能是将打开的文件关闭。根据系统调用头文件的说明，这个系统调用主要是根据打开文件的文件描述符关闭文件。具体实现来说，先读取 r3 寄存器中的值，r3 寄存器中存储的是打开文件的文件描述符。该文件描述符表现为 int 类型实际上是 Openfile 指针类型。将该 int 类型转换为指针类型，并且 delete 对应的指针即可关闭文件。

关键代码：

```
int fid = machine->ReadRegister(4);
printf("Read fid from Register4 fid = %d\n",fid);
OpenFile *openfile = (OpenFile*)fid;
printf("Now Delete the openfile obj to close the file\n");
delete openfile;
machine->PCOneTick();
```

编写测试代码测试：
Close(Open("testfile"));
测试结果:;

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
This is Open Syscall
Read filename Addr from Register4 : NameAddr = 288
Read filename from the given Addr : testfile
Then Open the file
Now Get the FileID(Handler) is : 155030272
Write this value into rg2 as the retrurn value
This is Close Syscall
Read fid from Register4 fid = 155030272
Now Delete the openfile obj to close the file
this is halt 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
```

- Write 系统调用主要是实现向文件中写内容的功能。根据系统调用头文件的说明，这个系统调用主要是根据文件标识符，从该文件中读取指定大小的内容到 buffer 中。该系统调用有三个参数分别是 char*buffer 标志要写入字符的首地址，int size 表示要写入内容的大小，OpenfileId id 表示要写入文件的文件描述符。具体来说在 r4 寄存器中获取 buffer 的首地址，在 r5 寄存器中获取 size 大小，在 r6 寄存器中获取文件描述符。而后根据 buffer 首地址和 size 在内存中获取要写入文件的内容，并且调用 nachos 文件系统的接口，将该内容写入文件中去。

关键代码：

```
printf("This is Write Syscall \n");
int bufferAddr = machine->ReadRegister(4);
int size = machine->ReadRegister(5);
int fid = machine->ReadRegister(6);
printf("Read filename Addr from Register4 : bufferA
char content[size];
int data;
for(int i = 0 ; i < size; i++)
{

    machine->ReadMem(bufferAddr + i , 1 , &data);
    content[i] = char(data);
    printf("data = %c\n", data);
}
printf("Now write the content into the file\n");
OpenFile *openfile = (OpenFile*) fid;
openfile->Write(content , size);
machine->PCOneTick();
```

编写测试代码测试：

```
Create("testfile");
int id = Open("testfile");
char * buffer = "01234567890123456789";
Write(buffer, 10, id);
Halt();
```

执行结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
This is Open Syscall
Read filename Addr from Register4 : NameAddr = 324
Read filename from the given Addr : testfile
Then Open the file
Now Get the FileID(Handler) is : 161366784
Write this value into rg2 as the retrurn value
This is Write Syscall
Read filename Addr from Register4 : bufferAddr = 304 Read size from Register5 : size = 10 Read fileId from Register6 : id = 161366784
data = 0
data = 1
data = 2
data = 3
data = 4
data = 5
data = 6
data = 7
data = 8
data = 9
Now write the content into the file
this is halt syscall
Machine halting!

Ticks: total 40, idle 0, system 10, user 30
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
```

```
userprog > ≡ testfile
  1    0123456789
```

- Read 系统调用主要是实现从文件中读内容的功能。根据系统调用头文件的说明，这个系统调用主要是根据文件标识符，从向该文件中写入指定大小的内容到 buffer 中。该系统调用有三个参数分别是 char*buffer 表示读取文件内容到 buffer 中，int size 表示要读取内容的大小，OpenfileId id 表示要读取文件的文件描述符。具体来说在 r4 寄存器中获取 buffer 的首地址，在 r5 寄存器中获取 size 大小，在 r6 寄存器中获取文件描述符。而后根据文件描述符和 size 大小调用 nachos 文件系统提供的接口，从指定文件中读取 size 大小的内容，放入一个临时数组验证，最后将该数组中的每一个人字符逐个写入内存中 buffer 指向的字符串。最终将读取的字节数作为返回值传递到 r2 寄存器中。

