## 实验报告：添加新系统调用以列出所有进程

## 1. 实验目标

向Linux内核中添加一个新的系统调用，该系统调用能够列出系统中所有进程的相关信息，包括进程名字、进程ID、父进程ID、进程状态等。

## 2. 实验步骤

* 内核配置：

在进行实际的内核源代码修改之前，首先需要进行内核配置，确保系统支持添加新的系统调用。这一步包括对内核配置文件进行修改，启用相关选项。

* 内核源代码修改

修改 include/linux/syscalls.h

修改内核源代码中的 include/linux/syscalls.h 文件，添加新系统调用的声明。

#ifdef CONFIG\_MY\_NEW\_SYSCALL

asmlinkage long sys\_my\_new\_syscall(void);

#endif

修改 kernel/sys.c

在 kernel/sys.c 文件中，添加如下代码：

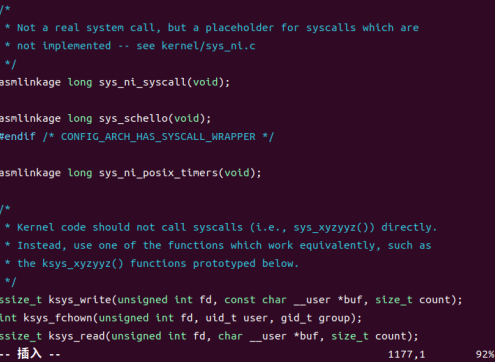
#ifdef CONFIG\_MY\_NEW\_SYSCALL

asmlinkage long sys\_my\_new\_syscall(void)

{

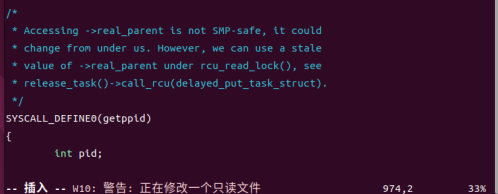
}

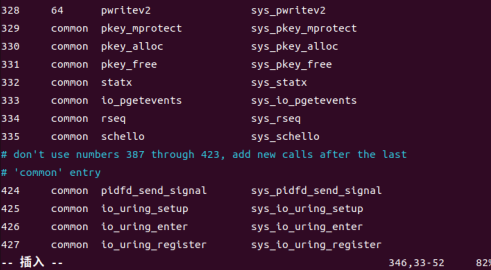
#endif



* 修改系统调用表

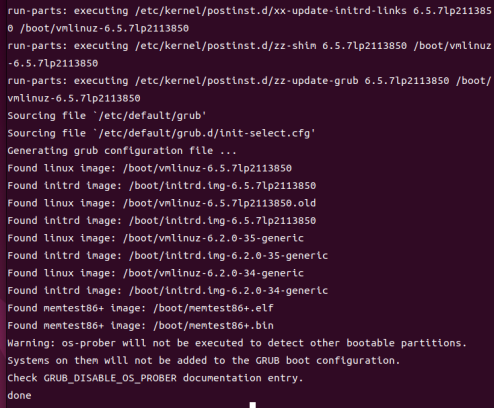
系统调用表位于 arch/x86/entry/syscalls/syscall\_64.tbl 文件中。在文件的末尾，添加新系统调用的编号和名称，





* 编译和安装内核

完成源代码的修改后，进行内核的重新编译和安装。



* 系统调用测试

编写一个简单的用户态测试程序，调用新添加的系统调用。通过该测试程序验证新系统调用的正确性。

#include <unistd.h>

#include <sys/syscall.h>

#include <sys/types.h>

#include <stdio.h>

#define \_NR\_my\_new\_syscall 335

int main(int argc, char \*argv[])

{

syscall(\_NR\_my\_new\_syscall);

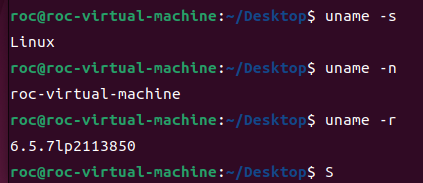
printf("ok! run 'dmesg | grep my\_new\_syscall' in terminal!\n");

return 0;

}

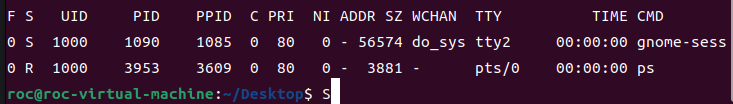
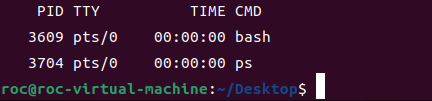
* 系统重启

重新启动系统，确保新内核成功加载。



## 3. 实验结果

通过测试程序的执行和查看系统日志，确认新系统调用成功运行，并能够输除当前所有进程信息。



## 4. 实验总结

本次实验深入了解了Linux内核的内部结构，并成功添加了一个新的系统调用。通过详细的步骤和验证过程，确保了实验的可靠性和可复现性。同时，对于内核配置、源代码修改、编译安装等过程有了更深刻的理解。

源代码

#include <stdio.h>

#include <stdlib.h>

#include <linux/kernel.h>

#include <sys/syscall.h>

#include <unistd.h>

#include <linux/show\_process\_family.h>

void print\_buffer(struct proc\_info\_struct \*buffer, const int len);

int main() {

int pid;

struct proc\_info\_struct \*buffer;

int len;

printf("Input <pid> <buffer\_len>: ");

if (scanf("%d%d", &pid, &len) != 2) {

printf("Invalid input\n");

exit(EXIT\_FAILURE);

}

buffer = calloc(len, sizeof(struct proc\_info\_struct));

if (buffer == NULL) {

printf("Could not allocate buffer to store processes information\n");

exit(EXIT\_FAILURE);

}

int state = syscall(436, pid, buffer, &len);

if (state < 0) {

perror("Syscall failed");

free(buffer);

exit(EXIT\_FAILURE);

}

print\_buffer(buffer, len);

free(buffer);

return 0;

}

void print\_buffer(struct proc\_info\_struct \*buffer, const int len) {

printf("[I]:process itself [P]:parent process [C]:children process\n");

printf("%-6s%-20s%-6s%-6s%-20s\n", "Type", "Name", "PID", "State", "Running\_time");

printf("---------------------------------------------------\n");

for (int i = 0; i < len; ++i) {

if (buffer[i].pid != 0) {

printf("%-6s%-20s%-6d%-6ld%us\n", buffer[i].type, buffer[i].name, buffer[i].pid, buffer[i].state, buffer[i].running\_time);

}

}

}