



华南理工大学  
South China University of Technology

# 课程设计报告书

题目：linux 内核实验

学 院     计算机科学与工程学院

专 业     计算机科学与技术

学生姓名     黄天晟

学生学号     201920141128

指导教师     吴一民

课程编号     S0812011

课程学分     2

起始日期

<p>教师评语</p>	<p>教师签名： 日期：</p>
<p>成绩评定</p>	

## 摘 要

本文是高级操作系统的课程实验报告。在本实验中，主要完成三个内核代码的编写以及调试任务。其中两个任务需要设计新的系统调用，并编译进新的 linux 内核中。另外一个任务则需要完成一个内核模块来做一些简单的文件读写操作，再动态编入内核之中。实验基于 ubuntu 18.04（内核版本为 2.15）。最终形成一个能正常运作且含有实验要求功能的内核，并打包成一个内核补丁。

**关键词：** linux 内核开发，系统调用，内核模块

# 目 录

摘要 .....	I
插图目录 .....	III
第一章 实验任务一 .....	1
1.1 题目要求 .....	1
1.2 实现过程 .....	1
1.3 实验结果 .....	6
1.4 运行说明 .....	7
第二章 实验任务二 .....	8
2.1 题目要求 .....	8
2.2 实现过程 .....	8
2.3 实验结果 .....	10
2.4 运行说明 .....	10
第三章 实验任务三 .....	11
3.1 题目要求 .....	11
3.2 实验过程 .....	11
3.3 实验结果 .....	13
3.4 运行说明 .....	14
第四章 内核编译安装说明 .....	15
第五章 总结 .....	16
参考文献 .....	17
致谢 .....	18

## 插图目录

1-1	arch/x86/entry 目录结构 . . . . .	2
1-2	menuconfig 命令界面 . . . . .	4
1-3	任务一实验结果 . . . . .	6
2-1	实验二实验结果 . . . . .	10
2-2	make module . . . . .	10
2-3	install module . . . . .	10
3-1	任务三实验结果 . . . . .	13
3-2	任务三实验结果验证 . . . . .	14
3-3	任务三打印内核日志 . . . . .	14



## 第一章 实验任务一

### 1.1 题目要求

修改 `system_call()`，使内核能够记录每一个系统调用被使用的次数。同时，为了使应用程序能够查询到这些数据，本实验要求实现两个系统调用，一个供应用程序来查询某个特定系统调用被使用的次数，另一个系统调用将系统调用计数清零。编制一个用户态程序调用你所增加的这两个系统调用，统计在一段时间内各系统调用被调用的次数。（注：使用的是 4.15 内核版本）

### 1.2 实现过程

为了达到记录每个调用的被调用次数，我们在 `/kernel/sys.c` 文件下定义一个 `long` 类型的数组变量，数组的大小设定为系统调用的总数（定义在宏 `NR_syscalls`）。注意，在 4.15 版本的内核中，`NR_syscalls` 宏并没有在 `asm/unistd.h` 定义，而是在 `asm/asm-offsets.h` 中。在原版本的 `sys.c` 中是没有引入 `asm/asm-offsets.h` 的，所以我们需要在文件头引入这个头文件。这里也困扰了我不少时间，网上教程多是针对低版本内核（3.0 以下），在那些版本的内核中 `unistd.h` 定义了 `NR_syscalls`，所以他们不需要像现在那样引入新的头文件。在定义了该记录数组后，需要使用 `EXPORT_SYMBOL()` 向整个内核公开这个数组。这是因为我们后期需要在其他的文件中对这个数组进行自增操作。

定义完记录数组后，我们先不考虑对它进行自增。先完成两个供应用程序进行查询和置零操作的系统调用。其中 `sys_check_count` 是查询数组的计数值，它从用户空间获得两个参数，一个记录需要查询的调用号，另一个用户空间的内存用来记录查询结果。代码非常简单，直接从全局数组获得某一个调用的计数值，然后使用 `copy_to_user()` 函数将查询结果粘贴到用户内存。对于第二个置零操作，我们用 `sys_reset_count()` 来实现。实现原理也很简单，直接使用用户传来的调用号对记录数组置零，如果成功则返回 0，失败则返回 -1。这里注意内核中日志的打印应该使用 `printk()` 函数。

```
1 ... (前省略)
2 #include <generated/utsrelease.h>
3 #include <linux/uaccess.h>
4 #include <asm/io.h>
5 #include <asm/unistd.h>
6 #include <asm/asm-offsets.h>
7 ... (前省略)
8 long hts_count [NR_syscalls];
9 EXPORT_SYMBOL(hts_count);
```

```
linux@ubuntu:/u01/my_source/arch/x86/entry$ ls
calling.h  common.o.ur-safe  entry_64_compat.o.ur-safe  entry_64.o.ur-detected  entry_64.S  syscall_32.c  syscalls  thunk_64.S
common.c  entry_32.S        entry_64_compat.S        entry_64.o.ur-safe      Makefile    syscall_64.c  thunk_32.S  vdso
```

