操作系统 Lab 03 内核编程

姓名: 雍崔扬
学号: 21307140051

Task 1: 新增系统调用

本实验我们了解内核如何实现和提供一个系统调用.
以 `getpid` 为例:

```
/**
* sys_getpid - return the thread group id of the current process
*
* Note, despite the name, this returns the tgid not the pid. The tgid and
* the pid are identical unless CLONE_THREAD was specified on clone() in
* which case the tgid is the same in all threads of the same group.
*
* This is SMP safe as current->tgid does not change.
*/
SYSCALL_DEFINE0(getpid)
{
    return task_tgid_vnr(current);
}
```

其中 `SYSCALL_DEFINE0` 是一个宏，用来定义一个 0 个参数的系统调用.
这个宏会展开成一个函数，函数名为 `sys_getpid`，并且返回值为 `long`
Linux 使用 `sys_call_table` 的数组来存放所有系统调用的函数指针.
每一个系统调用都对应一个系统调用号，一旦分配就不能有任何改变，否则就会导致兼容性问题.
即使随着 Linux 内核的版本迭代，某些系统调用被删除，其号码也不能被重新回收利用，
而是将一个 "未实现" 系统调用 `sys_ni_syscall()` 放在 `sys_call_table` 对应的位置.
这个系统调用除了返回 `-ENOSYS` 之外什么也不做.

(1) 添加系统调用

在 Linux 内核源码目录下，
首先我们在 `linux/arch/x86/entry/syscalls/syscall_32.tbl` 文件末尾添加新的系统调用号和系统调用名称:

```
222 i386 my_syscall sys_my_syscall
```

其次在 `linux/include/linux/syscalls.h` 文件中添加我们的系统调用的声明:

```
asmlinkage long sys_my_syscall(void);
```

然后添加我们系统调用的实现，可以将其放在 `linux/kernel/sys.c` 中:

```
SYSCALL_DEFINE0(my_syscall)
{
    printk(KERN_INFO "Hello, this is my syscall!\n");
    return 0;
}
```

最后在 `linux` 目录下使用命令行删除内核镜像，重新编译内核后，就可以使用用户态程序调用新的系统调用了:

```
rm arch/x86/boot/bzImage
cd tools/labs
make zImage
```

(2) 测试系统调用

由于工作目录 `linux/tools/labs/` 下的 `skels` 是一个共享文件夹，
故我们在主机编译的用户态程序可以直接在虚拟机中运行.
我们可以在 `skels` 目录下新建一个名为 `lab3` 的目录，然后在其中新建一个名为 `test_syscall.c` 的文件:

```
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

#define __NR_my_syscall 222

int main()
{
    long ret = syscall(__NR_my_syscall);
    printf("Return value: %ld\n", ret);
    return 0;
}
```

编译这个程序:

```
# -m32 表示编译为 32 位程序
# --static 表示静态链接，不依赖动态库
gcc -m32 --static test_syscall.c -o test_syscall
```

并在虚拟环境 `qemux86` (通过在 `linux/tools/labs` 目录下 `make console` 进入) 中运行这个程序:

```
qemux86 login: root
random: crng init done

root@qemux86:~# ./skels/lab3/test_syscall
Hello, this is my syscall!
Return value: 0
root@qemux86:~#
```

我们能够看到终端输出 `Hello, this is my syscall!`，并且用户态打印出 `Return value: 0`

```
qemux86 login: root
random: crng init done
root@qemux86:~# ./skels/lab3/test_syscall
Hello, this is my syscall!
Return value: 0
root@qemux86:~#
```

`Ctrl + A` 再按 `X` 退出 `qemu` 虚拟环境.

Task 2: 统计系统调用次数

进入系统调用的代码位于 `linux/arch/x86/entry/` 目录下.
在本次实验，我们不需要关心进入系统调用的具体细节 (即该目录下的几个 `entry.S` 文件)
而只需要关注 `common.c` 文件.
在这个文件中，有着根据系统调用号调用系统调用的代码:

```
/* Invoke a 32-bit syscall.  Called with IRQs on in CONTEXT_KERNEL. */
static __always_inline void do_syscall_32_irqs_on(struct pt_regs *regs, unsigned
int nr)
{
    if (likely(nr < IA32_NR_syscalls))
    {
        nr = array_index_nospec(nr, IA32_NR_syscalls);
        syscall_counts[nr]++;
        regs->ax = ia32_sys_call_table[nr](regs);
    }
}
```

我们的任务是在这个函数中添加统计系统调用次数的代码:

