



## 《操作系统》课第五次实验报告

学院:	软件学院
姓名:	郁万祥
学号:	2013852
邮箱:	yuwanxiang0114@163.com
时间:	2022.10.14

### 0. 开篇感言

经过上次系统调用的实验学习，这次再次添加新的系统调用的过程比较熟练了，所以此次实验的难点就是，如何借助 `task_struct` 对系统的进程信息进行展示，这主要借助于 `<linux/sched.h>` 文件当中对于 `task_struct` 结构中含有的信息的定义，我们进行直接的调用输出即可。

### 1. 实验题目

Add a new system call to list all processes

### 2. 实验目标

- 1、添加新的系统调用
- 2、系统调用实现罗列出进程的有关信息

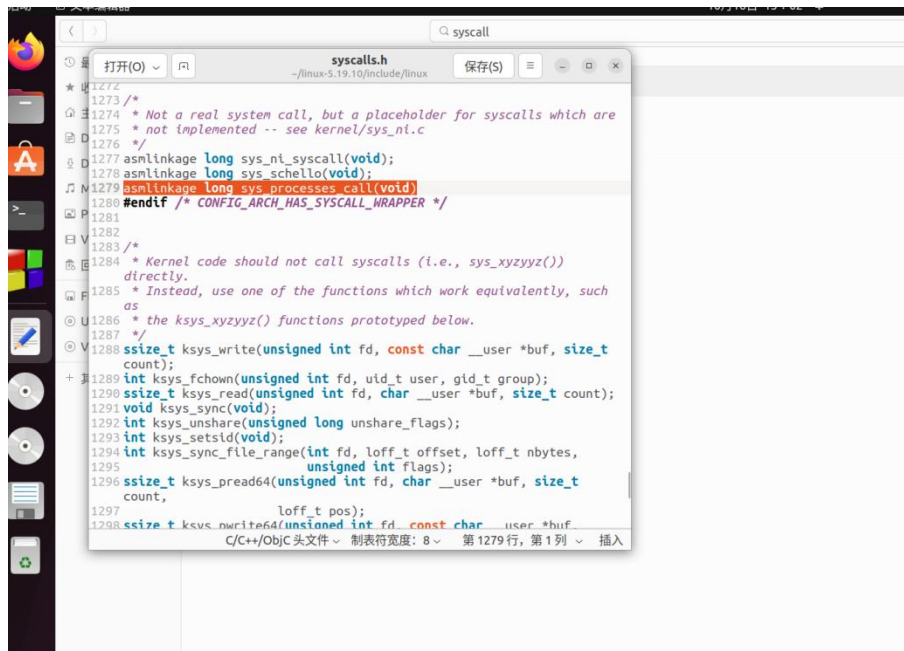


### 3. 原理方法

每个进程在内核中都有一个进程控制块(PCB)来维护进程相关的信息,Linux 内核的进程控制块是 task\_struct 结构体。task\_struct 是 Linux 内核的一种数据结构,它会被装载到 RAM 中并且包含着进程的信息。

### 4. 具体步骤

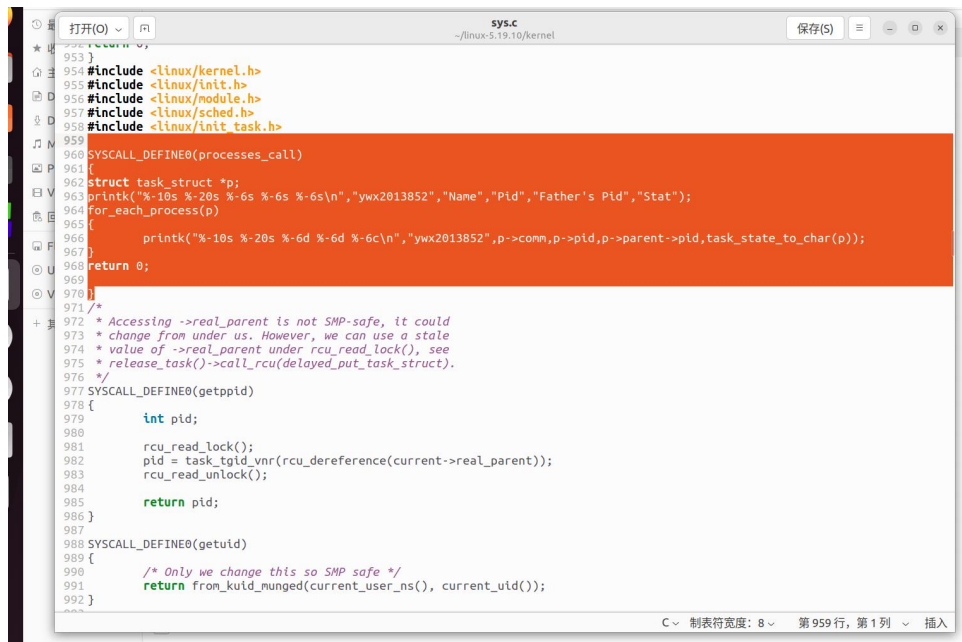
1、在 syscall.h 头文件当中添加 asmlinkage long processes\_call(void);



