添加系统调用

一、题目

添加一个系统调用（相关知识点参考实验一章的“系统调用添加”），该系统调用接受两个参数：参数 1：以整型数表示的自己学号的后 3 位；参数 2：flag，取值为 0 或 1，若为 0，

该系统调用的返回值为参数 1 的个位。若为 1。该系统调用的返回值为参数 1 的十位。

此外，加入内核互斥锁，使得两个进程在调用该系统调用时，能够做到互斥访问该系统调用。

i.自己所添加的系统调用的位置和修改点，以及为什么在这些位置上进行修改。

ii.自己系统调用关键语句的含义。

iii. 如何编译内核并调用自己的系统调用。

二、位置和修改点

1.定义系统调用号

文件：arch/x86/entry/syscalls/syscall_64.tbl

修改点：文件末尾添加

```
548 公共 peep_page sys_peep_page
```

原因：这里定义了系统调用的编号（548），名称（peep_page），以及内核中对应的系统调用处理函数（sys_peep_page）。

2. 声明系统调用

文件：include/linux/syscalls.h

修改点：文件末尾的#endif 前添加

```
asmlinkage long sys_peep_page(pid_t tar_pid_nr, unsigned long tar_addr,
 unsigned long my_addr);
```

原因：这里声明了系统调用函数 sys_peep_page，使得内核其他部分知道这个系统调用的存在。

3.定义系统调用号

文件：include/uapi/asm-generic/unistd.h

修改点：

```
#定义 __NR_peep_page 548

__SYSCALL(__NR_peep_page， sys_peep_page)

#定义 __NR_peep_page 548

__SYSCALL(__NR_peep_page， sys_peep_page)
```

原因：定义系统调用号 548，并关联到系统调用函数 sys_peep_page。

4.实现系统调用

文件：mmap.c
修改点：在文件中添加系统调用的实现

```
// 定义互斥锁
static DEFINE_MUTEX(peep_page_mutex);

static pte_t*
addr_to_pte(struct mm_struct *mm, unsigned long addr)
{
    return pte_offset_kernel(pmd_off(mm, addr), addr);
}

SYSCALL_DEFINE3(peep_page, pid_t, tar_pid_nr, unsigned long tar_addr
, unsigned long my_addr)
{
    int flag = (int)tar_addr;  // 使用 tar_addr 作为 flag 参数
    int param1 = (int)my_addr; // 使用 my_addr 作为 param1 参数
    int result;

    // 获取互斥锁
    mutex_lock(&peep_page_mutex);

    // 根据 flag 返回相应的值
    if (flag == 0) {
        result = param1 % 10;  // 返回个位
    } else if (flag == 1) {
        result = (param1 / 10) % 10;  // 返回十位
    } else {
        // 设置错误码并返回
        result = -EINVAL;  //会返回-1
        mutex_unlock(&peep_page_mutex);
        return result;
    }

    // 释放互斥锁
    mutex_unlock(&peep_page_mutex);

    return result;
}
```

原因：实现了系统调用的核心逻辑，并使用互斥锁保证系统调用的互斥访问。

三、关键语句的含义

1. 互斥锁定义和初始化

```
static DEFINE_MUTEX(peep_page_mutex);
```

含义：定义并初始化一个互斥锁，用于保护系统调用的互斥访问。

2. 系统调用定义

```
SYSCALL_DEFINE3(peep_page, pid_t, tar_pid_nr, unsigned long tar_addr, u
nsigned long my_addr)
```

含义：定义一个新的系统调用 peep_page，接受三个参数：tar_pid_nr、tar_addr、my_addr。

3. 参数解析和日志打印

```
1. int flag = (int)tar_addr;  // 使用 tar_addr 作为 flag 参数
2. int param1 = (int)my_addr; // 使用 my_addr 作为 param1 参数
3. printk("peep_page: running! param1: %d, flag: %d\n", param1, flag);
```

含义：解析传入的参数，将 tar_addr 解释为 flag，将 my_addr 解释为 param1，并打印日志以便调试。

4. 获取互斥锁

mutex_lock(&peep_page_mutex);
含义：在进入关键区域前获取互斥锁，保证系统调用的互斥访问。

5. 根据 flag 返回相应值

```
1. if (flag == 0) {
2.     result = param1 % 10;  // 返回个位
3. } else if (flag == 1) {
4.     result = (param1 / 10) % 10;  // 返回十位
5. } else {
6.     // 设置错误码并返回
7.     result = -EINVAL;
8.     mutex_unlock(&peep_page_mutex);
9.     return result;
10.}
```

含义：根据 flag 的值，返回 param1 的个位或十位。如果 flag 值非法，则返回错误码-EINVAL。
-flag 参数传递的详细说明
在 fine.c 中，调用了自定义的系统调用 peep_page，该系统调用需要两个参数：一个表示学

