卡通人物

描述已自动生成

**操作系统**

**课程实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 实验名称 | 系统调用 | | |
| 姓名 | 常添 | 院系 | 计算学部 |
| 班级 | 2203102 | 学号 | 2022111699 |
| 任课教师 | 郑铁然 | 指导教师 | 郑铁然 |
| 实验地点 | G709 | 实验时间 | 2024.12.8 |

****

**实验4：系统调用**

**1. 实验目的**

1. **深入理解系统调用机制**：通过向 Linux 0.11 添加新的系统调用，掌握系统调用的基本过程和核心实现。
2. **实现用户态与内核态的交互**：学会如何在用户空间与内核空间之间传递数据。
3. **编写和测试系统调用**：通过添加 iam() 和 whoami() 系统调用，验证自定义系统调用的功能。

**2. 实验环境**

1. **操作系统**：Linux 0.11 内核（通过 Bochs 模拟器运行）。
2. **开发工具**：
   * **Bochs**：x86 模拟器，用于运行 Linux 0.11。
   * **GCC**：C 编译器，版本 3.4.6。
3. **实验目录结构**：
   * **内核源码路径**：~/oslab/linux-0.11
   * **测试程序路径**：/usr/root（在虚拟机中）

**3. 实现功能**

1. **系统调用 iam()**：
   * 功能：将用户提供的字符串（名字）保存到内核空间中。
   * 接口原型：int iam(const char \*name);
   * 参数约束：名字长度不能超过 23 个字符。
   * 返回值：
     + 成功时返回拷贝的字符数。
     + 如果长度超过 23，返回 -1，并设置 errno 为 EINVAL。
2. **系统调用 whoami()**：
   * 功能：将内核空间保存的名字拷贝到用户提供的缓冲区中。
   * 接口原型：int whoami(char \*name, unsigned int size);
   * 参数约束：确保不会越界访问，size 说明缓冲区大小。
   * 返回值：
     + 成功时返回拷贝的字符数。
     + 如果 size 小于名字长度，返回 -1，并设置 errno 为 EINVAL。
3. **测试程序**：
   * 编写用户态程序 iam.c 和 whoami.c 测试系统调用的功能。
   * 通过命令行交互，验证名字的存取功能。

**4. 代码流程解析**

**4.1 修改内核头文件**

在 hdc挂载磁盘的include/unistd.h 中添加系统调用号：

#define \_\_NR\_iam 72

#define \_\_NR\_whoami 73

**4.2 实现系统调用函数**

在 kernel/who.c 文件中实现 sys\_iam 和 sys\_whoami：

#include <asm/segment.h>

#include <errno.h>

static char stored\_name[24]; // 保存用户的名字

int sys\_iam(const char \*name) {

int len = 0;

char c;

// 检查输入

if (!name) {

errno = EINVAL;

return -1;

}

// 拷贝数据，限制长度为 23

while ((c = get\_fs\_byte(name + len)) != '\0' && len < 23) {

stored\_name[len++] = c;

}

if (len >= 23) {

errno = EINVAL;

return -1;

}

stored\_name[len] = '\0';

return len;

}

int sys\_whoami(char \*name, unsigned int size) {

int len = 0;

if (!name || size <= strlen(stored\_name)) {

errno = EINVAL;

return -1;

}

while (stored\_name[len] != '\0' && len < size - 1) {

put\_fs\_byte(stored\_name[len], name + len);

len++;

}

put\_fs\_byte('\0', name + len);

return len;

}

然后在kernel/system\_call.s中把系统调用的总数从72改为74.

nr\_system\_calls=74.

**4.3 注册系统调用**

在 include/linux/sys.h中，将新系统调用加入系统调用表：

extern int sys\_iam(const char \*name);

extern int sys\_whoami(char \*name, unsigned int size);

fn\_ptr sys\_call\_table[] = {

// ...已有的系统调用

sys\_iam,

sys\_whoami

};

**4.4 修改 Makefile**

在 kernel/Makefile 中添加 who.o：

OBJS = ... who.o

**4.5 编写测试程序**

**iam.c**

#define \_\_LIBRARY\_\_

#include <unistd.h>

#include <stdio.h>

\_syscall1(int, iam, const char \*, name);

int main(int argc, char \*\*argv) {

if (argc < 2) {

printf("Usage: ./iam <name>\n");

return -1;

}

if (iam(argv[1]) < 0) {

perror("iam");

}

return 0;

}

**whoami.c**

#define \_\_LIBRARY\_\_

#include <unistd.h>

#include <stdio.h>

\_syscall2(int, whoami, char \*, name, unsigned int, size);

int main() {

char name[24];

if (whoami(name, 24) < 0) {

perror("whoami");

} else {

printf("Your name is: %s\n", name);

}

return 0;

