操作系统课程设计实验报告

> 姓名：霍志杰
>
> 学号：21009201175

1 基础题目

1.1 Shell编程

1.1.1 脚本解释

1. `#!/bin/bash`：这是一个shebang，告诉系统这个脚本应该使用bash shell来执行。
2. `if [ $# -ne 2 ]`：这是一个if语句，判断参数的数量是否不等于2（`$#`表示传递给脚本的参数个数，`-ne`表示“不等于”）。
3. `then echo "Usage:$0 filename r/w"`：如果参数数量不等于2，那么打印出脚本的正确使用方式。`$0`代表脚本自身的名称。
4. `exit 1`：退出脚本并返回状态码1，表示出现了错误。
5. `fi`：结束if语句。
6. `filename=$1` 和 `mode=$2`：这两行将传递给脚本的第一个和第二个参数赋值给变量filename和mode。
7. `if [ ! -f $filename ]`：这行检查filename变量所代表的文件是否存在。`-f`表示“文件”，`!`表示“非”或“不是”。`if [ $mode = "r" ]`：这些行检查mode变量是否等于"r"，表示读取文件。
8. `content=$(cat $filename)`：使用cat命令读取filename代表的文件内容，并将结果赋值给content变量。
9. `elif [ $mode = "w" ]`：如果mode变量等于"w"，表示写入文件。
10. `echo "175 MYFILE" > $filename`：将字符串"175 MYFILE"写入到filename变量代表的文件中。
11. 最后的 `else` 部分处理mode变量既不是"r"也不是"w"的情况。

1.1.2 脚本运行

可以通过以下命令来运行这个脚本：

```
bash scriptName.sh filename r/w
```

或者可以使脚本可执行，然后直接运行它：

```
chmod +x scriptName.sh
./scriptName.sh filename r/w
```

`bash`命令告诉系统用bash shell来运行脚本，`chmod +x`命令使脚本变为可执行，`./`命令执行当前目录下的脚本。

1.1.3 Bash流程

当你运行一个shell脚本时，Bash解释器会进行以下步骤：

1. 读取脚本文件。
2. 解析脚本文件中的命令。
3. 在一个子shell环境中顺序执行解析到的命令。
4. 如果命令执行成功，返回0，否则返回非0的错误代码。

1.1.4 write系统调用

这个脚本表面上并没有直接调用write系统调用。但是，shell的内建命令`echo`和重定向符号`>`在底层都是通过write系统调用实现输出的，所以间接上来说，这个脚本使用了write系统调用。

证据：

使用strace命令跟踪程序运行过程中的系统调用信号

```
strace -f -e write bash script.sh filename r/w
```

```
[pid  3197] write(1, "MYFILE 175\n", 11) = 11
[pid  3197] write(1, "\345\267\262\345\260\206MYFILE 175\345\206\231\345\205\245\346\226\207\344\273\266test"..., 43已将MYFILE 175写入文件test.txt中。
) = 43
[pid  3197] +++ exited with 0 +++
```

1.2 系统调用编程

1.2.1 题目1

1.2.1.1 程序中关键句的含义

- `open(file, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);` 这个系统调用尝试打开文件进行写操作。如果文件不存在，它将创建一个新文件。文件的权限被设置为用户读写。
- `write(fd, data, 18);` 这个系统调用将字符串 "21009201175 MYFILE" 写入文件。注意，字符串长度是18，不包括字符串的终止符 '\0'。
- `open(file, O_RDONLY);` 这个系统调用尝试打开文件进行读操作。
- `read(fd, buffer, 18);` 这个系统调用从文件读取18个字符并将其放入buffer中。
- `buffer[18] = '\0';` 这将字符串的终止符添加到读取的数据后面，使其可以作为字符串打印。

1.2.1.2 编译程序的命令及其含义：

使用 `gcc` 命令编译这个程序，命令如下：

```
gcc myprogram.c -o myprogram
```

这个命令的含义是：调用 GNU C 编译器 `gcc`，`-o myprogram` 用于指定输出的可执行文件名为 "myprogram"，而 "myprogram.c" 是源代码文件。

