

重 庆 大 学

学 生 实 验 报 告

实验课程名称 操作系统

开课实验室 DS1502

学 院 大数据与软件学院 年级 22 专业班 软工 2 班

学 生 姓 名 潘铷葳 学 号 20221982

开 课 时 间 2023 至 2024 学年第 二 学期

总 成 绩	
教师签名	

软件学院制

《操作系统》实验报告

开课实验室：DS1502

2024 年 3 月 9 日

学院	大数据与软件学院	年级、专业、班	2022 级软件工程 2 班	姓名	潘铷葳	成绩	
课程名称	操作系统原理	实验项目名称	实验一 系统调用	指导教师	刘寄		
教师评语	教师签名： 年 月 日						

一、实验目的

通过编写系统调用，你可以学习系统调用的实现过程和原理，包括系统调用的注册、调用和处理过程，以及系统调用与用户态库函数之间的关系。

二、实验内容

1、K1——添加实现函数 `time_t sys_time()`

- (1) 获取定时器中断总次数：首先，代码通过 `g_timer_ticks` 变量获取了系统自启动以来的定时器中断总次数，假设这个变量是正确的记录了定时器中断次数。
- (2) 计算自启动以来的秒数：将获取到的定时器中断总次数除以定时器中断频率 `HZ`，得到了自系统启动以来的秒数。这里的计算假设每个定时器中断都是恰好一秒发生一次。
- (3) 计算当前系统时间：将自启动以来的秒数加上系统启动时刻 `g_startup_time`，得到了当前的系统时间。这里假设 `g_startup_time` 记录了系统启动时刻距离格林尼治时间 1970 年 1 月 1 日午夜的秒数。
- (4) 返回当前系统时间：将计算得到的当前系统时间作为函数的返回值，返回给调用方。

```
//K1
time_t sys_time()
{
    long
    t=g_startup_time+g_timer_trick/HZ;
    return t;
}
```

2、K2——声明函数 `time_t sys_time()`

打开 `kernel/kernel.h` 文件。
在文件合适的位置，加入以下声明：

```
time_t sys_time();
```

```
void mi_startup()
#endif /* _KERNEL_H*/
```

3、K3——定义系统调用号码

在 include/syscall-nr.h 中定义系统调用号码： 在该文件中添加以下内容：

```
#define SYSCALL_putchar    1000
#define SYSCALL_getchar    1001
#define SYSCALL_time       2024
```

4、K4——增加分支

在 syscall 函数的 switch 语句中增加一个分支，用于处理系统调用。在这个分支中，需要读取参数值、调用系统调用的实现函数并返回结果。

```
case SYSCALL_time:{
    time_t *loc=*(time_t**)(ctx->esp+4);
    ctx->eax=sys_time();
    if(loc!=NULL)
        *loc = ctx-> eax;
    break;}
```

5、U1——加入汇编语言接口

在 userapp/lib/syscall-wrapper.S 中末尾加入汇编语言接口：

```
WRAPPER(task_exit)
WRAPPER(task_create)
WRAPPER(task_getid)
WRAPPER(task_yield)
WRAPPER(task_wait)
WRAPPER(reboot)
WRAPPER(mmap)
WRAPPER(munmap)
WRAPPER(sleep)
WRAPPER(nanosleep)
WRAPPER(beep)
WRAPPER(vm86)
WRAPPER(putchar)
WRAPPER(getchar)
WRAPPER(recv)
WRAPPER(send)
WRAPPER(ioctl)
WRAPPER(time)
```

