**中国矿业大学计算机学院**

**2020 级本科生课程报告**

课程名称 操作系统课程设计

报告时间 2022年5月29日

学生姓名 杨再润

学 号 06203203

专 业 数据科学与大数据

任课教师 鲍 宇

目录

[实验四 系统调用 3](#_Toc104740325)

[【实验目的】 3](#_Toc104740326)

[【实验内容】 3](#_Toc104740327)

[【实验步骤】 3](#_Toc104740328)

[【运行结果】 5](#_Toc104740329)

[【思考练习】 7](#_Toc104740330)

[【实验感受】 10](#_Toc104740331)

# 实验四 系统调用

## 【实验目的】

* 深入了解Linux系统调用的执行过程，建立对系统调用的深入认识。
* 学会增加系统调用及添加内核函数的方法。

## 【实验内容】

1. **准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

1. **在Linux 0.11内核中添加新的系统调用**

为Linux 0.11添加一个新的系统调用max函数，该函数实现比较两个参数的大小并将较大值返回的功能。

1. **在Linux 0.11应用程序中测试新的系统调用**

编辑main.c文件中的源代码，在源代码文件中使用C语言添加max函数的完整实现，并在其中通过中断和系统调用号访问新添加的系统调用。

1. **调试系统调用执行的过程**

根据 \_syscall0、\_syscall1、\_syscall2、\_syscall3这四个宏（在文件include/unistd.h中的第167行定义）可知系统调用执行的过程如下：应用程序通过调用int 0x80中断，进入内核中的系统调用总入口\_system\_call 函数，该函数以存放在EAX寄存器中的系统调用号作为系统调用函数指针表的索引，找到相应的内核函数，并通过EBX、ECX、EDX将参数传递给内核函数，最后通过call指令执行该内核函数并使用EAX寄存器返回结果。

为了调试系统调用执行的过程，需要在系统调用总入口\_system\_call函数中添加一个断点。但是，因为Linux操作系统在启动的过程中会多次产生int 0x80调用，在\_system\_call处的断点就会被多次命中。为了减少断点的命中次数，需要将此断点设置为一个条件断点。

1. **提交作业**

实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

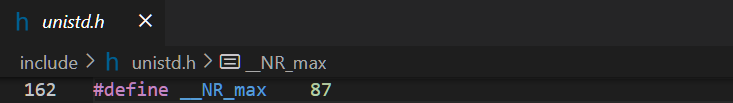
## 【实验步骤】

1. **准备实验**
2. 构建项目
3. Bochs 运行（不调试）
4. **在Linux 0.11内核中添加新的系统调用**
5. 首先要为新系统调用分配一个唯一的系统调用号，以原来的最大系统调用号为基础加1作为新的系统调用号。
6. 添加新系统调用号的同时，也要使系统调用总数在原来的基础上增加1。
7. 在include/linux/sys.h文件中的第88行使用C语言声明内核函数的原型。
8. 在kernel/sys.c文件的最后编写代码，实现新系统调用对应的内核函数。
9. 生成Linux 0.11内核，修改语法错误直到生成成功。
10. **在Linux 0.11应用程序中测试新的系统调用**
11. 按F5启动调试。
12. 编辑main.c文件中的源代码.
13. 使用命令gcc main.c -o main生成可执行文件main。
14. 执行sync命令，将文件保存到磁盘。
15. 执行 chmod +x main 命令为main文件添加可执行权限。
16. 执行./main命令运行main。
17. **调试系统调用执行的过程**
18. 结束之前的调试过程。
19. 打开kernel/system\_call.s文件，在标号\_system\_call后的第一行汇编代码处（第102行）添加一个断点。
20. 在刚刚添加的断点上点击鼠标右键，在弹出的菜单中选择“Edit Breakpoint”，会在编辑器中显示出用于输入条件表达式的编辑框。在编辑框中设置断点条件 $eax==87 后按回车确认
21. 按F5启动调试（注意，由于添加了一个条件断点，需要调试器频繁验证条件是否满足，这会导致启动过程明显变慢，请读者耐心等待启动完毕）。
22. 待Linxu 0.11启动后输入命令./main运行应用程序main，会命中刚刚添加的条件断点。
23. 选择“View”菜单中的“Run”，打开左侧的 “运行与调试”窗口。
24. 在“运行与调试”窗口展开CPU寄存器，会发现EAX寄存器中的值为0x57（十进制为87），和max系统调用的调用号一致，说明应用程序正在调用max系统调用函数。查看EBX和ECX寄存器的值，存放的分别是max函数的参数0x64（十进制为100）和0xc8（十进制为200）。
25. 按F10单步调试，直到黄色箭头指向第110行。其中第103行进行错误检查，第104-106行将各个段寄存器的值压入栈进行现场保护，第107-109行将保存在EBX、ECX和EDX寄存器中的参数压入栈，这与C语言参数从右到左压入栈的顺序是一致的。
26. 按F10继续单步调试，直到黄色箭头指向第119行。该行代码使用EAX寄存器存放的系统调用号作为系统调用函数指针表的下标，通过call指令调用对应的内核函数。
27. 按F11调试进入kernel/sys.c文件中的系统调用对应的内核函数
28. 按F10单步调试，直到从内核函数返回到kernel/system\_call.s文件中的第120行。从内核函数返回后，返回值200会存放在EAX寄存器中
29. 按F10单步调试，直到黄色箭头指向第168行。其中，因为后续工作（进程调度、信号处理）会用到EAX寄存器，所以120-121行会将EAX存放的返回值（200）入栈，并把当前任务数据结构地址存入EAX寄存器，122-160行会进行进程调度以及信号的处理工作，161行-167行，恢复现场，恢复通用寄存器以及段寄存器，与保护现场时的顺序相反。此时查看监视窗口中的EAX，EBX和ECX寄存器的值，分别恢复为200（返回值），100（参数一），200（参数二）
30. 按F5继续运行，第168行的iret指令从0x80中断返回到main程序中。此时打开Bochs虚拟机的Display窗口，可以看到main程序已经执行完毕了。
31. 结束调试。
32. **提交作业**

