

**嵌入式操作系统课程设计报告**

|  |  |
| --- | --- |
|  |  |
| 姓 名： | 潘翔 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 物联网工程 |
| 班 级： | IOT1601 |
| 学 号： | U201614898 |
| 指导教师： | 石柯 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2018 年 7月 12日

**目 录**

[1 设计目的 1](#_Toc1115782452)

[2 文件拷贝 2](#_Toc1548934316)

[2.1 设计目的 2](#_Toc232988332)

[2.2 实验要求和内容 2](#_Toc356708768)

[2.3 环境及步骤 2](#_Toc930572190)

[2.3.1 开发环境 2](#_Toc1842241090)

[2.3.2 开发步骤 2](#_Toc1332719190)

[2.4 关键代码 4](#_Toc1522614422)

[2.4.1 copyF2F 4](#_Toc1368939827)

[2.4.2 copyD2D 5](#_Toc259508635)

[2.5 调试记录及运行结果 6](#_Toc965262464)

[2.6 设计感想 7](#_Toc450878224)

[3 图形化进程并发 8](#_Toc1970380693)

[3.1 实验要求和内容 8](#_Toc1882875440)

[3.2 实验过程与结果 8](#_Toc863605689)

[3.3 实验结果分析 8](#_Toc991585319)

[3.4 心得与体会 9](#_Toc882879904)

[4 添加系统调用 10](#_Toc1313250386)

[4.1 实验要求和内容 10](#_Toc709888989)

[4.2 实验步骤 10](#_Toc750950238)

[4.2.1添加源代码 10](#_Toc677429568)

[4.2.2 添加新的系统调用 11](#_Toc1360195109)

[4.2.3 编译Linux内核 11](#_Toc914812988)

[4.3 实验过程与结果 13](#_Toc21267230)

[4.3.1 查看内核版本 13](#_Toc626320372)

[4.3.2 获取内核源码 13](#_Toc2074796534)

[4.3.3 添加源代码 13](#_Toc1745514267)

[4.4 实验结果分析 15](#_Toc1784904179)

[4.5 心得与体会 15](#_Toc9323428)

[5 设备驱动 16](#_Toc1351776157)

[5.1 设计目的和内容 16](#_Toc500860308)

[5.2 环境及步骤 16](#_Toc1125105880)

[5.2.1 开发环境 16](#_Toc753226825)

[5.2.2 开发步骤 16](#_Toc733848640)

[5.3 设计实现及关键代码 16](#_Toc1481814648)

[5.3.1 编写设备驱动程序 16](#_Toc1683799016)

[5.3.2 设备驱动Makefile 16](#_Toc428606082)

[5.3.3 加载设备驱动模块 16](#_Toc667050190)

[5.3.4 设备驱动测试 16](#_Toc1058929790)

[5.5 设计感想 17](#_Toc1797545909)

[6 QT系统监控器 18](#_Toc926558825)

[6.1 设计目的 18](#_Toc2024192254)

[6.2 设计内容 18](#_Toc100940485)

[6.3 环境及步骤 18](#_Toc749455870)

[6.3.1 开发环境 18](#_Toc1759584046)

[6.3.2 运行环境 18](#_Toc964546174)

[6.3.3 开发步骤 18](#_Toc1741041190)

[6.4 设计实现及关键代码 19](#_Toc494980302)

[6.4.1 CPU 19](#_Toc130312912)

[6.4.2 Process 22](#_Toc303446531)

[6.4.3 Memory 23](#_Toc1245930540)

[6.4.4 Net 24](#_Toc807742480)

[6.4.5 FileSystem 24](#_Toc1663641640)

[6.6 调试记录及运行结果 25](#_Toc13259880)

[6.7 设计感想 28](#_Toc829009710)

[7 模拟文件系统设计 29](#_Toc142478365)

[7.1 设计目的 29](#_Toc2088056414)

[7.2 设计内容 29](#_Toc427040330)

[7.3 环境及步骤 29](#_Toc1927382544)

[7.3.1开发环境 29](#_Toc2097379842)

[7.3.2开发步骤 29](#_Toc1778816487)

[7.3.2系统结构 30](#_Toc280759204)

[7.4 内存版本设计实现 30](#_Toc1075002075)

[7.4.1 FreeNode 30](#_Toc384559664)

[7.4.2 Inode 31](#_Toc1014607844)

[7.4.3 DirEntry 31](#_Toc409333075)

[7.4.4 myFs(file system operation) 32](#_Toc2068358680)

[7.5 硬盘版本设计实现 33](#_Toc1443213926)

[7.5.1 Buffer 33](#_Toc1076383266)

[7.5.2 SuperBlock 35](#_Toc979804822)

[7.5.3 myFs(file system operation) 36](#_Toc1093276187)