图 1-1 arch/x86/entry 目录结构

```

11 asmlinkage long sys_check_count(long call_number, char __user *buffer){
12     if (call_number < NR_syscalls) {
13         long count=hts_count [call_number];
14         printk ("INFO The count of<%d> is %d\n", call_number, count);
15         copy_to_user( buffer , &hts_count[call_number], sizeof(long));
16     }
17     else {
18         printk ("NO SUCH CALL!\n");
19         return -1;
20     }
21     return 0;
22 }
23
24 asmlinkage long sys_reset_count(long call_number){
25     if (call_number < NR_syscalls) {
26         hts_count[call_number]=0;
27         printk ("INFO Reset the count of<%d>, which is %d\n",call_number, hts_count[call_number]);
28     }
29     else {
30         printk ("NO SUCH CALL!\n");
31         return -1;
32     }
33     return 0;
34 }

```

/kernel/sys.c

完成对 sys.c 的修改后，我们完成核心功能，对记录数组在被调用时进行增 1。查阅资料<sup>[1,2]</sup>发现这些实验报告都采用了对/arch/i386/kernel/entry.S 进行修改来完成这个自增操作。但是，这些报告使用的是较低的内核版本，2.0+ 版本。而在 4.15 版本的内核中 entry 的位置已经和以前不一样。4.15 已经不存在这个目录，entry.S 在 4.15 中是放在 arch/x86/entry 之中，分为 entry32.S 和 entry64.S。点开发现这些文件是使用汇编代码编写，代码结构和前人的实验报告里的代码结构和逻辑都有很大区别，而且个人对汇编代码不太熟悉，这是有些无从入手。这里不太确定是否能直接对记录数组用汇编进行引用自增。后来经过很长时间的 api 资料查阅<sup>[3]</sup>以及和同学交流讨论，发现无论是 64 位还是 32 位的系统调用都会先调用 commons.c 里面的 do\_syscall\_64() 或者 do\_syscall\_32() 函数来进行初始化。而由于我们之前 export 了记录数组，在内核的 c 文件应该都是可以直接调用记录数组的。所以我们可以直接对 do\_syscall\_64() 进行一些小的修改以完成任务。

```

1 extern long hts_count [];
2 __visible void do_syscall_64( struct pt_regs *regs)
3 {
4     struct thread_info *ti = current_thread_info ();
5     unsigned long nr = regs->orig_ax;

```



```

7   enter_from_user_mode();
   local_irq_enable ();

9

11  if (READ_ONCE(ti->flags) & TIF_WORK_SYSCALL_ENTRY)
       nr = syscall_trace_enter (regs);

13  /*
   * NB: Native and x32 syscalls are dispatched from the same
15  * table. The only functional difference is the x32 bit in
   * regs->orig_ax, which changes the behavior of some syscalls.
17  */
   if (likely ((nr & __SYSCALL_MASK) < NR_syscalls)) {
19       hts_count[nr]++;
       nr = array_index_nospec(nr & __SYSCALL_MASK, NR_syscalls);
21       regs->ax = sys_call_table[nr](
           regs->di, regs->si, regs->dx,
23       regs->r10, regs->r8, regs->r9);
   }

25   syscall_return_slowpath (regs);
27 }

```

arch/x86/entry/commons.c

这里我们主要是先用 `extern` 关键字引用之前定义的记录数组。然后在引入系统调用表之前对被调用的系统调用在记录数组对应的位置进行加 1 操作。个人感觉这个逻辑和 2.x 版本的汇编代码的逻辑很像，有可能是以前的 `entry.S` 的一些功能现在放在 `commons.c` 里面了。

完成自增操作后，剩下的工作就比较简单了。首先回忆到我们在 `sys.c` 中曾经写了两个系统调用（获取计数和清零）的实现。现在我们要在 `/include/linux/syscalls.h` 中去声明这两个系统调用。见下：

```

1  asm linkage long sys_check_count(long call_number, char __user *buffer);
   asm linkage long sys_reset_count(long call_number);

```

arch/x86/entry/commons.c

另外，还需要在 `/arch/x86/entry/syscalls/syscall_64.tbl` 中注册这两个系统调用，如下

```

334   common reset_count          sys_reset_count
2 335   common check_count         sys_check_count