实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

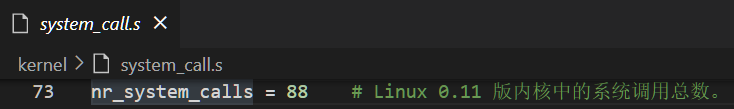
## 【运行结果】

1. **在Linux 0.11内核中添加新的系统调用**



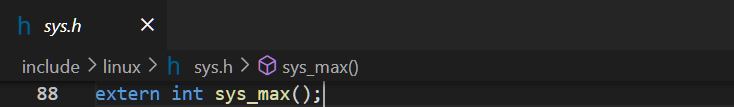
**图 添加新的调用号**

**打开include/unistd.h文件，在第162行添加新的系统调用号\_\_NR\_max**



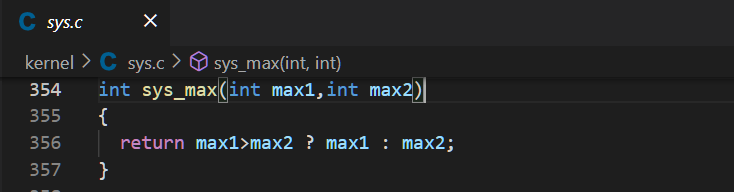
**图 系统调用总数**

**打开kernel/system\_call.s文件，修改在第73行定义的系统调用总数**



**图 系统调用表中增加函数指针**

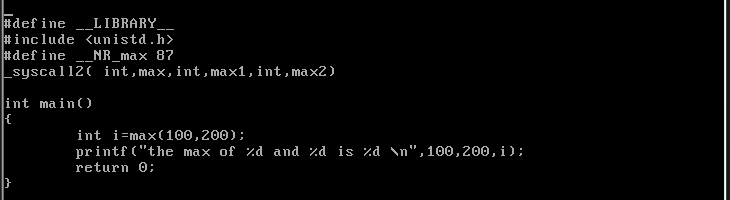
**在include/linux/sys.h文件中的第88行使用C语言声明内核函数的原型**