- ① 在 `linux/arch/x86/entry/common.c` 文件中添加一个全局数组 `syscall_counts`，用于统计系统调用次数.
  这个数组初始化为 `0` ，且大小是系统调用数量 `IA32_NR_syscalls`

  ```
  /*
   * Define a global array to keep track of the number of times each system
  call is invoked.
   * The size of the array is IA32_NR_syscalls, initialized to 0.
   */
  static unsigned long syscall_counts[IA32_NR_syscalls] = {0};
  ```

- ② 在 `do_syscall_32_irqs_on` 函数中，每次调用系统调用时，将对应该系统调用号的计数加一.

  ```
  /*
   * Invoke a 32-bit syscall.  Called with IRQs on in CONTEXT_KERNEL.
   */
  static __always_inline void do_syscall_32_irqs_on(struct pt_regs *regs,
  unsigned int nr)
  {
      /* Check if the system call number is within the valid range */
      if (likely(nr < IA32_NR_syscalls))
      {
  ```

```
        /* Use a safe array indexing function to prevent out-of-bounds
access */
        nr = array_index_nospec(nr, IA32_NR_syscalls);

        /* Increment the count for the corresponding system call */
        syscall_counts[nr]++;

        /* Call the appropriate system call and store the return value in
the ax register */
        regs->ax = ia32_sys_call_table[nr](regs);
    }
}
```

- ③ 删除 Task 1 中我们在 `kernel/sys.c` 中添加的系统调用代码.
- ④ 在 `do_syscall_32_irqs_on` 函数的附近添加一个新的系统调用，用于获取系统调用的调用次数.
  其声明为 `SYSCALL_DEFINE1(my_sysall, int, nr)`，返回值是系统调用号的调用次数.

```
/*
 * Define a new system call my_syscall that takes one parameter nr,
 * which is used to retrieve the count of calls for the specified system
call.
 */
SYSCALL_DEFINE1(my_syscall, int, nr)
{
    /* Check if the input parameter is valid */
    if (nr < 0 || nr >= IA32_NR_syscalls)
        return -EINVAL;

    /* Return the count of the corresponding system call */
    return syscall_counts[nr];
}
```

- ⑤ 在 `linux` 目录下重新编译内核:

```
rm arch/x86/boot/bzImage
cd tools/labs
make zImage
```

修改 Task 1 中的用户态程序 `test_syscall.c`:

```
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

#define __NR_my_syscall 222  // Define the system call number for my_syscall

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <syscall_number>\n", argv[0]);
        return 1;
    }
```

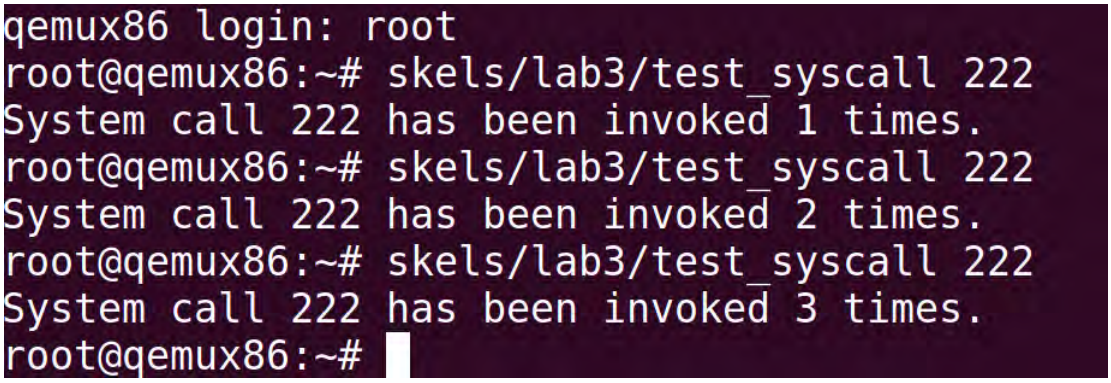
```c
    // Convert command-line argument to an integer
    int syscall_number = atoi(argv[1]);

    // Call my_syscall with the specified syscall number
    long ret = syscall(__NR_my_syscall, syscall_number);

    if (ret < 0)
    {
        perror("syscall");  // Print error if syscall fails
        return 1;
    }

    // Print the invocation count
    printf("System call %d has been invoked %ld times.\n", syscall_number,
ret);
    return 0;
}
```

编译这个程序:

```c
# -m32 表示编译为 32 位程序
# --static 表示静态链接，不依赖动态库
gcc -m32 --static test_syscall.c -o test_syscall
```