```

/arch/x86/entry/syscalls/syscall\_64.tbl

其中，334 和 335 为这两个调用的系统调用号。完成以上操作，我们可以开始编译内核。按顺序输入以下指令：

```

2  sudo apt-get install libncurses5-dev
   sudo make mrproper
4  sudo make menuconfig

```

```
sudo make -j12
sudo make modules_install
sudo make install
```

command in linux console

其中, `libncurses5-dev` 为编译内核必须要的包。另外系统不同状态, 有可能需要安装额外的包以保证编译顺利进行。`sudo make mrproper` 会删除之前编译生成的目标文件, 在首次编译不是必须要运行这条指令。如果使用虚拟机进行编译, 运行 `sudo make menuconfig` 时需要注意保持虚拟机分辨率足够大, 或者全屏显示虚拟机, 否则会出现一些不必要的错误。`sudo make menuconfig` 顺利运行后会弹出一个选择框如图 1-2. 在该界面可以直接选择 `exit` 保存默认配置为编译配置。`sudo make -j12` 后的 `-j12` 表示使用多少个线程并行编译, 使用者可以按照自己机器的实际线程数设置线程数。通常来说, 多线程能加快编译速度, 在笔者的机器这个编译过程需要达到 40 分钟。编译顺利完成后, 需要修改 `grub`

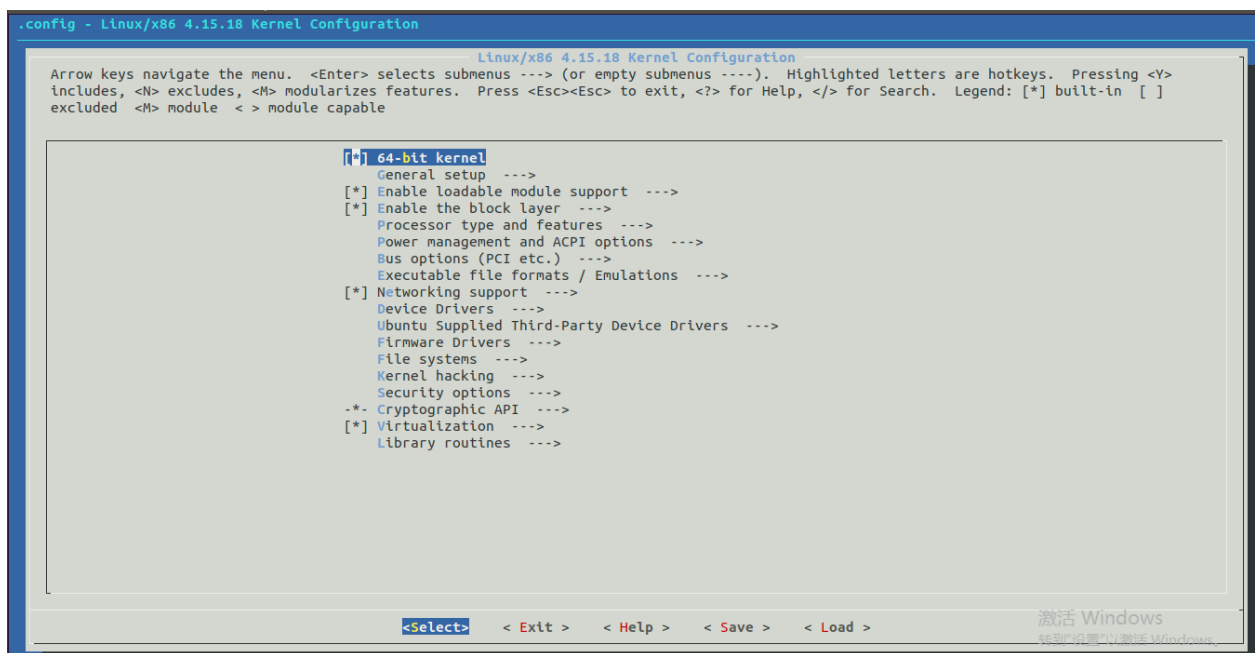


图 1-2 menuconfig 命令界面

配置以使得可以在开机 boot 时选择要使用的内核版本。输入以下命令 `sudo gedit /etc/default/grub`, 将 `grub` 配置中的 `GRUB_TIMEOUT_STYLE=hidden` 和 `GRUB_TIMEOUT=0` 注释掉。

```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
# info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
#GRUB_TIMEOUT_STYLE=hidden
```

```
#GRUB_TIMEOUT=0
```

```
9 ...
```

```
/etc/default/grub
```