  关键代码：

```
printf("This is Read Syscall \n");
int bufferAddr = machine->ReadRegister(4);
int size = machine->ReadRegister(5);
int fid = machine->ReadRegister(6);
printf("Read filename Addr from Register4 : bufferAddr = %d

OpenFile *openfile = (OpenFile*) fid;
char content[size];
int readNum = openfile->Read(content , size);
printf("Read content into from file:");
for(int i = 0 ; i < size ; i ++)
{
    machine->WriteMem(bufferAddr+i , 1 , int(content[i]));
    printf("%c",content[i]);
}
printf("\nNow write the retNum = %d into Rg2\n" , readNum);
machine->WriteRegister(2,readNum);
machine->PCOneTick();
```

编写测试函数：

```
Create("testfile");
int id = Open("testfile");
char * buffer = "01234567890123456789";
Write(buffer, 10, id);
Read(buffer, 10, Open("testfile"));
Halt();
```

测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
This is Open Syscall
Read filename Addr from Register4 : NameAddr = 288
Read filename from the given Addr : testfile
Then Open the file
Now Get the FileID(Handler) is : 164037376
Write this value into rg2 as the retrurn value
This is Read Syscall
Read filename Addr from Register4 : bufferAddr = 0 Read size from Register5 : size = 10 Read fileId from Register6 : id = 164037376
Read content into from file:0123456789
Now write the retNum = 10 into Rg2
this is halt 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...
```

## 第三部分

**Exercise 4**　系统调用实现

实现如下系统调用：Exec，Fork，Yield，Join，Exit。Syscall.h 文件中有这些系统调

用基本说明。

**Exercise 5** 编写用户程序

编写并运行用户程序，调用练习 4 中所写系统调用，测试其正确性。

- Exec 系统调用的主要功能是执行用户程序。根据系统调用头文件的说明，这个系统调用的主要参数是文件名，根据文件名找到需要执行的用户文件，创建线程运行用户程序，并且返回线程标识符。具体的实现，先通过 r3 寄存器得到文件名所在的地址，再根据地址得到文件名。得到了用户可执行文件的文件名，那么就可以为用户程序分配内存空间，将可以执行文件读取进入内存，创建新线程，使用该分配的内存空间，运行用户程序。最后将新建的线程 id 写入 r2 寄存器中。关键代码：

```
printf("this is Exec syscall \n");
int nameAddr = machine->ReadRegister(4);
printf("Read filename Addr from Register4 : NameAddr = %d\n",nameAddr);

Thread *userThread = new Thread("StartUerPro");
userThread->Fork(StartUerPro , nameAddr);
printf("StartUserPro Pid %d\n", userThread->getPid());
machine->WriteRegister(2 , userThread->getPid());
machine->PCOneTick();
```

```
void StartUerPro(int nameAddr)
```

```
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;
space->InitRegisters();
space->RestoreState();
machine->Run();
ASSERT(FALSE);
```

编写测试代码测试系统调用：

Exec("../test/sort");

测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
Successfully Add Thread main to GlobalList!
this is Exec syscall
Read filename Addr from Register4 : NameAddr = 272
Successfully Add Thread StartUerPro to GlobalList!
This is func startuserpro
Exec filename from the given Addr : ../test/sort
Thread StartUerPro EXIT
Successfully Remove Thread StartUerPro from GlobalList!
Thread main EXIT
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 7024, idle 0, system 30, user 6994
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...
```

- Join 系统调用。根据系统调用文件头的描述，主要是等待一个线程结束。该系统调用主要的参数是线程 id，表示着需要等待结束的线程 id。主要的实现思路是循环判断该线程 id 是否存在在当前机器中，如果存在的话，直接让出 CPU 让线程继续执行，知道该 id 的线程被结束。关键代码：

```
int threadid = machine->ReadRegister(4);
printf("Get thread %d\n",threadid);
while(globalThreadManager->GlobalThreadList[threadid]!=NULL)
{
    currentThread->Yield();
}
printf("Thread %d has exited\n",threadid);
//globalThreadManager->ShowListInfo();
machine->PCOneTick();
```

编写测试代码测试：

Join(Exec("../test/sort"));

测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
Successfully Add Thread main to GlobalList!
this is Exec syscall
Read filename Addr from Register4 : NameAddr = 348
Successfully Add Thread StartUerPro to GlobalList!
StartUserPro Pid 1
This is Join syscall
Get thread 1
This is func startuserpro
Exec filename from the given Addr : ../test/sort
this is Exit syscall
Thread StartUerPro Exit with status : 0
Successfully Remove Thread StartUerPro from GlobalList!
Thread 1 has exited
Machine halting!