**图 内核函数的实现**

**在kernel/sys.c文件的最后编写代码，实现新系统调用对应的内核函数**

1. **在Linux 0.11应用程序中测试新的系统调用**



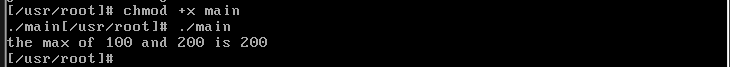
**图 编辑main.c文件**

**使用vi编辑器新建一个main.c文件。编辑main.c文件中的源代码（如图4-6所示）。其中，需要定义 \_\_LIBRARY\_\_ 宏及包含unistd.h头文件，还需要再次定义 \_\_NR\_max 宏。**



**图 编译main.c文件**

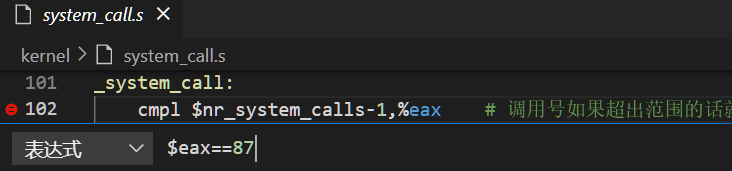
**使用命令gcc main.c -o main生成可执行文件main**



**图 运行main可执行文件**

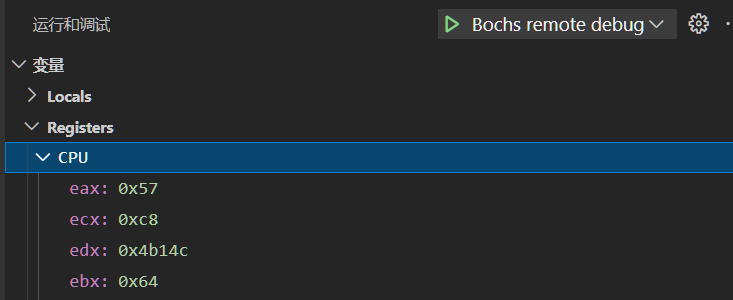
**执行 chmod +x main 命令为main文件添加可执行权限，执行./main命令运行main。**

1. **调试系统调用执行的过程**



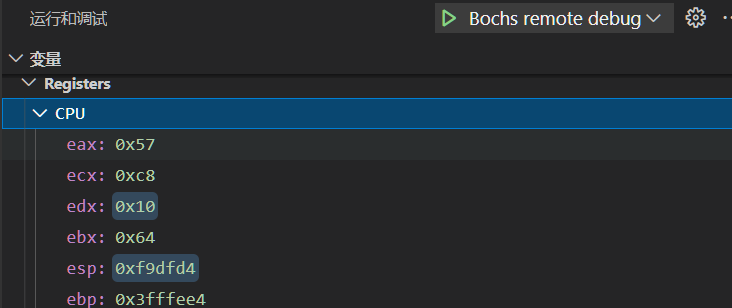
**图 编辑断点**

**在编辑框中设置断点条件 $eax==87**



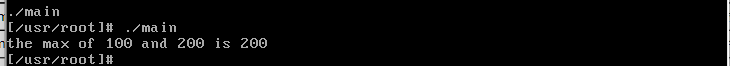
**图 命中断点时寄存器的值**

**和max系统调用的调用号一致，说明应用程序正在调用max系统调用函数。查看EBX和ECX寄存器的值，存放的分别是max函数的参数**



**图 运行若干步后寄存器的值**

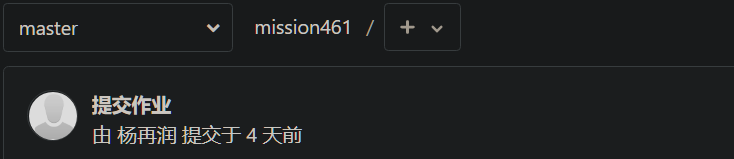
**运行若干步后寄存器edx的值变为0x10,esp的值变为0xf9dfd4**



**图 运行结束**

**调试完成后，发现程序已经运行结束。**

1. **提交作业**





**图 提交作业**

## 【思考练习】

1. 参考《Linux内核完全注释》第8.5.3.3节的内容，编写一个汇编程序，直接使用系统调用。

编写代码如下

volatile void \_exit(int exit\_code)

{

\_asm\_("int $0x80"::"a"(\_NR\_exit),"b"(int exit\_code));

}

这里用到了嵌入汇编，这里执行的动作就是调用系统中断int 0x80，在调用系统中断之前，将系统调用号\_NR\_exit放到寄存器eax中，退出码exit\_code放到寄存器ebx中。

用于关键字volatile告诉编译器gcc告诉该函数没有返回，所以在嵌入汇编中并没有给出输出寄存器。

1. 在Linux 0.11内核中添加两个系统调用函数Iam和Whoami，函数原型如下：

* int Iam(const char\* name)；将字符串name的内容保存到内核中，返回值是拷贝的字符数，如果name长度大于32，则返回-1，并置errno为EINVAL。
* int Whoami(char\* name, int size)；将Iam保存到内核中的字符串拷贝到数据缓冲区name中，size为数据缓冲区name的长度，返回值是拷贝的字符数。如果size小于所需空间，则返回-1，并置errno为EINVAL。

编写两个应用程序。其中，Iam应用程序调用Iam函数，并使用命令行中的第一个参数作为Iam函数的参数；Whoami应用程序调用Whoami函数，并打印输出获取的字符串。