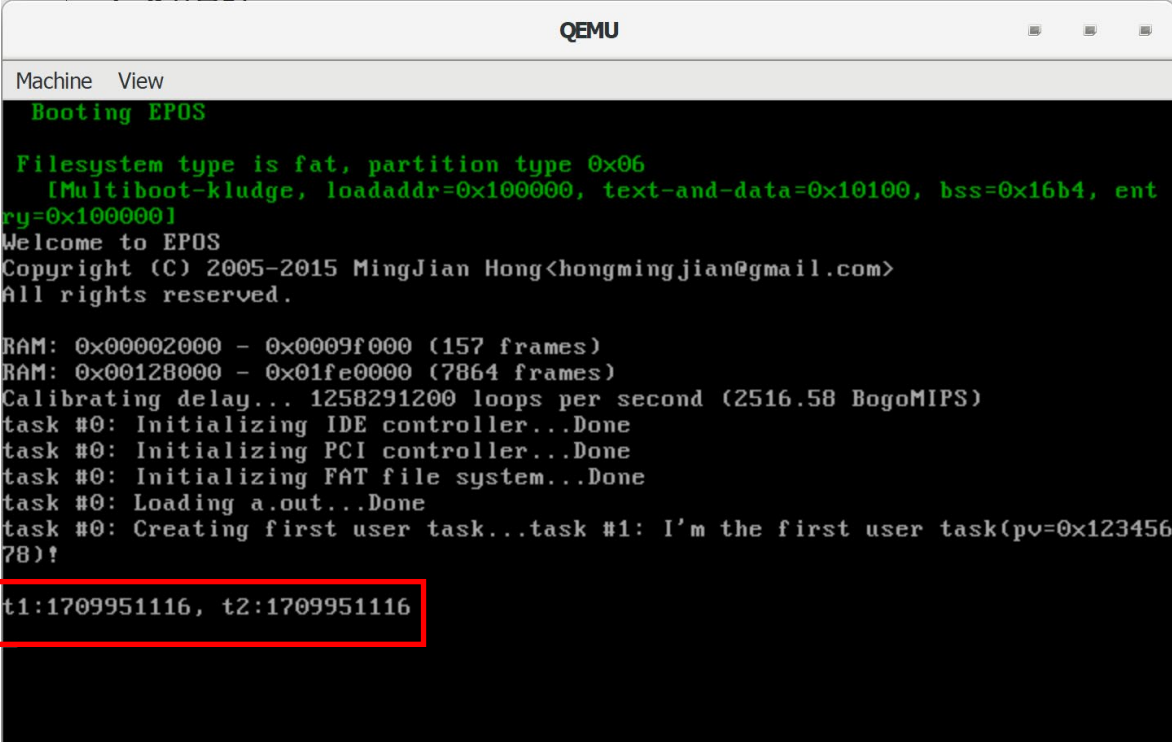
6、U2——加入声明

在 userapp/include/syscall.h 中加入 C 语言声明：

```
int    ioctl(int fd, uint32_t req, void *pv);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
/*U2 加入声明*/
time_t time(time_t *loc);
```

7、U3——调用系统调用，打印结果

在 userapp/main.c 中调用系统调用并打印结果：



```
QEMU
Machine  View
Booting EPOS
Filesystem type is fat, partition type 0x06
[Multiboot-kludge, loadaddr=0x100000, text-and-data=0x10100, bss=0x16b4, entry=0x100000]
Welcome to EPOS
Copyright (C) 2005-2015 MingJian Hong<hongmingjian@gmail.com>
All rights reserved.

RAM: 0x00002000 - 0x0009f000 (157 frames)
RAM: 0x00128000 - 0x01fe0000 (7864 frames)
Calibrating delay... 1258291200 loops per second (2516.58 BogoMIPS)
task #0: Initializing IDE controller...Done
task #0: Initializing PCI controller...Done
task #0: Initializing FAT file system...Done
task #0: Loading a.out...Done
task #0: Creating first user task...task #1: I'm the first user task(pv=0x12345678)!

t1:1709951116, t2:1709951116
```

在实验过程中遇到的问题：

1. 插入代码位置错误：

编译错误： 如果插入的代码位置不正确，可能会导致编译错误，因为它可能会破坏现有的代码结构或引入语法错误。

链接错误： 如果插入的代码与其他部分的链接方式不兼容，可能会导致链接错误，导致无法生成可执行文件。

运行时错误： 如果插入的代码影响了程序的执行流程或数据状态，可能会导致程序在运行时出现错误，例如内存访问错误、逻辑错误等。

功能不完整或未实现： 如果插入的代码不在正确的位置，可能会导致某些功能无法完全实现或者根本无法实现。

2. 调试过程中使用 **make clean**

删除生成的中间文件和目标文件：**make clean** 通常会删除项目中生成的中间文件（如 `.o` 文件）和目标文件（如可执行文件），这些文件是编译和链接过程的中间产物，删除它们可以释放磁盘空间。

清除编译环境：通过清除中间文件和目标文件，**make clean** 可以清除编译环境，以确保下次编译时从头开始，而不会受到之前编译结果的影响。