Ticks: total 2388, idle 0, system 40, user 2348
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...
```

- Yield 系统调用，根据系统调用文件头的描述，主要是让当前执行的线程让出 CPU，该调用没有参数也没有返回值，故而，直接调用 currentthread->Yield()让出 CPU 即可。关键代码：

```
printf("This is Yield Syscall \n");
machine->PCOneTick();
currentThread->Yield();
```

- Fork 系统调用，根据系统调用文件头的描述，该系统调用会传递一个函数指针作为参数，需要一个新的线程来执行这个函数中的内容。具体的实现逻辑是新建线程，调用 nachos 提供的 fork 函数，执行自定义的 syscallFork 函数，把当前线程的地址空间的指针作为参数传入函数。同时新增一个成员变量 funcAddr 在 space 类中，funcAddr 记录着用户程序传入的函数入口地址。在 syscallFork 中，执行的是另一个线程。将用户线程的地址空间和新线程的地址空间共用，同时将 PC 值设置为 funcAddr，即用户定义函数的入口地址。新线程执行的就是用户自定义函数的内容。

关键代码：

```
Thread *syscallForkThread = new Thread("syscallForkThread");
int funcAddr = machine->ReadRegister(4);
currentThread->space->funcAddr = funcAddr;
syscallForkThread->Fork(syscallFork , int(currentThread->space));
machine->PCOneTick();
```

```
void syscallFork(int spacePointer)
{
    printf("Come into syscallFork function  current thread: %s\n", currentThread->getName());
    currentThread->space = (AddrSpace*)spacePointer;
    machine-> WriteRegister(PCReg,currentThread->space->funcAddr);
    machine-> WriteRegister(NextPCReg,currentThread->space->funcAddr+sizeof(int));
    machine->Run();
}
```

```
printf("this is Exit syscall\n");
int status = machine->ReadRegister(4);
printf("Thread %s Exit with status : %d \n",currentThread->getName() , status);
//machine->clear();
machine->PCOneTick();
currentThread->Finish();
```

编写测试代码测试：

```
void
testfunc()
{
    Open("testfile");
    Exit(0);
}
int
main()
{
    Fork(testfunc);
```

测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
Successfully Add Thread main to GlobalList!
this is Fork syscall
Successfully Add Thread syscallForkThread to GlobalList!
Come into syscallFork function  current thread: syscallForkThread
This is Open Syscall
Read filename Addr from Register4 : NameAddr = 336
Read filename from the given Addr : testfile
Then Open the file
Now Get the FileID(Handler) is : 152238080
Write this value into rg2 as the retrurn value
This is Yield Syscall
this is halt syscall  current thread is : main
Machine halting!

Ticks: total 63, idle 0, system 30, user 33
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...
```

- Exit 系统调用：该系统调用主要功能是退出线程。从参数中读取退出的状态，而后调用 currentThread->Finish()结束当前线程。关键代码：

```
printf("this is Exit syscall\n");
int status = machine->ReadRegister(4);
printf("Thread %s Exit with status : %d \n",currentThread->getName() , status);
//machine->clear();
machine->PCOneTick();
currentThread->Finish();
```

编写测试函数进行测试:

Exit(0);

测试结果：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/halt
Successfully Add Thread main to GlobalList!
this is Exit syscall
Thread main Exit with status : 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 22, idle 0, system 10, user 12
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

# 内容三：遇到的困难以及解决方法

## 困难

系统调用的困难之处其实主要是前期在代码阅读中的遇到的困难。因为对汇编代码的了解不足够，所以需要辅助 nachos 的文档来查看哪些寄存器存储哪些内容。对于 Fork 函数的实现思路，参考了网上许多其他的实现思路，最终决定共享地址空间。由于本次实验是在其他实验的基础上进行的，所以相比较于其他实验，不用从头写起。

# 内容四：收获及感想

之前对系统调用只停留在会用的层面，不知道具体的系统调用的实现过程。通过本次实验，亲手模拟了系统调用的实现过程，让我对系统调用有了更加深刻的认识。特别是用户如何传递系统调用参数，在系统调用执行完毕之后，要返回到用户程序的下一个 PC 的地址，否则会陷入系统调用中没有办法返回用户态。对于这些 Bug 有了更加深刻和具体的认识。

# 内容五：对课程的意见和建议

暂时无。

# 内容六：参考文献

*[1]《nachos 中文教程》*
*https://wenku.baidu.com/view/905197a9e209581b6bd97f19227916888486b90b.html*
*[2]《nachos 学习笔记（五）》https://blog.csdn.net/darord/article/details/83303765*