号后 3 位的整型数 param1，一个表示操作标志的整型数 flag。具体的传参过程如下：

1）在用户空间调用系统调用

```
int result0 = peep_page(param1, flag);
```

在 fine.c 中，peep_page 函数被调用并传入两个参数 param1 和 flag。其中，param1 是整型数 894，flag 是整型数 0 或 1。

2）peep_page 函数定义

```
1. static inline long peep_page(int param1, int flag)
2. {
3.     return syscall(__NR_peep_page, param1, flag);
4. }
```

该函数内部使用 syscall 函数来调用系统调用 peep_page，__NR_peep_page 是系统调用号 548。参数 param1 和 flag 被传递给内核空间的系统调用处理函数。

3）内核空间系统调用处理函数

```
SYSCALL_DEFINE3(peep_page, pid_t, tar_pid_nr, unsigned long tar_addr, u
nsigned long my_addr)
```

在内核中，系统调用处理函数 sys_peep_page 接收三个参数：tar_pid_nr、tar_addr、my_addr。在用户空间传入的 param1 和 flag 分别对应于 my_addr 和 tar_addr。通过参数转换：

```
1. int flag = (int)tar_addr;  // 使用 tar_addr 作为 flag 参数
2. int param1 = (int)my_addr; // 使用 my_addr 作为 param1 参数
```

flag 和 param1 就被正确传递并使用。

6. 释放互斥锁并返回结果

```
1. mutex_unlock(&peep_page_mutex);
2. return result;
```

含义：在关键区域结束后释放互斥锁，并返回计算结果。

四、编译内核并调用自己的系统调用

1.编译内核

确保所有修改都已经保存。

（1）进入内核源代码目录：

```
cd /path/to/kernel/source
```

（2）配置内核：

```
make menuconfig
```

确保新的系统调用已经包含在配置中。

（3）编译内核和模块:

```
make -j$(nproc)
make modules_install
make install
```

（4）重启系统并选择新编译的内核。

（5）如遇到内核空间不够，需要及时清理旧内核和不需要的日志文件而不是扩容。

2. 调用系统调用

（1）编写用户空间程序 fine.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "peep_page.h"

int main() {
    int param1 = 894; // 以整型数表示的学号后3位
    int flag = 0;

    int result0 = peep_page(param1, flag);
    if (result0 == -1) {
        perror("sys_peep_page with flag 0 failed");
    } else {
        printf("Result with flag 0: %d\n", result0); // 应该输出学号后3位的个位
    }

    flag = 1;
    int result1 = peep_page(param1, flag);
    if (result1 == -1) {
        perror("sys_peep_page with flag 1 failed");
    } else {
        printf("Result with flag 1: %d\n", result1); // 应该输出学号后3位的十位
    }

    return 0;
}
```

定义 param1 为 894，表示学号后 3 位。

定义 flag 为 0，表示将要获取 param1 的个位数。
调用 peep_page(param1, flag)，如果返回-1，打印错误信息；否则打印返回值（个位数）。
将 flag 设为 1，表示将要获取 param1 的十位数。
再次调用 peep_page(param1, flag)，如果返回-1，打印错误信息；否则打印返回值（十位数）。
（2）编写头文件 peep_page.h

```
#ifndef PEEP_PAGE_H
#define PEEP_PAGE_H

#include <unistd.h>
#include <sys/syscall.h>

#define __NR_peep_page 548

static inline long peep_page(int param1, int flag)
{
    return syscall(__NR_peep_page, param1, flag);
}

#endif
```

-为什么需要 peep_page.h

peep_page.h 是一个头文件，定义了用户空间如何调用我们自定义的系统调用。它的作用如下：

```
#define __NR_peep_page 548
```

它定义了系统调用的编号 548，确保用户空间程序知道调用哪个系统调用号。
内联函数 peep_page：

```
static inline long peep_page(int param1, int flag)
{
    return syscall(__NR_peep_page, param1, flag);
}
```

该内联函数封装了 syscall 系统调用，方便用户空间程序调用自定义的系统调用，而不必直接使用 syscall 函数。这样可以提高代码的可读性和可维护性。
头文件保护：

```
#ifndef PEEP_PAGE_H
#define PEEP_PAGE_H
// ...
#endif
```

头文件保护防止头文件被重复包含，引起编译错误。
peep_page.h 头文件通过定义系统调用号和封装系统调用的内联函数，为用户空间程序提供了方便的接口，以便调用自定义的系统调用。
（3）编译运行用户空间程序

```
1. gcc -g -o fine fine.c
2. ./fine
```

输出结果如图所示：

```
hacker@ok:~/桌面$ ./fine
Result with flag 0: 4
Result with flag 1: 9
```