命令行中输入 `sudo update-grub` 应用修改。然后输入 `reboot` 重启机器。然后在 `advance options` 中选择我们刚刚安装的内核版本。如果按照默认设置，系统会默认选择版本号最高的内核版本。如果没有意外，我们已经进入使用新编译安装的内核的系统中，接下来我们需要编写一些用户程序来验证我们系统调用的有效性。首先我们先写一个 `count.c` 来输出我们对某一个系统调用的调用次数查询结果。这里需要包含一些头文件来赋能我们的系统调用。`syscall` 定义在 `sys/syscall.h` 中，是所有系统调用的入口。其中第一个参数是我们需要调用的系统调用号，后面的参数则是具体系统调用的入参。根据我们之前的代码，335 系统调用对应的就是查询系统调用的调用号，我们将该参数传入 `syscall`，附带我们在用户空间创建的数组地址和要查询的调用号。如果查询成功则输出查询的结果。这里的调用号 333 对应的是我们在任务 3 中自己编写的一个系统调用。

```
1 #include <linux/kernel.h>
2 #include <sys/syscall.h> // 系统调用的头文件
3 #include <unistd.h>
4 #include <stdio.h>
5 using namespace std;
6 int main(){
7     long user_space[1];
8     long a= syscall(335,333,user_space);
9     if(a==0){
10         printf("The count number of call %d is %d\n",333,user_space[0]);
11     }
12     else{
13         printf("Check failure ! Something happens!\n");
14     }
15     return 0;
16 }
```

```
count.c
```

另外，我们创建了一个 `reset` 的用户程序，负责对系统调用 333 进行计数清零。代码和之前的用户程序类似，读者可自行分析验证。

```
1 #include <linux/kernel.h>
2 #include <sys/syscall.h> // 系统调用的头文件
3 #include <unistd.h>
4 #include <stdio.h>
5 using namespace std;
6 int main(){
7     long a= syscall(334,333);
8     if(a==0){
9         printf("Reset success !\n");
10     }
11     else{
```

```

12     printf("Reset failure ! Something happens!\n");
13     }
14     return 0;
15 }

```

reset.c

编写完两个程序之后，对两个 c 应用程序进行编译，运行以下命令：

```

1 g++ reset.c -o reset.o
g++ count.c -o count.o

```

linux command

这两条指令会在当前文件夹输出 count.o 和 reset.o 两个可运行程序。

### 1.3 实验结果

```

linux@ubuntu:~/count_syscall$ ./count.o
The count number of call 333 is 0
linux@ubuntu:~/count_syscall$ ./call333.out
real pid:2098
current process's pid:2098
current process's status:1
current process's parent: 1882
first child's pid: 941139205
current proccess's running time(system): 0
current proccess's running time(user): 0
linux@ubuntu:~/count_syscall$ ./call333.out
real pid:2098
current process's pid:2098
current process's status:1
current process's parent: 1882
first child's pid: 941139205
current proccess's running time(system): 0
current proccess's running time(user): 0
linux@ubuntu:~/count_syscall$ ./count.o
The count number of call 333 is 2
linux@ubuntu:~/count_syscall$ ./reset.o
Reset success!
linux@ubuntu:~/count_syscall$ ./count.o
The count number of call 333 is 0
linux@ubuntu:~/count_syscall$

```

图 1-3 任务一实验结果

实验结果见图 1-3，其中 call333.out 是我们任务 3 编写的用户程序，运行一次会调

用调用号 333 的系统调用一次。可以看出我们执行 `call333.out` 两次后，查询到的调用次数为 2，而执行 `reset` 后查询到的调用次数为 0. 符合程序运行的基本逻辑。

## 1.4 运行说明

为了复现该程序，首先需要通过提供的 `patch` 包更新内核源代码，并编译安装新的内核代码。(这个过程在第四章会详细说明)。在内核代码安装成功后，我们提供了两个用户程序的 `.c` 和 `.o` 代码，按正常来说 `.o` 代码应该能直接执行。如果执行失败，需要手工重新编译两个 `.c` 文件。另外我们在包内包含了任务三的 `.c` 和 `.out` 文件。在实际测试过程中可能要用到任务三的代码调用 333 系统调用。

## 第二章 实验任务二

### 2.1 题目要求

编写一个内核模块，在/sysfs 文件系统中增加一个目录 **hello**，并在这个目录中增加一个文件 **world**，文件的内容为 **hello world**。

