

操作系统第11次作业

1. 实验步骤

1.1 在sys.c里面编写线程查询函数和文件拷贝函数

```
SYSCALL_DEFINE3(copy, char*, buf, char*, source, char*, destination)

{
    char buffer[128];
    struct file*src_file =NULL;
    struct file*dest_file=NULL;
    src_file = filp_open(source, O_RDWR | O_APPEND | O_CREAT, 0644);
    dest_file = filp_open(destination, O_RDWR | O_APPEND | O_CREAT, 0644);

    if (IS_ERR(src_file)) {
        printk("fail to open file");
        return 0;
    }
    if (IS_ERR(dest_file)) {
        printk("fail to open file");
        return 0;
    }
    int len;
    loff_t src=src_file->f_pos;
    loff_t dest=dest_file->f_pos;
    while((len=kernel_read(src_file,buffer,128,&src))>0)
    {
        kernel_write(dest_file,buffer,len,&dest);
    }
    filp_close(src_file,NULL);
    filp_close(dest_file,NULL);

    return 0;
}
```

使用 `filp_open` 函数尝试打开源文件和目标文件。如果文件不存在，则创建它们。`O_RDWR` 表示读写模式，`O_APPEND` 表示追加数据，`O_CREAT` 表示如果文件不存在则创建，`0644` 是文件权限。接着通过 `while` 循环，使用 `kernel_read` 从源文件中读取数据到 `buffer`，然后使用 `kernel_write` 将 `buffer` 中的数据写入目标文件。从而实现文件的拷贝。

```

SYSCALL_DEFINE2(alcall,int,cmd,char*,buf)
{
    int sum=0;
    struct task_struct *p;
    printk("Hello new system call alcall! 2112966陈高楠\n");
    printk("Hello new system call alcall (%d,%x)!\n",cmd,buf);
    printk("%-20s %-6s %-6s\n","Name","Pid","Stat");
    for (p = &init_task; (p = next_task(p)) != &init_task;)
    {sum+=1;
    printk("%-20s %-6d %-6ld\n",p->comm,p->pid,p->__state);
    }
    printk("the number of the process is %d",sum);
    return 0;
}
-----

```

这个函数通过for循环遍历进程并输出，统计进程数并输出。

1.2 在syscalls.h里面定义函数接口

写入asm linkage long __x64_sys_alcall(int cmd,char*buf),

asm linkage long __x64_sys_copy(char*buf,char*source,char*destination)。

```
打开(O) 保存(S) syscalls.h ~/linux-6.5.7/include/linux
1169
1170
1171 /*
1172  * Not a real system call, but a placeholder for syscalls which are
1173  * not implemented -- see kernel/sys_ni.c
1174  */
1175 asmlinkage long sys_ni_syscall(void);
1176 asmlinkage long sys_schello(void);
1177 asmlinkage long __x64_sys_alcall(int cmd, char*buf);
1178 asmlinkage long __x64_sys_copy(char*buf, char*source, char*destination);
1179 #endif /* CONFIG_ARCH_HAS_SYSCALL_WRAPPER */
1180
1181 asmlinkage long sys_ni_posix_timers(void);
1182
1183 /*
1184  * Kernel code should not call syscalls (i.e., sys_xyzzyz()) directly.
1185  * Instead, use one of the functions which work equivalently, such as
1186  * the ksys_xyzzyz() functions prototyped below.
1187  */
1188 ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count);
1189 int ksys_fchown(unsigned int fd, uid_t user, gid_t group);
1190 ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count);
1191 void ksys_sync(void);
1192 int ksys_unshare(unsigned long unshare_flags);
1193 int ksys_setsid(void);
1194 int ksys_sync_file_range(int fd, loff_t offset, loff_t nbytes,
1195                          unsigned int flags);
1196 ssize_t ksys_pread64(unsigned int fd, char __user *buf, size_t count,
1197                     loff_t pos);
1198 ssize_t ksys_pwrite64(unsigned int fd, const char __user *buf,
1199                      size_t count, loff_t pos);
1200 int ksys_fallocate(int fd, int mode, loff_t offset, loff_t len);
1201 #ifdef CONFIG_ADVICE_SYSCALLS
1202 int ksys_fadvise64_64(int fd, loff_t offset, loff_t len, int advice);
1203 #else
1204 static inline int ksys_fadvise64_64(int fd, loff_t offset, loff_t len,
1205                                     int advice)
1206 {
1207     return -EINVAL;
```