2、在 kernal/sys.c 中添加自己的服务函数



## 《操作系统》课程实验报告



```
953 }
954 #include <linux/kernel.h>
955 #include <linux/init.h>
956 #include <linux/module.h>
957 #include <linux/sched.h>
958 #include <linux/init_task.h>
959
960 SYSCALL_DEFINE0(processes_call)
961 {
962     struct task_struct *p;
963     printk("%-10s %-20s %-6s %-6s %-6s\n", "ywx2013852", "Name", "Pid", "Father's Pid", "Stat");
964     for_each_process(p)
965     {
966         printk("%-10s %-20s %-6d %-6d %-6c\n", "ywx2013852", p->comm, p->pid, p->parent->pid, task_state_to_char(p));
967     }
968     return 0;
969 }
970
971 /*
972  * Accessing ->real_parent is not SMP-safe, it could
973  * change from under us. However, we can use a stale
974  * value of ->real_parent under rcu_read_lock(), see
975  * release_task()->call_rcu(delayed_put_task_struct).
976  */
977 SYSCALL_DEFINE0(getppid)
978 {
979     int pid;
980     rcu_read_lock();
981     pid = task_tgid_vnr(rcu_dereference(current->real_parent));
982     rcu_read_unlock();
983     return pid;
984 }
985
986 SYSCALL_DEFINE0(getuid)
987 {
988     /* Only we change this so SMP safe */
989     return from_kuid_munged(current_user_ns(), current_uid());
990 }
991
992 }
```

3、在 arch/x86/entry/syscalls/syscall\_64.tbl 中添加系统调用号，添加系统调用号和系统调用函数的对应关系。



syscall number	architecture	name	handler
369 445	common	landlock_add_rule	sys_landlock_add_rule
370 446	common	landlock_restrict_self	sys_landlock_restrict_self
371 447	common	memfd_secret	sys_memfd_secret
372 448	common	process_mrelease	sys_process_mrelease
373 449	common	futex_waitv	sys_futex_waitv
374 450	common	set_mempolicy_home_node	sys_set_mempolicy_home_node
375 460	common	schello	sys_schello
376 461	common	processes_call	sys_processes_call

377 #

378 # Due to a historical design error, certain syscalls are numbered differently

379 # in x32 as compared to native x86\_64. These syscalls have numbers 512-547.

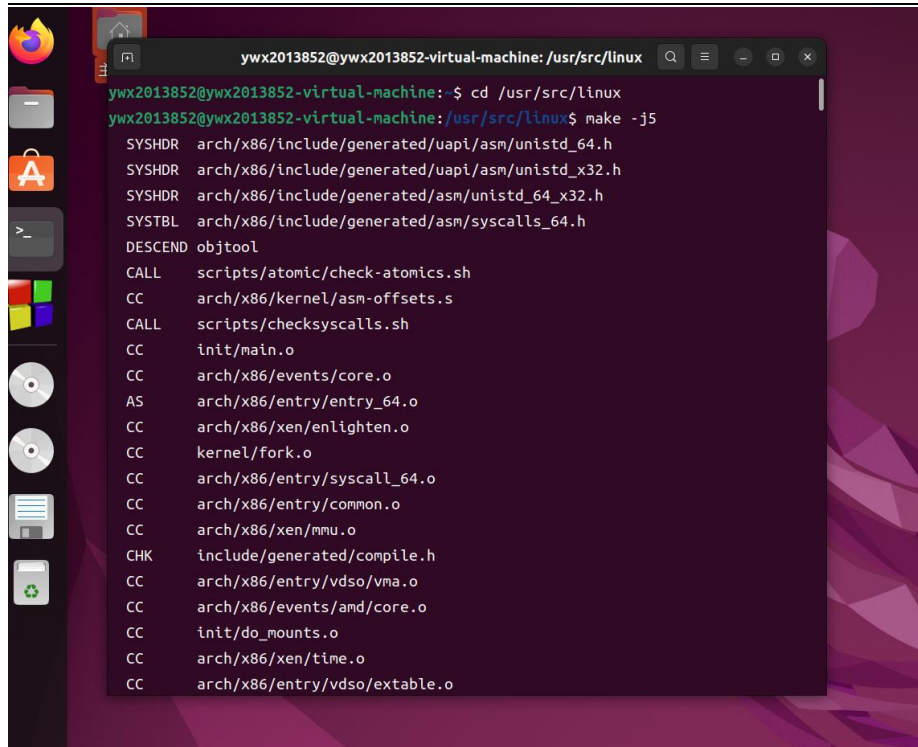
380 # Do not add new syscalls to this range. Numbers 548 and above are available