### 2.2 实现过程

这个任务比较简单，且无需重新编译内核，只需要编译模块然后动态编入内核之中。

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/uaccess.h>

#define MODULE_VERS "1.0"
#define MODULE_NAME "procfs_hello_world"

static struct proc_dir_entry *parent_dir;
char* message;
int len;
int bar_temp;

static ssize_t read_hello( struct file *filp, char __user *buf, size_t count, loff_t *offp ) {
    printk (KERN_INFO "count=%d\n", count);
    if (count>bar_temp)
        count=bar_temp;
    bar_temp=bar_temp-count;
    copy_to_user(buf, message, count);
    if (count==0)
        bar_temp=len;
    return count;
}

static const struct file_operations proc_fops={
    .read=read_hello
};

static int __init init_hello_world (void)
{
    parent_dir=proc_mkdir("hello", NULL);
    if (parent_dir==NULL)
    {
        printk (KERN_INFO "ERROR CREATING PROC ENTRY");
    }
    proc_create ("world",0, parent_dir,&proc_fops);
    message="hello world\n";
    len= strlen (message);
    return 0;
}
```

```

44 }
46 static void __exit cleanup_hello_world(void){
    printk (KERN_INFO"%s%s removed\n", MODULE_NAME, MODULE_VERS);
48 remove_proc_entry("hello", parent_dir);
    remove_proc_entry("world",NULL);
50 }

52 module_init( init_hello_world );
    module_exit(cleanup_hello_world);
54 MODULE_LICENSE("GPL"); //模块许可声明
    MODULE_AUTHOR("Tiansheng Huang");
56 MODULE_DESCRIPTION("hello world");

```

hello\_world.c

其中，我们主要写了一些在模块加载时需要做的初始化操作以及在模块退出要做的清除操作，分别用两个函数 `init_hello_world()` 和 `cleanup_hello_world()` 实现。在 `init_hello_world()` 中我们先调用 `proc_mkdir()` 创建一个 `hello` 文件夹，返回该文件夹的指针，然后调用 `proc_create()` 使用该指针在 `hello` 文件夹中创建一个 `world` 文件。并传入 `struct file_operations proc_fops` 来定义读取该文件时内核所需要做的操作。由于任务只要求读操作，我们仅仅为该文件赋予了读操作时所需要的内核操作。在读取操作时，用户空间传入一个用户内存的指针和可用的计数值。由于有可能出现用户的可用空间较少不够输出我们的信息，所以需要在 `read_hello()` 中循环的输出信息。即如果第一次请求给出的空间不足，那就先给对应空间的信息。在用户第二次请求时在返回剩下的信息。以此类推。对于模块退出后的清除操作，我们需要删除我们创建的文件和文件夹。清除 `proc` 内的文件夹和文件可以用 `remove_proc_entry()`，这些 `proc` 文件操作都声明在 `linux/proc_fs.h` 头文件中。

完成以上 `c` 文件，我们还需要写一个 `makefile` 文件以赋能内核模块的编译。代码如下：

```

TARGET=hello_world
2 KDIR :=/lib/modules/$(shell uname -r)/build
  PWD :=$(shell pwd)
4 obj-m :=$(TARGET).o
  default :
6     make -C $(KDIR) M=$(PWD) modules

8 clean :
    rm -f *.ko *.o *.mod.o *.mod.c *.symvers *.order

```

makefile

## 2.3 实验结果

在对模块进行编译加载后，实验结果显示模块顺利加载，且生成了指定文件夹和文件内容。注意当模块初次加载，cat 对应的文件可能没有输出，这是再 cat 一次就能正确加载内容。

```
linux@ubuntu:~$ cat /proc/hello/world
hello world
```

图 2-1 实验二实验结果

## 2.4 运行说明

模块编译安装过程需要按顺序执行下图命令。

```
linux@ubuntu:~$ make
make -C /lib/modules/5.3.0-28-generic/build M=/home/linux modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-28-generic'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-28-generic'
```

图 2-2 make module

```
linux@ubuntu:~$ sudo insmod hello_world.ko
linux@ubuntu:~$ lsmod
Module                Size  Used by
hello_world            16384  0
```

图 2-3 install module

这里 lsmod 是显示所有已加载的模块，显然我们的 hello world 模块也正常加载了。

最后我们调用 rmmod 来进行最后的收尾工作，注销模块。

```
1 sudo rmmod hello_world.ko
```

增加系统调用



## 第三章 实验任务三

### 3.1 题目要求

写一个新的系统调用，输出给定进程的子孙进程的相关信息。信息包括进程号 (PID), 状态和运行时间统计以及其父进程、第一个子进程的 PID。该系统调用有 3 个参数：给定进程的 PID，一个存储这些数据的缓冲，以及这个缓冲的大小。

### 3.2 实验过程

题目要求时写一个系统调用，我们按顺序对以下文件如任务一版进行编码：

```
1 /arch/x86/entry / syscalls / syscall_64 . tbl // 设置系统调用号，如果是32位就用syscall_32.tbl
3 /include / linux / syscalls . h // 系统调用的头文件
5 /kernel / sys . c // 定义系统调用函数
```

增加系统调用

首先我们先完成系统调用的主要实现，同样将以下函数加入 kernel/sys.c 中