1.2.1.3 系统调用 `open`，`read`，`write` 和 C 语言函数 `fopen`，`fread`，`fwrite` 之间的关系：

- `open`，`read`，`write` 是底层操作系统提供的系统调用，它们直接与操作系统交互来打开、读取和写入文件。这些函数通常处理的是文件描述符，这是一个整数值，由操作系统用来识别特定进程正在访问的特定文件。

- `fopen`, `fread`, `fwrite` 是C语言的库函数，它们封装了这些底层系统调用，提供了一种更易于使用和理解的方式来操作文件。这些函数处理的是 `FILE` 指针，这是一种数据结构，它包含了关于文件的各种信息，包括文件描述符、缓冲区、位置指针等等。
- 所以说说，`fopen`, `fread`, `fwrite` 是高级的，而 `open`, `read`, `write` 是较低级的。高级函数提供了更多的抽象，通常更易于使用，但可能会牺牲一些效率和控制。较低级的函数提供了更直接的访问和更多的控制，但使用起来可能更复杂。

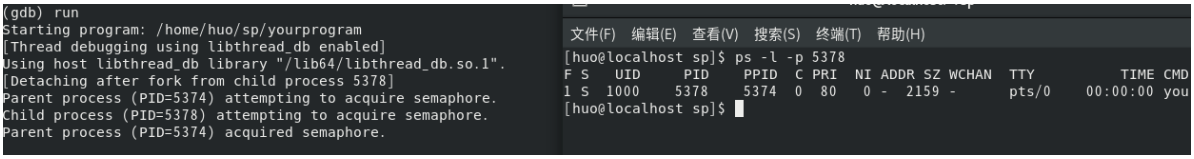
1.2.2 题目2

1.2.2.1关键句的含义

1. `sem_open(SEM_NAME, O_CREAT, 0644, 1);`：创建并初始化一个名为 "/semaphore_example" 的信号量，其初始值为 1。
2. `open(FILEPATH, O_CREAT | O_RDWR | O_TRUNC, S_IRUSR | S_IWUSR);`：创建并打开一个文件 "myfile.txt"，如果该文件已经存在则清空。
3. `fork();`：创建一个新的进程，新进程是原进程的复制品。
4. `sem_wait(mutex);`：使信号量值减1，如果值为0则挂起等待，直到信号量大于0。
5. `write(fd, WRITE1, sizeof(WRITE1));`：向文件中写入字符串 "175 PROC1 MYFILE1\n"。
6. `sem_post(mutex);`：使信号量值加1，如果有进程因此信号量挂起等待，则唤醒一个等待进程。
7. `sem_close(mutex)` and `sem_unlink(SEM_NAME);`：关闭并删除信号量 "/semaphore_example"。
8. `wait(NULL);`：使父进程等待直到子进程结束。

1.2.2.2实际操作证明

向程序添加必要输出后利用gdb和ps命令查看进程状态，结果如下



即证明了当一个进程占用semaphore后，另一个进程想要占用semaphore时进入睡眠。

1.3 内核编程

1.3.1 修改位置及原因

在添加系统调用的过程如下：

1. **在内核源代码的 kernel/sys.c 文件中添加处理逻辑的代码。**

```
SYSCALL_DEFINE2(os_exp,int, flag,int, number)
{
    //具体逻辑
}
```

原因：这个文件中的系统调用是Linux内核的一部分，用于处理用户空间到内核空间的交互。你在这里添加了你的系统调用的核心代码，这样当用户空间请求这个系统调用时，内核就知道如何执行这个请求。

2. **在 include/linux/syscalls.h 文件中声明系统调用。**
`asmlinkage long sys_os_exp(int flag,int number);`

原因：这个头文件是用于声明系统调用的地方，使得内核的其他部分可以引用和使用这个新添加的系统调用。

3. **在 arch/x86/entry/syscalls/syscall_64.tbl 文件中为新的系统调用分配了一个新的系统调用号。**

   `440 common  os_exp                    __x64_sys_os_exp`

   这个文件是系统调用号的查找表，它映射了系统调用号到相应的函数指针。你在这里添加了你的系统调用，分配了一个新的系统调用号，这样当用户空间使用这个号码请求系统调用时，内核就知道应该调用哪个函数。