381 # for non-x32 use.

382 #

syscall number	architecture	name	handler
383 512	x32	rt_sigaction	compat_sys_rt_sigaction
384 513	x32	rt_sigreturn	compat_sys_x32_rt_sigreturn
385 514	x32	ioctl	compat_sys_ioctl
386 515	x32	readv	sys_readv
387 516	x32	writev	sys_writev
388 517	x32	recvfrom	compat_sys_recvfrom
389 518	x32	sendmsg	compat_sys_sendmsg
390 519	x32	recvmsg	compat_sys_recvmsg
391 520	x32	execve	compat_sys_execve
392 521	x32	ptrace	compat_sys_ptrace
393 522	x32	rt_sigpending	compat_sys_rt_sigpending
394 523	x32	rt_sigtimedwait	compat_sys_rt_sigtimedwait
395 524	x32	compat_sys_rt_sigtimedwait_time64	compat_sys_rt_sigtimedwait_time64

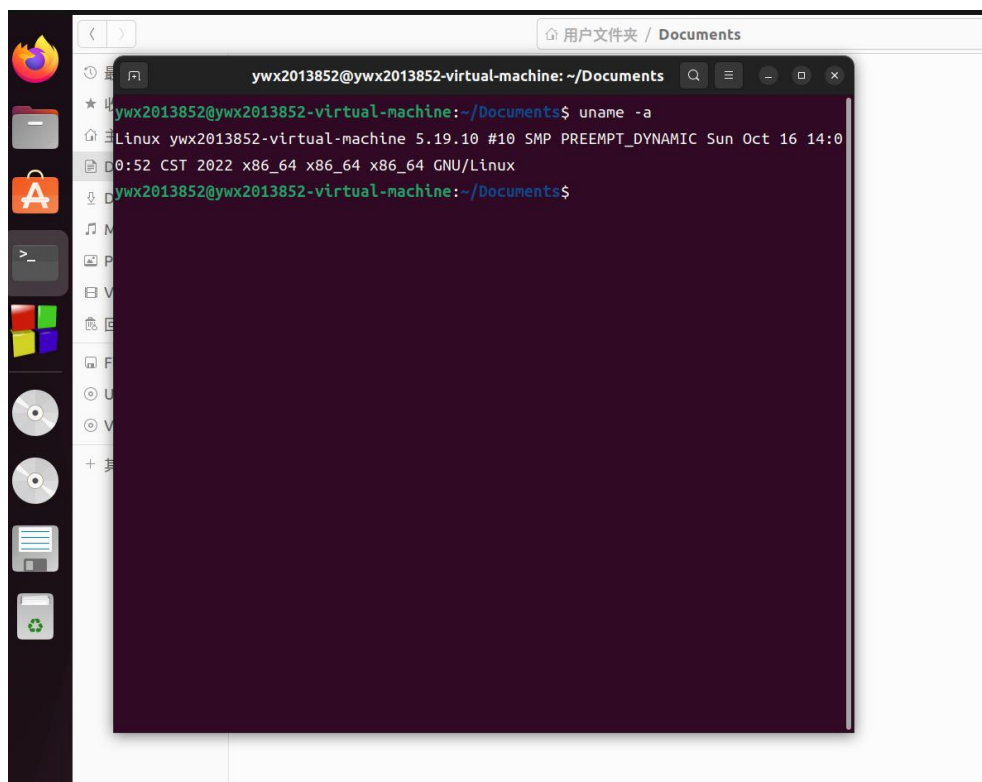
4、重新编译内核



```
ywx2013852@ywx2013852-virtual-machine: /usr/src/linux
ywx2013852@ywx2013852-virtual-machine:~$ cd /usr/src/linux
ywx2013852@ywx2013852-virtual-machine:/usr/src/linux$ make -j5
SYSHDR arch/x86/include/generated/uapi/asm/unistd_64.h
SYSHDR arch/x86/include/generated/uapi/asm/unistd_x32.h
SYSHDR arch/x86/include/generated/asm/unistd_64_x32.h
SYSTBL arch/x86/include/generated/asm/syscalls_64.h
DESCEND objtool
CALL scripts/atomic/check-atomics.sh
CC arch/x86/kernel/asm-offsets.s
CALL scripts/checksyscalls.sh
CC init/main.o
CC arch/x86/events/core.o
AS arch/x86/entry/entry_64.o
CC arch/x86/xen/enlighten.o
CC kernel/fork.o
CC arch/x86/entry/syscall_64.o
CC arch/x86/entry/common.o
CC arch/x86/xen/mmu.o
CHK include/generated/compile.h
CC arch/x86/entry/vdso/vma.o
CC arch/x86/events/amd/core.o
CC init/do_mounts.o
CC arch/x86/xen/time.o
CC arch/x86/entry/vdso/extable.o
```