```
1 asmlinkage long sys_hts_call (long pid1, char __user *buffer, long len)
{
3 printk (KERN_INFO "Welcome to hts call!\n");
  struct pid *kpid;
5 struct task_struct *task;
  struct task_struct *temp;
7 long long my_buffer[6];
  kpid = find_get_pid (pid1);
9 task = pid_task (kpid, PIDTYPE_PID);
  if (NULL == task) {
11     printk (KERN_INFO "NO SUCH A TASK!\n");
    return -1;
13 }
  if (len < sizeof (my_buffer)) {
15     printk (KERN_INFO "NO ENOUGH SPACE FOR INFORMATION!\n");
    return -1;
17 }
  printk (KERN_INFO "pid : %d\n", task->pid); /*当前进程PID*/
19 printk (KERN_INFO "Task state: %d\n", task->state); /*运行状态，-1为不可运行，0为可运行，>0为运行
   结束*/
  printk (KERN_INFO "Task name: %s\n", task->comm); /*进程名*/
21 printk (KERN_INFO "Running time (system): %d\n", task->stime);
  printk (KERN_INFO "Running time (user): %d\n", task->utime);
23 int i=0;
  for (temp=task->parent; temp!=&init_task; temp=temp->parent)
25 { /*输出父进程直到init*/
    i=i+1;
    printk (KERN_INFO "%d parent pid: %d\n", i, temp->pid);
27 }
  printk (KERN_INFO "child pid: %d\n", list_first_entry (&(task->children), struct task_struct, sibling )->
    pid);
29 printk (KERN_INFO "END!\n");
```

```

31 my_buffer[0] = task->pid;
    my_buffer[1] = task->state;
33 my_buffer[2] = task->parent->pid;
    my_buffer[3] = list_first_entry (&(task->children), struct task_struct , sibling )->pid;
35 my_buffer[4] = task->stime;
    my_buffer[5] = task->utime;
37 copy_to_user( buffer , my_buffer, sizeof(my_buffer));
    return 0;
39 }
    
```

kernel/sys.c

我们知道，进程运行的相关信息都以 `task_struct` 形式存储。而为了获取到 `pid` 对应的进程的 `task_struct`，我们需要先调用 `find_get_pid` 获取 `kpid` 再调用 `pid_task` 获取到对应的 `task_struct`。获取 `task_struct`，我们可以直接获得进程的父进程的 `task_struct` 指针，以及进程的状态，运行时间等。唯一比较有难度的一个点在于获取其第一个子进程的 `pid`。虽然 `task_struct` 内储存了一个 `children` 成员。但它并不是一个简单的指向其 `children` 的链表。某个 `task_struct` 的 `children` 存储的是它的 `children` 的 `sibling` 成员。这个 `sibling` 成员实际上是一个链表的一个头节点（这个头节点指向 `pid=0` 的 `task_struct`）。对于某一个 `parent` 的所有 `children`，它的 `sibling` 成员都是同一个 `sibling` 链的头节点。通过这种操作，可以保证 `sibling` 和 `children` 的一致性。所以我们在调用 `list_first_entry` 获取第一个孩子的指针时，需要在第三个参数传入 `sibling`，因为我们需要通过这个成员获取 `pid=0` 的进程的第一个 `sibling`，换句话说也就是待查询进程的第一个 `children`。这里相关资料见[4]。

最棘手的部分完成后，我们同样采用了一个 `long` 数组来存储我们将要返回给客户端的信息，将任务需要的信息以 `long` 形式使用 `copy_to_user` 存储到用户空间。如果发生查询不到等错误，返回-1 并使用 `printk()` 打印一些内核错误日志。接下来我们像任务一一样修改索引以及头函数声明。

```

1 asmlinkage long sys_hts_call (long pid1, char __user *buffer, long len);
    
```

include/linux/syscalls.h

```

1 ...
332 common statx sys_statx
3 333 64 hts_call sys_hts_call
    
```

arch/x86/entry/syscalls/syscall\_64.tbl

通任务一，进行内核代码编译：

```

1 sudo make mrproper
    sudo make menuconfig
3 sudo make -j12
    sudo make modules_install
5 sudo make install
    