1.3.2 系统调用关键语句

```
SYSCALL_DEFINE2(os_exp,int, flag,int, number)// SYSCALL_DEFINE2 宏，这是Linux 内核中定义系统调用的一种方式。SYSCALL_DEFINE2 的第一个参数是系统调用的名称，后面的2代表有2个参数。
{
    /**
    当 flag 等于 0 时，它返回 number 对 10 取模的结果，基本上就是 number 的个位数。
    当 flag 不等于 0 时，它返回 number 除以 10 之后再对 10 取模的结果，这实际上是返回 number 的十位数。
    */
    long int result = 0;
    if (flag == 0)
    {
        result = number%10;
    }else{
        result = number/10%10;
    }
    return result;//一般情况下系统调用的返回值类型是long 类型
}
```

1.3.3 编译内核并调用系统调用

然后，关于如何编译内核并调用自己的系统调用，你可以按照以下步骤进行：

1. **编译内核。**

   首先，你需要配置你的内核。在内核源代码的根目录下执行以下命令：

   ```
   make menuconfig
   ```

   然后，编译你的内核。执行以下命令：

   ```
   make -j8
   ```

   `-j8` 参数表示编译过程将使用8个线程。

2. **安装新内核。**

   编译完成后，执行以下命令：

   ```
   sudo make modules_install install
   ```

这将会安装新内核，并更新引导加载器配置。

3. **重启到新内核。**

   重启计算机，在引导加载器菜单中选择新内核。

4. **调用你的系统调用。**

   在用户空间程序中使用`syscall()`函数来调用新的系统调用。

```c
int main() {
    syscall(440, /* 参数 */);
    return 0;
}
```

1.4 驱动编程

1.4.1 关键语句含义

- `alloc_chrdev_region(&my_dev, 0, 1, DEVICE_NAME);` 这个函数是用来动态地为字符设备分配主设备号。主设备号被存储在`my_dev`中。
- `cdev_init(&my_cdev, &fops);` 这个函数用来初始化字符设备结构，`&fops`是一个指向设备的file_operations结构体的指针，这个结构体定义了设备支持的操作。
- `cdev_add(&my_cdev, my_dev, 1);` 这个函数向内核添加一个字符设备，设备的设备号是`my_dev`，设备的数量是1。
- `class_create(THIS_MODULE, CLASS_NAME);` 这个函数创建一个设备类，在`/sys/class/`目录下会创建一个相应的目录。
- `device_create(my_class, NULL, my_dev, NULL, DEVICE_NAME);` 这个函数在一个类下创建设备，设备名是`DEVICE_NAME`，设备号是`my_dev`。

1.4.2 驱动代码各个函数的调用时机

- `device_open`：当设备被打开时
- `device_release`：当设备被关闭时
- `device_read`：当设备被读取时
- `device_write`：当设备被写入时
- `device_ioctl` 当设备被ioctl调用时

1.4.3 编译和执行驱动程序

要编译驱动程序，需要在驱动源代码的同一目录下创建一个Makefile，内容如下

```
obj-m += mydevice.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

然后使用`make`命令来编译驱动程序。这将生成一个名为`mydevice.ko`的内核模块。

要加载驱动运行以下命令：