}

**4.6 编译与运行**

1. **重新编译内核**：

cd ~/oslab/linux-0.11

make all

1. **启动虚拟机**：

./run

1. **编译用户程序**：

gcc -o iam iam.c -Wall

gcc -o whoami whoami.c -Wall

文本

描述已自动生成

1. **运行测试**：

./iam changtianos

./whoami

输出结果：

文本

描述已自动生成

**5. 实验结果**

通过上述步骤，实现了 iam() 和 whoami() 系统调用，并编写了测试程序验证功能。运行结果如下：

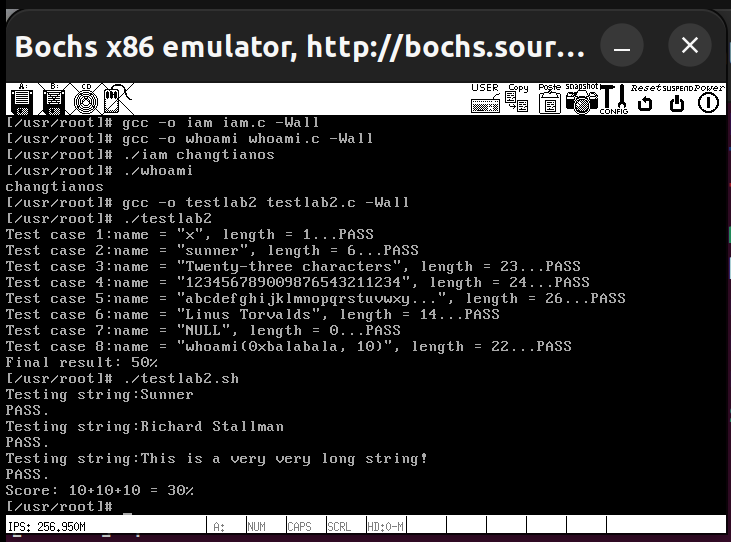
* 执行 ./iam changtianos，保存名字成功。
* 执行 ./whoami，输出changtianos成功。
* 运行评分脚本testlab2.c：

文本

描述已自动生成

评分脚本 testlab2.sh 显示得分 **50%**，满足实验要求。

* 运行评分脚本testlab2.sh：



评分脚本 testlab2.sh 显示得分 **30%**，满足实验要求。

综上，获得了全部的80%的分数。

**6. 问题回答**

**6.1 从 Linux 0.11 现在的机制看，它的系统调用最多能传递几个参数？**

在 Linux 0.11 中，系统调用的参数是通过 CPU 寄存器传递的，具体使用了以下寄存器：

* EAX：存放系统调用号。
* EBX、ECX、EDX：依次存放前三个参数。
* **Linux 0.11 中，最多可通过寄存器传递 3 个参数**。

如果需要传递 4 个及以上的参数，传统方法无法继续使用单纯的寄存器传递。

**6.2 你能想出办法来扩大这个限制吗？**

为了在 Linux 0.11 中突破 3 个参数的限制，可以采用以下方法：

**方法1：通过结构体传递参数**

* 将多个参数打包到一个结构体中，将结构体的地址作为单一参数传递给系统调用。
* 在系统调用函数中，通过结构体指针访问所有的参数。

**示例**： 假设系统调用 foo() 需要传递 5 个参数：

struct foo\_args {

int param1;

int param2;

int param3;

int param4;

int param5;

};

int sys\_foo(struct foo\_args \*args); // 通过结构体指针传递参数

在用户态代码中：

struct foo\_args args = {1, 2, 3, 4, 5};

foo(&args); // 将结构体地址作为单一参数传递

这种方法非常适合传递较多参数，同时避免了寄存器数量的限制。

**方法2：通过栈传递参数**

* 通过将额外的参数压入用户栈，系统调用的中断处理程序从用户栈中取出这些参数。
* 这需要对系统调用的汇编实现进行修改，使其支持从栈中获取参数。

**6.3 用文字简要描述向 Linux 0.11 添加一个系统调用 foo() 的步骤**

向 Linux 0.11 添加一个新系统调用 foo() 的步骤如下：

**步骤1：定义系统调用编号**

* 打开头文件 include/unistd.h。
* 为新系统调用 foo() 定义系统调用号，例如：

#define \_\_NR\_foo 74

**步骤2：实现系统调用函数**

* 在内核目录（例如 kernel/）中新建或修改源文件，例如 kernel/foo.c，编写 sys\_foo 函数：

int sys\_foo(int arg1, int arg2) {

// 系统调用的实现逻辑

return arg1 + arg2; // 示例返回值

}

**步骤3：注册系统调用**

* 在 kernel/sys\_call\_table.c 中，将新系统调用添加到系统调用表 sys\_call\_table：

extern int sys\_foo(int, int);

fn\_ptr sys\_call\_table[] = {

// ... 其他系统调用

sys\_foo, // 注册新系统调用

};

**步骤4：更新用户态接口**

* 在用户程序中使用 \_syscall2 宏提供用户接口（假设 foo() 有 2 个参数）：

#define \_\_LIBRARY\_\_

#include <unistd.h>

\_syscall2(int, foo, int, arg1, int, arg2);

**步骤5：重新编译内核**

* 进入内核源码目录，重新编译内核：

cd ~/oslab/linux-0.11

make all

**步骤6：测试系统调用**

* 编写用户态程序，调用新系统调用进行测试：

#include <stdio.h>

#define \_\_LIBRARY\_\_

#include <unistd.h>

\_syscall2(int, foo, int, arg1, int, arg2);

int main() {

int result = foo(10, 20);

printf("Result of foo: %d\n", result);

return 0;

}

* 在虚拟机中编译和运行测试程序：

gcc -o test\_foo test\_foo.c -Wall

./test\_foo

以上为全部测试步骤。