**过程如下：**

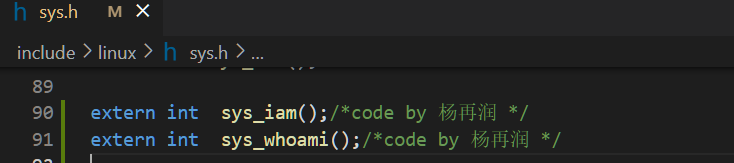


图 添加俩新的系统调用函数

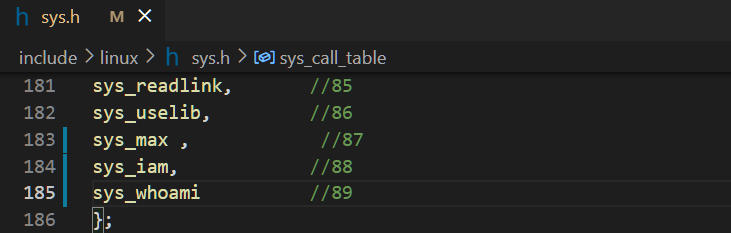


图 在调用表中注册俩新的函数

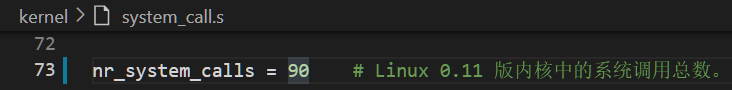


图 系统调用总数+2

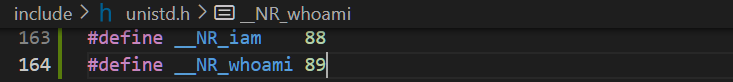


图 分配新的调用号

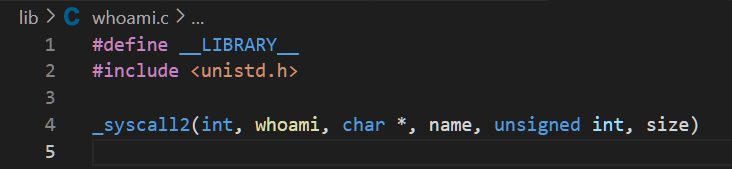


图 添加新文件whoami.c

该宏展开就是函数int whoami(char\* name, unsigned int size)

头文件<unistd.h>中定义了该宏的展开形式和规则，其通过内嵌汇编的方式调用中断

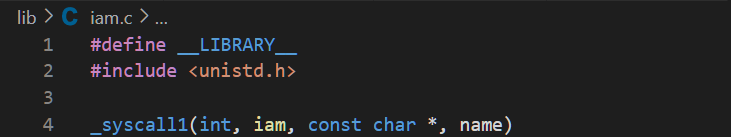


图 添加iam.c

该宏展开就是函数int iam(const char\* name)

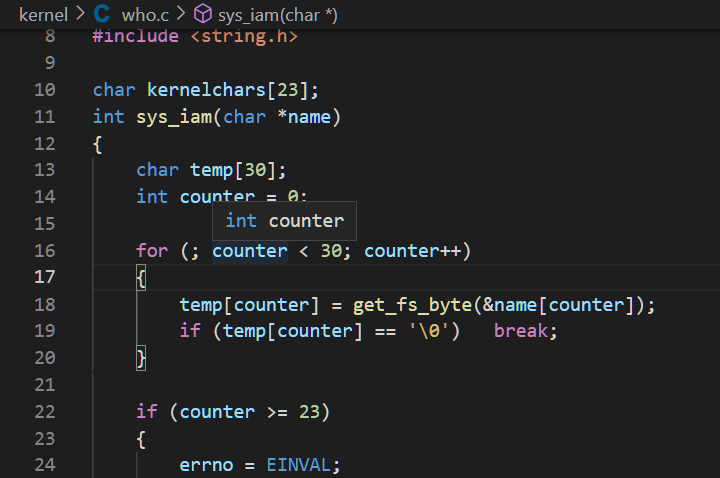


图 函数sys\_iam()和sys\_whoami()的实现

功能：将内核中由iam()保存的字符串读取出来，保存到name指向的用户空间(该用户空间应预留'\0'的位置)，同时确保不会对name访问越界

返回值：复制的字符数,包含'\0'

若size小于需要的空间，则返回-1，并设置errno=EINVAL



图 编写测试文件

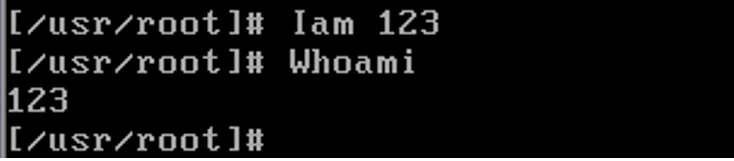


图 测试结果

## 【实验感受】

* 系统采用的老版本C语言与我所学的在语法上有些许出入，常常带来意想不到的麻烦。
* 通过这次实验，也掌握了vi编辑文本的技能，还有一些其他的常用Linux命令
* 通过本次实验，掌握了VSCode的调试功能
* 本次实验掌握了系统调用的实现，深入内核直达硬件，加深了对计算机体系结构的认识