1.3 在syscall_64.tbl里面写入

| syscall_64.tbl | | | |
|---------------------------------------|---|-------------------------|-----------------------------------|
| ~/linux-6.5.7/arch/x86/entry/syscalls | | | |
| 369 444 | common | landlock_create_ruleset | sys_landlock_create_ruleset |
| 370 445 | common | landlock_add_rule | sys_landlock_add_rule |
| 371 446 | common | landlock_restrict_self | sys_landlock_restrict_self |
| 372 447 | common | memfd_secret | sys_memfd_secret |
| 373 448 | common | process_mrelease | sys_process_mrelease |
| 374 449 | common | futex_waitv | sys_futex_waitv |
| 375 450 | common | set_mempolicy_home_node | sys_set_mempolicy_home_node |
| 376 451 | common | cachestat | sys_cachestat |
| 377 452 | common | alcall | sys_alcall |
| 378 453 | common | copy | sys_copy |
| 379 # | | | |
| 380 # | Due to a historical design error, certain syscalls are numbered differently | | |
| 381 # | in x32 as compared to native x86_64. These syscalls have numbers 512-547. | | |
| 382 # | Do not add new syscalls to this range. Numbers 548 and above are available | | |
| 383 # | for non-x32 use. | | |
| 384 # | | | |
| 385 512 | x32 | rt_sigaction | compat_sys_rt_sigaction |
| 386 513 | x32 | rt_sigreturn | compat_sys_x32_rt_sigreturn |
| 387 514 | x32 | ioctl | compat_sys_ioctl |
| 388 515 | x32 | readv | sys_readv |
| 389 516 | x32 | writew | sys_writew |
| 390 517 | x32 | recvfrom | compat_sys_recvfrom |
| 391 518 | x32 | sendmsg | compat_sys_sendmsg |
| 392 519 | x32 | recvmsg | compat_sys_recvmsg |
| 393 520 | x32 | execve | compat_sys_execve |
| 394 521 | x32 | ptrace | compat_sys_ptrace |
| 395 522 | x32 | rt_sigpending | compat_sys_rt_sigpending |
| 396 523 | x32 | rt_sigtimedwait | compat_sys_rt_sigtimedwait_time64 |
| 397 524 | x32 | rt_sigqueueinfo | compat_sys_rt_sigqueueinfo |
| 398 525 | x32 | sigaltstack | compat_sys_sigaltstack |
| 399 526 | x32 | timer_create | compat_sys_timer_create |
| 400 527 | x32 | mq_notify | compat_sys_mq_notify |
| 401 528 | x32 | kexec_load | compat_sys_kexec_load |
| 402 529 | x32 | waitid | compat_sys_waitid |
| 403 530 | x32 | set_robust_list | compat_sys_set_robust_list |
| 404 531 | x32 | get_robust_list | compat_sys_get_robust_list |
| 405 532 | x32 | vmsplice | sys_vmsplice |
| 406 533 | x32 | move_pages | sys_move_pages |
| 407 534 | x32 | preadv | compat_sys_preadv64 |

纯文本 制表符宽度: 8 第 378 行, 第 49 列 插入

1.4 编写测试文件，编译后进行测试

首先使用make命令进行重新编译

cd /usr/src/linux进入目录，之后make clean，然后make -j5。

编译完成后输入sudo make modules_install,sudo make install，之后输入reboot重启。

接着编写测试文件。

这是测试拷贝函数的c文件

```

1 #include <unistd.h>
2 #include <sys/syscall.h>
3 #include <sys/types.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #define __NR_copy 453
7 long mycopy(char*buf,char*source,char*destination){
8 return syscall(__NR_copy,buf,source,destination);