```

command in linux console

编译完成并重启后，我们着手编写应用程序进行测试。

```

1 #include <linux/kernel.h>
2 #include <sys/syscall.h> // 系统调用的头文件
3 #include <unistd.h>
4 #include <stdio.h>
5 using namespace std;
6 int main(){
7     long long user_space[6];
8     user_space[0]=3;
9     long pid=2098;
10    long a= syscall(333,pid,user_space,sizeof(user_space));
11    printf("real pid:%d\n",pid);
12    printf("current process's pid:%d\n",user_space[0]);
13    printf("current process's status:%d\n",user_space[1]);
14    printf("current process's parent: %d\n",user_space[2]);
15    printf("first child's pid: %d\n", user_space[3]);
16    printf("current proccess's running time(system): %d\n",user_space[4]);
17    printf("current proccess's running time(user): %d\n",user_space[5]);
18    return 0;
19 }

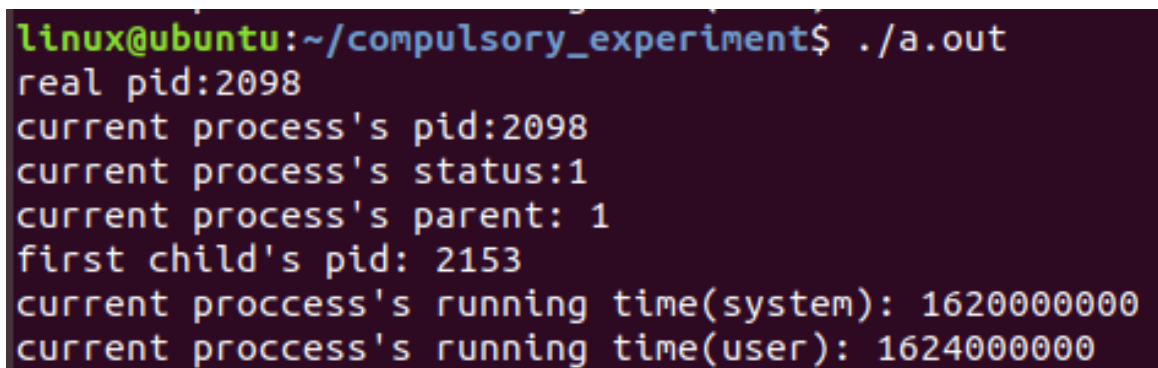
```

user\_interface

这里我们指定了需要查询的 pid 为 2098，并传入用户空间开辟的数组空间通过系统调用传入内核中，再打印相关信息。

### 3.3 实验结果

首先我们在编译 user\_interface.c 文件后，运行生成的.out 文件。输出任务需要打印的信息。注意这里后面的 running time 似乎有点异常，查阅资料发现，这个 running time 是以时钟数来计数的，所以数字比较大。在测试时 2098 进程是一个 firefox 浏览器进程，当我从后台状态转到显示状态，可以发现该进程的 running time 会有所增加。为了验证



```

linux@ubuntu:~/compulsory_experiment$ ./a.out
real pid:2098
current process's pid:2098
current process's status:1
current process's parent: 1
first child's pid: 2153
current proccess's running time(system): 1620000000
current proccess's running time(user): 1624000000

```

图 3-1 任务三实验结果

父子进程的 pid 是否正确。我们用 ps 指令详细查看 2098 进程及其相关的子进程，显示

如图 3-2 所示。可以看到第一个进程 2098 他的父进程的 pid 为 1，而 2153 进程，他的父进程是 2098，与我们在图 3-1 显示的数据一致。

```

current process is running time(user): 1624000000
linux@ubuntu:~/compulsory_experiment$ ps -ef|grep firefox
linux      2098      1  06:27 tty2      00:00:06 /usr/lib/firefox/firefox -new
-window
linux      2153     2098  0 06:27 tty2      00:00:01 /usr/lib/firefox/firefox -con
tentproc -childID 1 -isForBrowser -prefsLen 1 -prefMapSize 206206 -parentBuildI
D 20200117190643 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox/br
owser/omni.ja -appdir /usr/lib/firefox/browser 2098 true tab
linux      2211     2098  0 06:27 tty2      00:00:00 /usr/lib/firefox/firefox -con
tentproc -childID 2 -isForBrowser -prefsLen 6827 -prefMapSize 206206 -parentBui
ldID 20200117190643 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox
/browser/omni.ja -appdir /usr/lib/firefox/browser 2098 true tab
linux      2291     2098  0 06:27 tty2      00:00:00 /usr/lib/firefox/firefox -con
tentproc -childID 4 -isForBrowser -prefsLen 7618 -prefMapSize 206206 -parentBui
ldID 20200117190643 -greomni /usr/lib/firefox/omni.ja -appomni /usr/lib/firefox
/browser/omni.ja -appdir /usr/lib/firefox/browser 2098 true tab
linux      2529    1890  0 06:38 pts/1      00:00:00 grep --color=auto firefox
    
```

图 3-2 任务三实验结果验证

通过在控制台输入 `dmesg`, 我们可以看到内核输出的信息, 见图 3-3.