```
sudo insmod mydevice.ko
```

创建设备文件，输入以下命令：

```
sudo mknod /dev/mydevice c 主设备号 次设备号
```

测试程序核心代码：

```
fd = open("/dev/mydevice", O_RDWR);
read(fd, buf, sizeof(buf));
printf("Read from device: %s\n", buf);
write(fd, "Hello, world!", 13);
ioctl(fd, IOCTL_READ_OLD);
read(fd, buf, sizeof(buf));
```

2 中级题目

2.1 Shell编程

2.1.1 核心步骤说明

1. 创建新的脚本：这是通过 `cat > new_script.sh << 'EOF'` 这行代码开始的部分完成的。这行代码的作用是创建一个新的文件 `new_script.sh` 并将之后的所有内容（直到遇到 'EOF' 这个标志）写入这个新文件。这部分的内容就是新的脚本的代码。
2. 给新的脚本添加可执行权限：这是通过 `chmod +x new_script.sh` 这行代码完成的。`chmod` 是一个 Linux 命令，用于更改文件的权限。`+x` 表示添加执行权限。所以，这行代码的作用是给 `new_script.sh` 这个文件添加执行权限，使我们可以运行它作为一个脚本。
3. 执行新的脚本：这是通过 `./new_script.sh test.txt w` 这行代码完成的。`./` 是一个常用的方式来执行当前目录下的脚本或程序。这行代码的作用是运行 `new_script.sh` 这个脚本，并向它传递两个参数：`test.txt` 和 `w`。

2.2 系统调用编程

1.2.1 题目1

1.2.1.1 turn

算法中使用了共享内存来存储`turn`变量。共享内存是一种可以让多个进程访问的内存区域，它可以用于进程间的通信和同步。

因为进程间需要通过`turn`变量来协调它们的操作。

共享内存的创建和使用在下面的代码行中实现：

```
int *turn;
//mmap函数来创建一个共享内存区域
turn = mmap(NULL, sizeof *turn, PROT_READ | PROT_WRITE|MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
//turn指针设置为指向这个区域
*turn = 0;
```

1.2.1.2 关键代码

使用strict alternation算法的关键部分是如下的代码：

```
while (__sync_lock_test_and_set(turn, 1)) {} // Busy wait until it's this
process's turn
writeToFile(PROC1_MSG);
__sync_lock_release(turn); // Give turn to other process
```

`__sync_lock_test_and_set`和`__sync_lock_release`，这些是GCC提供的内建函数，用于实现低级别的同步原语。

代码首先调用了`__sync_lock_test_and_set`函数。这个函数试图将`turn`变量的值设为1，并返回原来的值。如果原来的值是1（表示另一个进程正在执行关键部分），那么这个函数会返回1，于是当前进程将进入忙等待状态，直到`turn`变量的值变为0。

然后，进程将执行关键部分（即写入文件）。完成关键部分后，进程调用`__sync_lock_release`函数将`turn`变量的值设回0，表示它已经完成关键部分，现在可以让其他进程执行关键部分了。

1.3 内核编程

1.3.1 题目1

1.3.1.1 内核锁函数的含义

在这个程序中使用mutex作为内核锁:

- `DEFINE_MUTEX(name)`：这是一个宏，用于在编译时定义并初始化一个名为`name`的互斥体。这个互斥体初始化为“未锁定”状态。
- `mutex_lock(mutex)`：这个函数用于获取一个互斥体的锁。如果互斥体已经被锁定，那么这个函数会导致当前的进程阻塞（即，停止运行），直到互斥体被解锁。如果互斥体未被锁定，那么这个函数会立即锁定互斥体并返回。
- `mutex_unlock(mutex)`：这个函数用于释放一个互斥体的锁。如果有其他的进程正在等待这个互斥体，那么其中一个进程将被唤醒，然后可以继续执行。

1.3.1.2 关于gOSE

如果`gOSE`是一个局部变量，那么它的生命周期就只在函数的执行过程中，即每次调用系统调用时，都会创建一个新的`gOSE`变量。在这种情况下，`gOSE`变量就只会被当前的进程使用，不会被其他的进程共享，`gOSE`变量的值在系统调用结束后就会丢失，不会在多次系统调用之间保持一致。因此不需要使用互斥体来保护`gOSE`变量的访问。

1.3.2 题目2

1.3.2.1 获取变量地址

以下是获取turn变量的虚拟地址和物理地址的步骤：

1. **运行应用程序得到进程的PID以及turn变量的虚拟地址。**：

```
printf("The PID is: %d\n", getpid());
printf("The virtual address of turn is: %p\n", (void*)turn);
```

这里，`getpid()`函数会返回当前进程的PID，`turn`变量本身就是一个指针，直接打印它的值就可以得到它的虚拟地址。

2. **修改并加载添加的内核模块**：书写新的内核模块用于获取真实的物理地址，将上一步得到的PID和虚拟地址替换到内核模块的代码中，重新编译和加载内核模块。