5、重新启动后，检查内核版本，编写 demo 进行系统调用，进行测试：

内核版本检查：

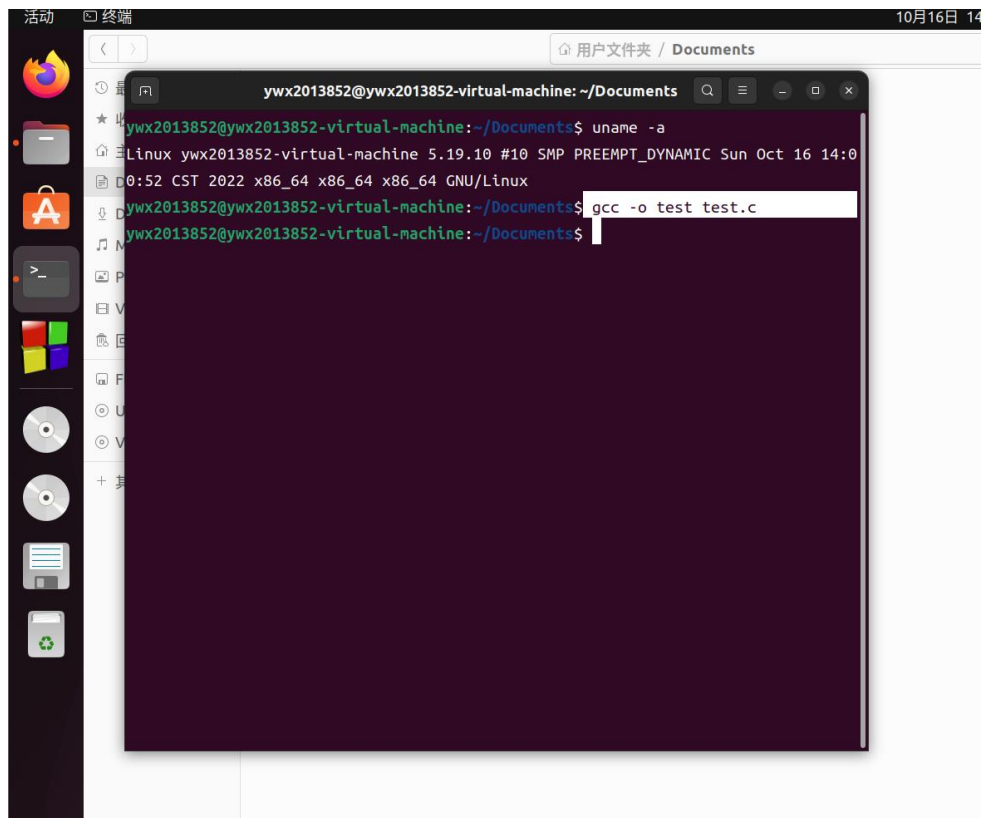


```
ywx2013852@ywx2013852-virtual-machine: ~/Documents
ywx2013852@ywx2013852-virtual-machine:~/Documents$ uname -a
Linux ywx2013852-virtual-machine 5.19.10 #10 SMP PREEMPT_DYNAMIC Sun Oct 16 14:0
0:52 CST 2022 x86_64 x86_64 x86_64 GNU/Linux
ywx2013852@ywx2013852-virtual-machine:~/Documents$
```