并在虚拟环境 `qemux86` (通过在 `linux/tools/labs` 目录下 `make console` 进入) 中运行这个程序.
重复统计 `222` 号系统调用的调用次数，能看到每次获取的调用次数都会增加 `1`:

```
qemux86 login: root

root@qemux86:~# skels/lab3/test_syscall 222

System call 222 has been invoked 1 times.

root@qemux86:~# skels/lab3/test_syscall 222

System call 222 has been invoked 2 times.

root@qemux86:~# skels/lab3/test_syscall 222

System call 222 has been invoked 3 times.

root@qemux86:~#
```



`Ctrl + A` 再按 `X` 退出 `qemu` 虚拟环境.

Task 3: kmalloc 和 kfree

内核编程与用户空间编程有很大不同.
内核是一个独立的实体，不能使用用户态的库 (甚至不能使用 `libc`)
因此通常的用户态函数 (`printf`、`malloc`、`free`、`open`、`read`、`write`、`memcpy` 和 `strcpy` 等) 都不再可用.
内核编程基于一套全新的独立的 API，也即内核向自己提供的一系列帮助函数.

- **字符串/内存处理:**
  `strcpy()`, `strncpy()`, `strlcpy()`, `strcat()`, `strncat()`, `strlcat()`, `strcmp()`, `strncmp()`, `strnicmp()`, `strchr()`, `strnchr()`, `strrchr()`, `strstr()`, `strlen()`, `memset()`, `memmove()`, `memcmp()`
  它们的声明位于 `include/linux/string.h`
- **打印:**
  使用 `printk()` 系列函数
- **内存分配与释放:**
  在 Linux 内核中，我们使用 `kmalloc()` 来分配空间，使用 `kfree()` 释放内存.
  其中 `kmalloc()` 的函数原型是:

  ```
  void *kmalloc(size_t size, gfp_t flags);
  ```

  第一个参数 `size_t size` 用于指定分配区域的大小 (以字节为单位)
  第二个参数 `gfp_t flags` 指定应如何进行分配，其最常用的值为 `GFP_KERNEL` 和 `GFP_ATOMIC` (前者代表当前进程在执行 `kmalloc()` 中途可被内核打断，后者确保当前进程在执行 `kmalloc()` 中途可被内核打断)

生成代码框架:

```
LABS=kernel_api/3-memory make skels
```

我们需要分配 4 个 `struct task_info` 类型的结构体并初始化它们 (在 `memory_init` 中),
然后打印并释放它们 (在 `memory_exit` 中)

- ① 在 `struct task_info` 结构体分配内存并初始化其字段:
  - 将 `pid` 字段初始化为传递的参数 `PID`
  - 将 `timestamp` 字段初始化为 `jiffies` 变量的值，该变量保存自系统启动以来发生的时钟滴答数 (tick)
    时钟滴答是内核用于计时的一个基本单位，它的长度取决于系统硬件和内核配置 (通常是 $1/100$ 秒或 1 毫秒)

```
static struct task_info *task_info_alloc(int pid)
{
    struct task_info *ti;

    /* TODO 1: allocated and initialize a task_info struct */
    // Allocate memory for a task_info struct
    ti = kmalloc(sizeof(*ti), GFP_KERNEL);
```

```
    if (!ti)
    {
        printk(KERN_ERR "Failed to allocate memory for task_info\n");
        return NULL;
    }

    // Initialize the fields
    ti->pid = pid;
    ti->timestamp = jiffies; // Initialize timestamp with current jiffies
value

    return ti;
}
```

- ② 为当前进程、父进程、下一个进程、下一个进程的下一个进程分配 `struct_task_info`，并包含以下信息:
  - 当前进程的 `PID` 可通过在 `task_struct` 结构体中获取 (`pid`)，该结构体由 `current` 宏返回.

    ```
    /* TODO 2: call task_info_alloc for current pid */
    // Allocate and initialize task_info for current PID
    ti1 = task_info_alloc(current->pid);
    ```

  - 当前进程的父进程的 `PID` 可通过在 `task_struct` 结构体中获取 (`parent`)
    注意，`task_struct` 结构体包含两个字段来指定任务的父进程:
    - `real_parent` 指向创建任务的进程，或者如果父进程完成其执行，则指向 `PID` 为 `1` 的进程 (`init`)
    - `parent` 指向当前任务的父进程 (任务执行完成时将报告的进程)
      通常来说这两个字段的值是相同的，但在某些情况下它们会有所不同，例如使用 `ptrace` 系统调用时.