```
pid_t pid = 12345;
unsigned long va = (unsigned long)0x7f5e9b5fe000;
```

在这个内核模块中，我们通过遍历进程的虚拟内存区，找到对应的物理地址。

3. **查看物理地址**：最后，你可以通过`dmesg`命令查看内核模块的输出，从而得到物理地址。

1.3.2.2关键步骤

- **获取虚拟地址**：在用户空间，每个进程都有自己的独立虚拟地址空间。在这个地址空间中，每个变量都有一个唯一的虚拟地址。我们可以直接打印变量的地址，来获取这个虚拟地址。
- **获取物理地址**：物理地址是实际存在于物理内存中的地址。由于内存的管理是由操作系统负责的，用户空间的进程并不能直接获取物理地址。因此，我们需要使用内核模块来获取物理地址。在内核空间，我们可以通过遍历进程的虚拟内存区，然后根据页表将虚拟地址转换为物理地址。

总的来说，获取虚拟地址是在用户空间进行的，获取物理地址需要进入内核空间。这是因为物理地址是由操作系统管理的，用户空间的进程并不能直接访问。

1.4 驱动编程

*本节中请至少说明：i.**实现mmap**接口的相关代码 ii.**这些代码中关键代码的含义 iii. 测试mmap**接口的应用程序关键代码的含义*

1.4.1 实现mmap接口的相关代码

在驱动代码中，`device_mmap`函数就是mmap接口的实现。这是函数的定义：

```c
static int device_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long physical = virt_to_phys((void*)rwbuf);
    unsigned long offset = offset_in_page((unsigned long)rwbuf);
    vma->vm_start += offset;
    vma->vm_end -= offset;
    if (remap_pfn_range(vma, vma->vm_start, physical >> PAGE_SHIFT, vma->vm_end - vma->vm_start, vma->vm_page_prot)) {
        return -EAGAIN;
    }
    return 0;
}
```

1.4.2这些代码中关键代码的含义

- `virt_to_phys((void*)rwbuf)`：这行代码获取了 `rwbuf` 的物理地址。在驱动代码中，我们通常只有虚拟地址，但在进行内存映射时，我们需要物理地址。
- `offset_in_page((unsigned long)rwbuf)`：这行代码获取了 `rwbuf` 在其所在物理页中的偏移量。
- `vma->vm_start += offset;` 和 `vma->vm_end -= offset;`：这两行代码调整了请求映射的虚拟内存区域的开始和结束地址，以考虑 `rwbuf` 在其所在物理页中的偏移量。
- `remap_pfn_range(vma, vma->vm_start, physical >> PAGE_SHIFT, vma->vm_end - vma->vm_start, vma->vm_page_prot)`：这行代码将请求映射的虚拟内存区域与 `rwbuf` 所在的物理内存区域进行映射。

1.4.3测试mmap接口的应用程序关键代码的含义

以下是测试mmap接口的关键代码：

```c
mapped_data = mmap(0, MAX_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

这行代码请求将设备内存映射到用户空间。参数解释如下：

- 第一个参数（0）：这是期望的映射开始地址。传0表示让内核决定映射开始地址。
- 第二个参数（MAX_SIZE）：这是期望映射的内存大小。它应该与驱动代码中 `rwbuf` 的大小一致。
- 第三个参数（PROT_READ | PROT_WRITE）：这是期望的内存保护。这里请求可读可写的内存。
- 第四个参数（MAP_SHARED）：这表示创建一个共享映射，对映射区域的修改将直接写回物理设备。
- 第五个参数（fd）：这是需要映射的设备文件描述符。
- 第六个参数（0）：这是文件偏移，从文件的这个偏移位置开始映射。这里从文件开始位置开始映射。