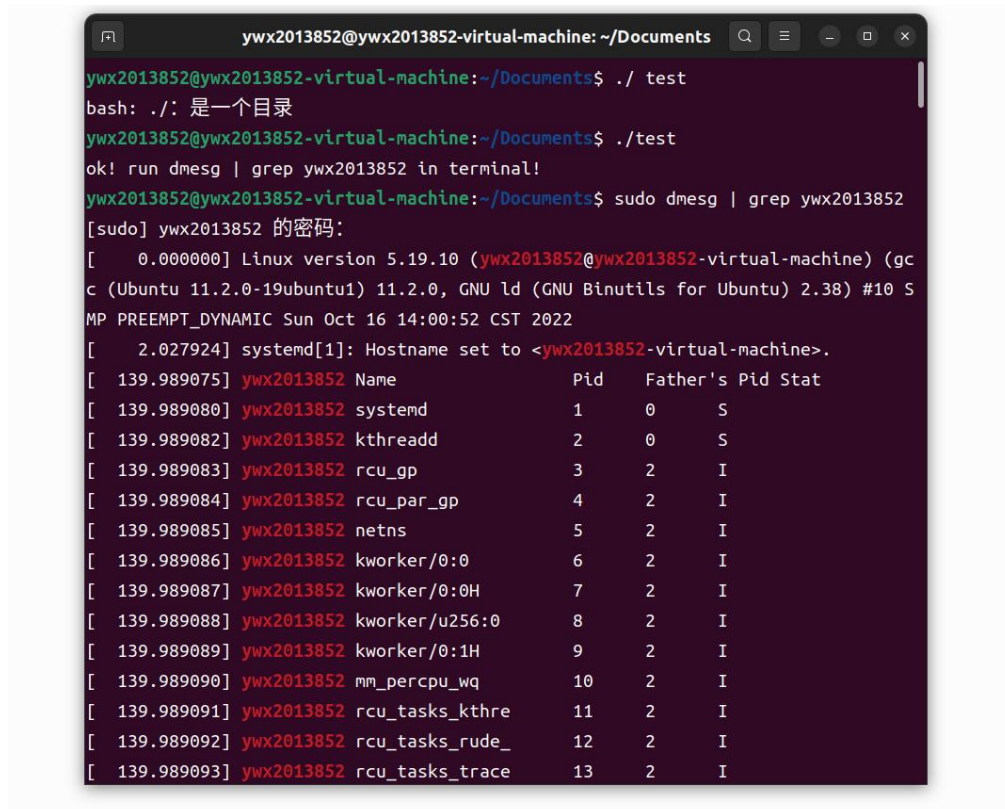
系统调用 c 文件编译：



## 《操作系统》课程实验报告



调用结果:





## 5. 总结心得

task\_struct 数据结果保存了 linux 当前的进程信息，并且，所有运行在系统中的进程都以 task\_struct 链表的形式存在在内核当中，对于 task\_struct，本次使用到了：

- 1、进程的状态 state
- 2、进程的唯一表示 pid
- 3、父进程 parent
- 4、进程的名字 comm

因为进程结构体在内核当中，都是以链表形式存在，因此我们在以 C 语言编写的 linux 内核之下，借助于指针，通过遍历列表，就可以得到系统运行的所有进程，通过访问指针的成员，就可以得到上述的所有进程的信息。

## 6. 参考资料

### 源代码

#### 服务函数：

```
SYSCALL_DEFINE0(processes_call)
{
    struct task_struct *p;
    printk("%-10s %-20s %-6s %-6s %-6s\n", "ywx2013852", "Name", "Pid", "Father's Pid", "Stat");
    for_each_process(p)
    {
        printk("%-10s %-20s %-6d %-6d %-6c\n", "ywx2013852", p->comm, p->pid, p->parent->pid, task_state_to_char(p));
    }
    return 0;
}
```



调用文件：

```
#include<unistd.h>
#include<sys/syscall.h>
#include<sys/types.h>
#include<stdio.h>
#define _NR_processes_call 461
int main(int argc,char *argv[])
{
    syscall(_NR_processes_call);
    printf("ok! run dmesg | grep ywx2013852 in terminal!\n");
    return 0;
}
```