```

```

9 }
10 int main(int argc, char *argv[])
11 {
12
13 char *source=malloc(sizeof(char)*30);
14 char*destination=malloc(sizeof(char)*30);
15 source=argv[1];
16 destination=argv[2];
17 char*buf=malloc(sizeof(char)*16000);
18 printf("copy %s to %s",source,destination);
19 mycopy(buf,source,destination);
20 printf(" ok! \n");
21 free(buf);
22 return 0;
23 }
24

```

这是线程查询的c文件

```

1 #include <unistd.h>
2 #include <sys/syscall.h>
3 #include <sys/types.h>
4 #include <stdio.h>
5 #define __NR_alcall 452
6 long alcall(int cmd,char*buf){
7 return syscall(__NR_alcall,cmd,buf);
8
9 }
10 int main(int argc, char *argv[])
11 { int cmd;
12 char buf[256];
13 cmd=9;
14 alcall(cmd,buf);
15 printf("ok! run dmesg|grep alcall in terminal\n");
16 return 0;
17 }
18

```

2. 实验结果

在命令行执行测试文件效果如下：

查看线程以及线程数：

```
[ 215.584181] Hello new system call alcall! 2112966陈高楠
[ 215.584184] Hello new system call alcall (9,b1561170)!
[ 215.584186] Name          Pid      Stat
[ 215.584188] systemd          1         1
[ 215.584189] kthreadd          2         1
[ 215.584190] rcu_gp             3        1026
[ 215.584192] rcu_par_gp         4        1026
[ 215.584193] slub_flushwq       5        1026
[ 215.584194] netns              6        1026
[ 215.584195] kworker/0:0        7        1026
[ 215.584196] kworker/0:0H       8        1026
[ 215.584196] kworker/0:1        9        1026
[ 215.584197] kworker/u24:0      10       1026
[ 215.584198] mm_percpu_wq       11       1026
[ 215.584199] rcu_tasks_kthre    12       1026
[ 215.584200] rcu_tasks_rude_    13       1026
[ 215.584201] rcu_tasks_trace    14       1026
[ 215.584202] ksoftirqd/0        15         1
[ 215.584202] rcu_preempt        16       1026
[ 215.584203] migration/0        17         1
```

```
[ 215.584474] fwupd              1807         1
[ 215.584474] qq                  2513         1
[ 215.584475] qq                  2631         1
[ 215.584476] qq                  2634         1
[ 215.584477] qq                  2660         1
[ 215.584478] chrome_crashpad     2736         1
[ 215.584479] Xwayland            2739         1
[ 215.584479] gsd-xsettings       2743         1
[ 215.584480] ibus-x11            2775         1
[ 215.584481] qq                  2804         1
[ 215.584482] qq                  2808        8193
[ 215.584483] qq                  3131        8193
[ 215.584484] qq                  3162        8193
[ 215.584485] kworker/11:3        3199        1026
[ 215.584486] qq                  3320        8193
[ 215.584487] update-notifier     3399         1
[ 215.584488] deja-dup-monito     3582         1
[ 215.584488] gnome-terminal-     3671         1
[ 215.584489] bash                3697         1
[ 215.584490] alcall              3714         0
[ 215.584490] the number of the process is 293
```

拷贝文件并检查是否成功拷贝

```
cg2112966@cg2112966-VirtualBox:~$ gcc -o mycopy testcopy.c
cg2112966@cg2112966-VirtualBox:~$ ./mycopy /home/cg2112966/test /home/cg2112
966/test2
copy /home/cg2112966/test to /home/cg2112966/test2 ok!
cg2112966@cg2112966-VirtualBox:~$ diff -r test test2
cg2112966@cg2112966-VirtualBox:~$
```

由结果可得实验结果成功，可以显示线程以及线程数目，也可以拷贝文件。

3. 实验心得

在本次操作系统实验中，我探索了Linux内核，特别是系统调用的实现和扩展。在这一过程中，我增强了对操作系统核心机制的理解。我通过实现文件拷贝函数，我学习了如何在内核空间打开、读取、写入和关闭文件。这加深了我对操作系统核心概念的理解。