3 个人总结

3.1 问题及解决

> 此次实验活动中遇到的问题众多，这里只展示两个最关键的重大问题，都是经过多次尝试后才得到正确解决

3.1.1 内核编译

问题：

在一开始做基础题时，感觉课程资料提供的参考内核太老，心想时代在进步要用新一点的，于是选择了是大众常用的Ubuntu20.04以及最新版本的6.3.5内核，结果发现可能是Ubuntu更新换代太多，加的乱七八糟的功能太多，网上的教程又良莠不齐，一编译一个报错。最让人头疼的还是：**一个问题在参考教程解决了以后又出现了另一个问题，而新的问题是由于在解决旧问题时改的配置所导致的。**

好几次等它编译等了一个小时结果给你来个报错。

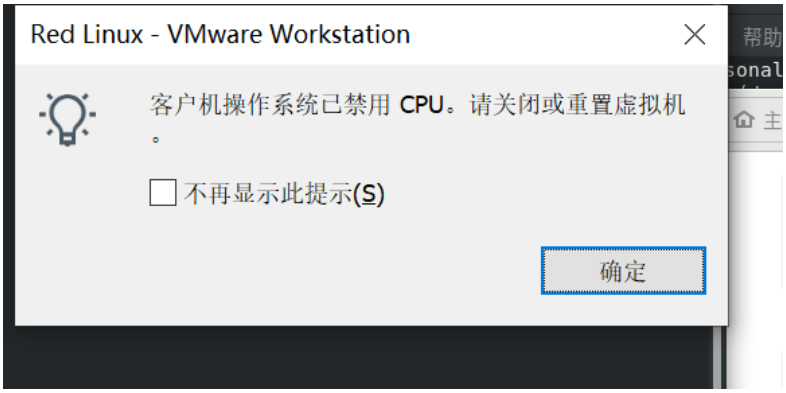
解决：

在经过一段实验寻找后，我决定放弃Ubuntu，而使用相对稳定的开源Linux后起之秀： Rocky Linux 。这也感谢一位在B站分享自己经验的学长。 Rocky Linux 的官网提供了明确的添加系统调用的教程，整个过程下来基本没有什么意外情况，让人十分舒服。

3.1.2 内存映射权限

问题：

在完成中级题目的驱动编程时，在运行程序后虚拟机直接报错重启：



经过多方搜索检查，这个错误提示通常意味着虚拟机试图执行某些操作，但这些操作被主机操作系统阻止了。

发现可能由于使用`mapped_data = mmap(0, MAX_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);`即虚拟机试图访问主机物理内存被一些安全机制强制阻止并退出虚拟机。这可能是由于虚拟机配置问题，也可能是由于Linux内核配置问题。

解决：

1. 在Vmware中设置虚拟机，启用VT-x/AMD-V，即启用硬件虚拟化
2. 修改Linux内核的配置。将内核配置文件中CONFIG_STRICT_DEVMEM ，这是一个安全选项，用于防止用户空间程序直接访问物理内存，但是我们确实需要访问物理内存，所以要禁用这个选项。然后重新编译安装内核。

3.2启发

3.2.1 技术

- 对于复杂项目的修改（往内核添加系统调用）一定要慎之又慎，仔细检查好每一处修改，哪怕是一个小符号的问题都可能导致整个项目在编译一小时后突然报错。
- 多参考官方权威文档，网上分享经验的人太多，鱼目混珠。可能按照别人的方法会带来更多的问题（点名批评csdn）。一切以官方文档为主。
- 加深了对操作系统的理解，揭开了内核神秘的面纱

3.2.2 学习做事方法

- 永远对技术保持敬畏，经过这次实验见识了操作系统内核的复杂与精妙，感叹前辈伟大的同时更加确信学无止境。
- 只有学的够深入，对知识理解的够到位才难以被chatGPT等大模型取代。本次实验中借助了不少chatGPT的成果，在简单逻辑上它的表现确实惊艳，但面临复杂逻辑涉及知识关联的情况下它还是能力有限。例如下面的回答：

> 我建议你查阅你的内核版本的相关文档，或者寻求专业的 Linux 内核开发者的帮助。你也可以考虑在一些专门的论坛或邮件列表上提问，例如 Linux 内核邮件列表（LKML）或 StackOverflow。在那里，有很多内核开发者和专家能够提供帮助。