[7.5 实验总结 38](#_Toc2002942091)

[参考文献 39](#_Toc856513428)

[附录 40](#_Toc1194216672)

[QTTest 40](#_Toc604914314)

[mycp 40](#_Toc468613826)

[mySystemCall 40](#_Toc11279199)

[myDevDriver 40](#_Toc198471856)

[myLinuxMonitor 40](#_Toc963594129)

[myFileSystem 40](#_Toc141592111)

[myFileSystem\_mem 40](#_Toc501918387)

# 

# 1 设计目的

1. 掌握Linux操作系统的使用方法
2. 了解Linux系统内核代码结构
3. 掌握实例操作系统的实现方法

# 2 文件拷贝

## 2.1 设计目的

熟悉和理解Linux编程环境

## 2.2 实验要求和内容

编写一个C程序，用read、write等系统调用实现文件拷贝功能。命令形式：

copy <源文件名> <目标文件名>

## 2.3 环境及步骤

### 2.3.1 开发环境

1. 操作系统：Arch Linux x64
2. 内核版本：4.18.5-arch1-1-ARCH
3. 编译工具：gcc (GCC) 8.2.0

### 2.3.2 开发步骤

1. 解析调用参数，判断copy类型，支持递归，软链接，硬链接
2. 检查参数数量是否正确
3. 解析参数，获取源文件地址和目标文件地址
4. 检查是否支持copy类型
5. 进行copy操作

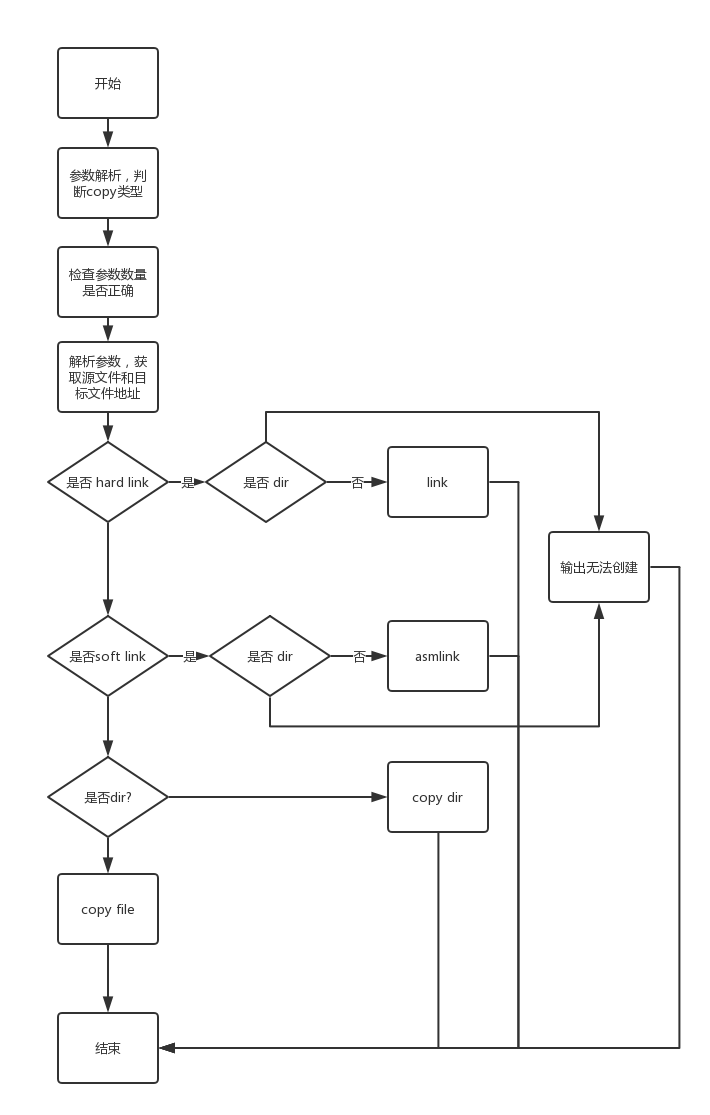


图2-1 copy函数流程图

## 2.4 关键代码

### 2.4.1 copyF2F

1. 函数原型

int copyF2F(char \*src\_file, char \*dest\_file)

1. 函数流程

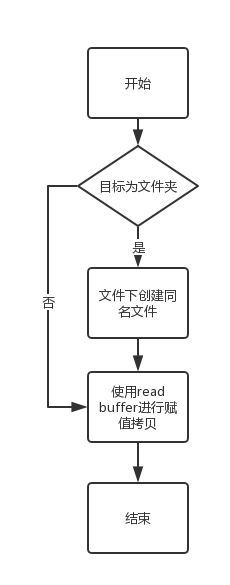


图2-2 copyF2F函数流程图

1. 关键代码

while ((n\_chars = read(in\_fd, buf, BUFFERSIZE)) > 0)