    ```
    /* TODO 2: call task_info_alloc for parent PID */
    // Allocate and initialize task_info for parent PID
    ti2 = task_info_alloc(current->real_parent->pid);
    ```

  - 进程列表中的下一个进程的 `PID` 可通过调用 `next_task` 宏得到，该宏返回指向下一个进程的 `task_struct*` 指针

    ```
    /* TODO 2: call task_info alloc for next process PID */
    // Allocate and initialize task_info for next process PID
    ti3 = task_info_alloc(next_task(current)->pid);
    ```

  - 进程列表中的下下一个进程的 `PID` 可通过调用 `next_task` 宏两次

    ```
    /* TODO 2: call task_info_alloc for next process of the next process */
    // Allocate and initialize task_info for the next process of the next
    process
    ti4 = task_info_alloc(next_task(next_task(current))->pid);
    ```

合并得到 `memory_init` 函数:

```
static int memory_init(void)
```

```
{
    /* TODO 2: call task_info_alloc for current pid */
    // Allocate and initialize task_info for current PID
    ti1 = task_info_alloc(current->pid);

    /* TODO 2: call task_info_alloc for parent PID */
    // Allocate and initialize task_info for parent PID
    ti2 = task_info_alloc(current->real_parent->pid);

    /* TODO 2: call task_info alloc for next process PID */
    // Allocate and initialize task_info for next process PID
    ti3 = task_info_alloc(next_task(current)->pid);

    /* TODO 2: call task_info_alloc for next process of the next process */
    // Allocate and initialize task_info for the next process of the next
process
    ti4 = task_info_alloc(next_task(next_task(current))->pid);

    return 0;
}
```

- ③ 使用 `printk` 输出上述四个结构体的两个字段: `pid` 和 `timestamp`

```
/* TODO 3: print ti* field values */
// Print ti* field values
if (ti1)
    printk(KERN_INFO "Current PID: %d, Timestamp: %lu\n", ti1->pid, ti1-
>timestamp);
if (ti2)
    printk(KERN_INFO "Parent PID: %d, Timestamp: %lu\n", ti2->pid, ti2-
>timestamp);
if (ti3)
    printk(KERN_INFO "Next PID: %d, Timestamp: %lu\n", ti3->pid, ti3-
>timestamp);
if (ti4)
    printk(KERN_INFO "Next of Next PID: %d, Timestamp: %lu\n", ti4->pid,
ti4->timestamp);
```

- ④ 使用 `kfree` 释放这些结构体占用的内存

```
/* TODO 4: free ti* structures */
// Free ti* structures
kfree(ti1);
kfree(ti2);
kfree(ti3);
kfree(ti4);
```

③④ 合并得到 `memory_exit` 函数:

```
static void memory_exit(void)
{
    /* TODO 3: print ti* field values */
    // Print ti* field values
    if (ti1)
```

```
        printk(KERN_INFO "Current PID: %d, Timestamp: %lu\n", ti1->pid, ti1->timestamp);
    if (ti2)
        printk(KERN_INFO "Parent PID: %d, Timestamp: %lu\n", ti2->pid, ti2->timestamp);
    if (ti3)
        printk(KERN_INFO "Next PID: %d, Timestamp: %lu\n", ti3->pid, ti3->timestamp);
    if (ti4)
        printk(KERN_INFO "Next of Next PID: %d, Timestamp: %lu\n", ti4->pid, ti4->timestamp);

    /* TODO 4: free ti* structures */
    // Free ti* structures
    kfree(ti1);
    kfree(ti2);
    kfree(ti3);
    kfree(ti4);
}
```

在 `linux/tools/labs` 目录下编译内核模块:

```
Linux:~/Desktop/OS/Lab2/linux/tools/labs$ make build
```

加载和卸载模块 `memory.ko`:

```
qemux86 login: root
root@qemux86:~# cd skels/kernel_api/3-memory
root@qemux86:~/skels/kernel_api/3-memory# insmod memory.ko
root@qemux86:~/skels/kernel_api/3-memory# lsmod | grep memory
memory 16384 0 - Live 0xe085c000 (O)
root@qemux86:~/skels/kernel_api/3-memory# rmmod memory.ko
Current PID: 652, Timestamp: 1406623
Parent PID: 630, Timestamp: 1406623
Next PID: 0, Timestamp: 1406623
Next of Next PID: 1, Timestamp: 1406623
root@qemux86:~/skels/kernel_api/3-memory#
```