```

[ 2813.595196] Welcome to hts call!
[ 2813.595199] pid : 2098
[ 2813.595199] Task state: 1
[ 2813.595199] Task name: firefox
[ 2813.595200] Running time (system):1620000000
[ 2813.595200] Running time (user):1624000000
[ 2813.595201] 1 parent pid:1
[ 2813.595201] child pid:2153
[ 2813.595201] END!
    
```

图 3-3 任务三打印内核日志

### 3.4 运行说明

同样, 要运行我们的实验需要按步骤打 `patch` 然后编译内核代码。对于应用程序, 由于测试机未必含有 2098 进程, 建议测试者用 `ps` 指令查看本机的一些现有运行的进程, 获取其进程号。然后将该进程号填入 `user_interface` 的 `pid` 属性中。然后输入以下命令进行编译运行:

```

g++ user_interface .c -o user_interface .out
./ user_interface .out
    
```

linux command

## 第四章 内核编译安装说明

提供的实验压缩包里面包含了 linux-4.15.0.patch 文件以及各个实验的源码。测试者需要在 windows 下解压实验压缩包，取出 .patch 文件和源码，然后将 .patch 文件应用到自己下载的内核源码中。详细教程见下：

若需要整合我们提供的内核版本，需要首先下载内核 4.15 版本的源代码并进行解压，然后使用提供的 patch 进行更新，使用下指令：

```
1 sudo apt-get install linux-source-4.15.0
2 cd usr/src
3 sudo tar -xvjf linux-source-4.15.0.tar.bz2
4 cd linux-source-4.15.0
5 patch -p1 < linux-4.15.0.patch
```

linux command

打包完成后，进行内核编译。

```
1 sudo apt-get install libncurses5-dev
2 sudo make mrproper
3 sudo make menuconfig
4 sudo make -j12
5 sudo make modules_install
6 sudo make install
```

linux command

编译顺利完成后，需要修改 grub 配置以使得可以在开机 boot 时选择要使用的内核版本。输入以下命令 `sudo gedit /etc/default/grub`，将 grub 配置中的 `GRUB_TIMEOUT_STYLE=hidden` 和 `GRUB_TIMEOUT=0` 注释掉。

```
1 # If you change this file, run 'update-grub' afterwards to update
2 # /boot/grub/grub.cfg.
3 # For full documentation of the options in this file, see:
4 #   info -f grub -n 'Simple configuration'
5
6 GRUB_DEFAULT=0
7 #GRUB_TIMEOUT_STYLE=hidden
8 #GRUB_TIMEOUT=0
9 ...
```

/etc/default/grub

命令行中输入 `sudo update-grub` 应用修改。然后输入 `reboot` 重启机器。然后在 `advance options` 中选择我们刚刚安装的内核版本。如果按照默认设置，系统会默认选择版本号最高的内核版本。

## 第五章 总结

本次实验看起来难度比较简单，但是非常容易出错。而且内核编译时间比较长，一个小错误可能会导致 40 分钟的编译时间。内核开发实在是需要非常细心。而且在 linux 环境下写代码没有 ide 的错误提示，有时候一些小的语法错误一不注意也会导致浪费半个钟的编译时间，所以耐心在内核开发中也是非常重要。所以这次实验也花了不少时间，不断调试找错误。另外就是网上的一些资料很多建立在低版本的内核中，而内核代码基本上每代的更新都有非常大的区别，需要搞清楚原理再动手进行编写，而不能生搬硬套。通过本次实验，对 linux 环境以及内核的了解极大的提升，在以后工作学习需要的话可以很快上手功能开发。总体来说，这次实验还是学到了不少新的知识，希望将来能派上用场。

## 参考文献

- [1] 华南理工实验报告 1[M].[S.l.]: [s.n.] . <http://www.docin.com/p-1716928383.html>.
- [2] 华南理工实验报告 2[M]. [S.l.]: [s.n.] . <https://wenku.baidu.com/view/6be8be064b73f242336c5fc6.html>.
- [3] linux kernel documentation[M].[S.l.]: [s.n.] . <https://www.kernel.org/doc/html/latest/core-api/index.html>.
- [4] children and sibling in task\_struct[M].[S.l.]: [s.n.] . <https://stackoverflow.com/questions/8207160/kernel-how-to-iterate-the-children-of-the-current-process>.

## 致谢

The authors want to thank Prof. Yimin Wu for his most devoted teaching during the lesson.  
This original work is inspired by Prof. Wu's insightful talk in class.

黄天晟

2020 年 3 月 8 日