{

if (write(out\_fd, buf, n\_chars) != n\_chars)

{

printf("%s write file fail！", dest\_file);

return 1;

}

if (n\_chars == -1)

{

printf("%s read file fail！", src\_file);

return 1;

}

}

### 2.4.2 copyD2D

1. 函数原型

int copyD2D(char \*src\_dir, char \*dest\_dir)函数流程

1. 函数流程

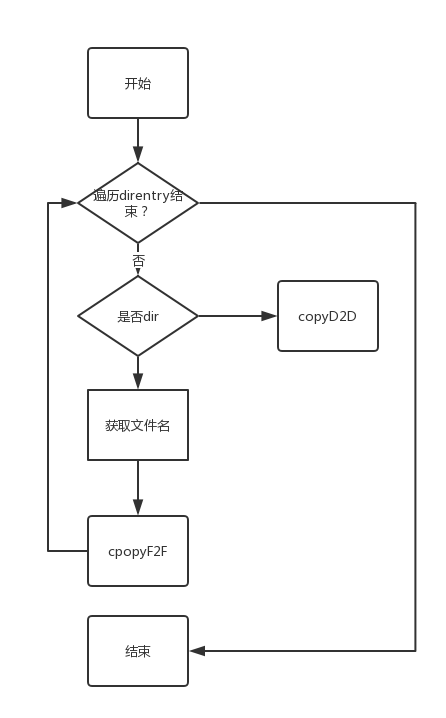


图2-3 copyD2D函数流程图

1. 关键代码

//open dir

if ((dp = opendir(src\_dir)) == NULL)

return 1;

else

{

//get dirent

while ((dirp = readdir(dp)))

{

struct stat file\_stat;

if (!isdir(dirp->d\_name))

{

//link name

strcat(tempDest, dirp->d\_name);

strcat(tempSrc, dirp->d\_name);

//copy file

copyF2F(tempSrc, tempDest);

//recover name

strcpy(tempDest, dest\_dir);

strcpy(tempSrc, src\_dir);

}

}

//close dir

closedir(dp);

return 0;

}

## 2.5 调试记录及运行结果



图2-4 copyF2F Test



图2-5 copyD2D Test

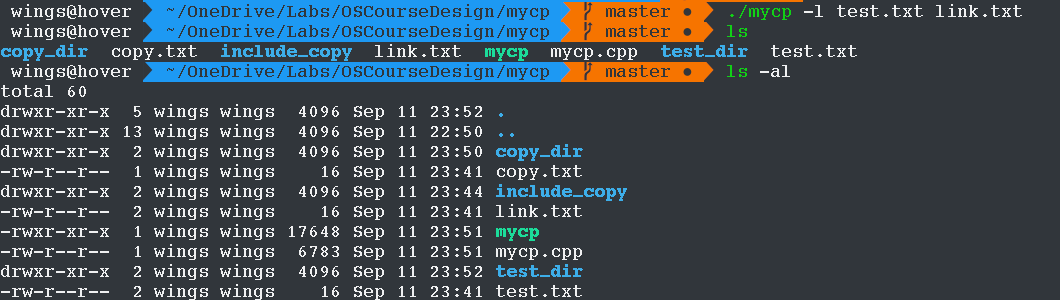


图2-6 link Test

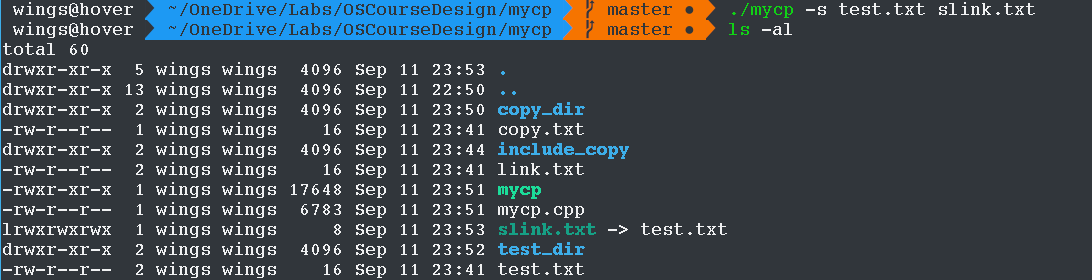


图2-7 syblink Test

## 2.6 设计感想

参照linux cp进行设计，其中D2D进行递归操作，类似于ls，需要注意栈空间的占用，设计过程中，学习不同Linux不同的文件格式，以及了解软链接和硬链接背后的实现过程，为后面文件系统做准备。