运行截图:

```
qemux86 login: root
root@qemux86:~# cd skels/kernel_api/3-memory
root@qemux86:~/skels/kernel_api/3-memory# insmod memory.ko
root@qemux86:~/skels/kernel_api/3-memory# lsmod | grep memory
memory 16384 0 - Live 0xe085c000 (O)
root@qemux86:~/skels/kernel_api/3-memory# rmmod memory.ko
Current PID: 652, Timestamp: 1406623
Parent PID: 630, Timestamp: 1406623
Next PID: 0, Timestamp: 1406623
Next of Next PID: 1, Timestamp: 1406623
root@qemux86:~/skels/kernel_api/3-memory# exit
```

Task 4: list

内核在 `linux/lnclude/linux/list.h` 中定义了链表以及操作链表的函数:

- 链表的定义如下:

```
struct list_head {
    struct list_head *next, *prev;
};
```

- 进程结构体 `task_struct` 中就有一个链表，用于连接所有进程:

```
struct task_struct {
    ...
    struct list_head tasks;
    ...
};
```

- 链表的操作函数例如:
  `LIST_HEAD(name)` 定义一个链表头 (哨兵)
  `INIT_LIST_HEAD(struct list_head *list)()` 初始化一个链表
  `list_add(struct list_head *new, struct list_head *head)` 将一个新节点插入到链表头部
  `list_add_tail(struct list_head *new, struct list_head *head)` 将一个新节点插入到链表尾部
  `list_del(struct list_head *entry)` 从链表中删除一个节点
  `list_entry(ptr, type, member)` 获取节点指针对应的结构体指针
  `list_for_each(pos, head)` 遍历链表
  `list_for_each_safe(pos, n, head)` 遍历链表，但是可以在遍历过程中删除节点

生成代码框架:

```
LABS=kernel_api/4-list make skels
```

我们需要将 Task 3 中的四个结构体添加到一个链表里.
链表将在模块加载时由 `task_info_add_for_current` 函数构建，
并在 `list_exit` 函数和 `task_info_purge_list` 函数中打印和删除.

- ① 完成 `task_info_add_to_list` 函数，以分配一个 `struct task_info` 结构体并将其添加到链表中

```
static void task_info_add_to_list(int pid)
{
    struct task_info *ti;

    /* TODO 1: Allocate task_info and add it to list */
    ti = task_info_alloc(pid); // Allocate task_info structure
    if (ti)
    {
        list_add(&ti->list, &head); // Add to the head of the list
    }
}
```

- ② 完成 `task_info_purge_list` 函数，使用 `list_for_each_entry_safe` 宏删除链表中的所有元素.

```
static void task_info_purge_list(void)
{
    struct list_head *p, *q;
    struct task_info *ti;

    /* TODO 2: Iterate over the list and delete all elements */
    list_for_each_safe(p, q, &head)
    {
        ti = list_entry(p, struct task_info, list); // Get the task_info
struct
        list_del(p); // Remove from the list
        kfree(ti); // Free the allocated memory
    }
}
```

- ③ 在 `linux/tools/labs` 目录下编译内核模块:

```
Linux:~/Desktop/OS/Lab2/linux/tools/labs$ make build
```

- ④ 按照内核显示的消息加载和卸载模块:

```
qemux86 login: root

root@qemux86:~# cd skels/kernel_api/4-list

root@qemux86:~/skels/kernel_api/4-list# insmod list.ko

root@qemux86:~/skels/kernel_api/4-list# lsmod | grep list

list 16384 0 - Live 0xe085c000 (O)

root@qemux86:~/skels/kernel_api/4-list# rmmod list.ko

before exiting: [

(1, 1297351)

(0, 1297351)

(624, 1297351)

(626, 1297351)

]

root@qemux86:~/skels/kernel_api/4-list#
```

四对 `(pid, timestamp)` 表示我们成功地为当前进程、其父进程、下一个进程和下下一个进程分配了 `task_info` 结构体，
并在链表中正确存储了这些信息运行截图:

```
qemux86 login: root
root@qemux86:~# cd skels/kernel_api/4-list
root@qemux86:~/skels/kernel_api/4-list# insmod list.ko
root@qemux86:~/skels/kernel_api/4-list# lsmod | grep list
list 16384 0 - Live 0xe085c000 (O)
root@qemux86:~/skels/kernel_api/4-list# rmmod list.ko
before exiting: [
(1, 1297351)
(0, 1297351)
(624, 1297351)
(626, 1297351)
]
root@qemux86:~/skels/kernel_api/4-list# exit
```