Linux的硬链接不允许为目录建立硬链接，但是链接可以存在递归，而软链接可以跨越文件系统，故提供了灵活的“拷贝”机制。

# 3 图形化进程并发

## 3.1 实验要求和内容

要求：熟悉和理解Linux编程环境

内容：

编写一个C程序，使用图形编程库 (QT/GTK)分窗口显示三个并发进程的运行(一个窗口实时显示当前系统时间，一个窗口循环显示0到9，一个窗口做1到1000的累加求和，刷新周期均为1秒)。

## 3.2 实验过程与结果

1. 利用fork进行进程并发
2. 利用QTimer进行定时，同事利用信号槽机制进行窗口间的状态同步

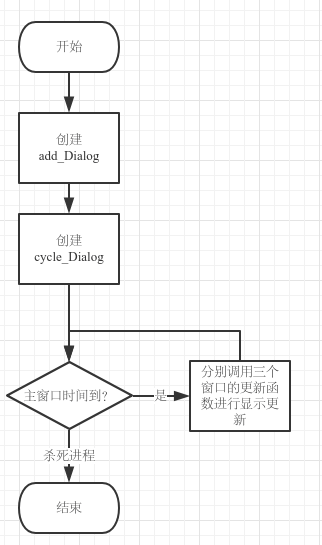


图3-1 三窗口流程图

## 3.3 实验结果分析

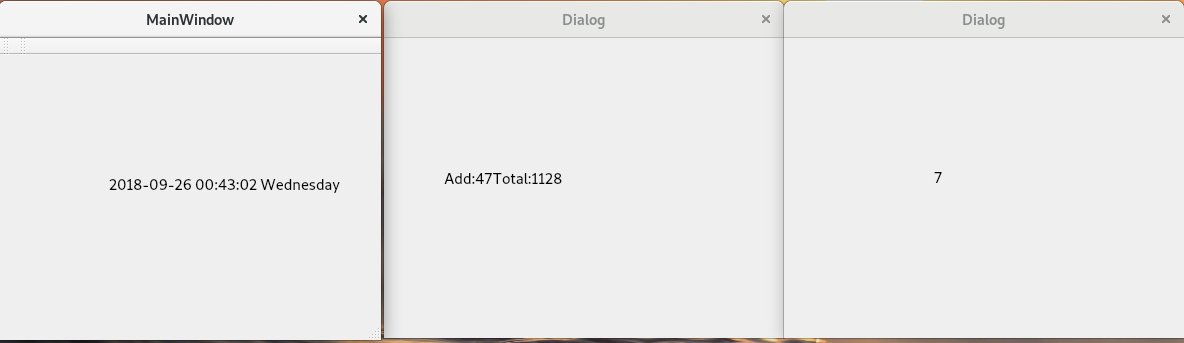


图3-2 三窗口测试

## 3.4 心得与体会

进行进程并发，Qt如果使用同一窗口，不同的widget那么不同的widget之间为三个线程，而如果为进程的话，可以创建三个Project在一个主Project中进行QProcess启动三个程序，或者直接利用fork进行进程创建。

# 4 添加系统调用

## 4.1 实验要求和内容

要求： 掌握添加系统调用的方法

内容:

采用编译内核的方法， 添加一个新的系统调用， 实现文件拷贝功能编写一个应用程序， 测试新加的系统调用

## 4.2 实验步骤

### 4.2.1添加源代码

编写添加到内核中的源程序，函数名以sys\_开头。

如：mycall(int num)，在arch/kernel/sys.c文件中添加如下调用源码：

SYSCALL\_DEFINE2(mycopy, const char \*, src, const char \*, dst)

系统调用分析：在arch中采用宏定义的方式提供了函数参数检查和一些安全性保护，其中如果想要返回某些数值，而在返回前已经设置了空间范围，宏将会自动完成返回值从内核空间到用户空间的转换。

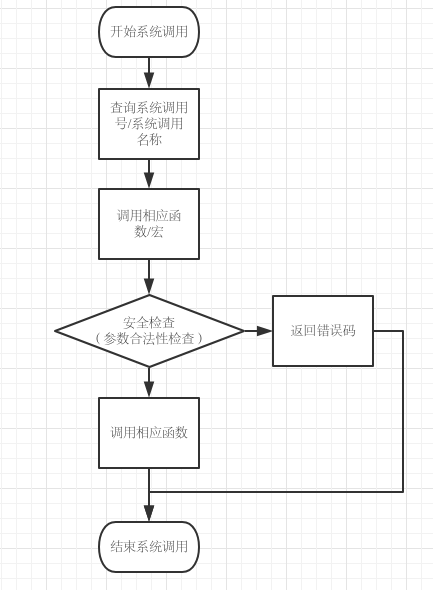


图4-1 系统调用分析

### 4.2.2 添加新的系统调用

在传统的Linux系统调用需要更改两个文件使内核的其余部分知道该系统调用的存在。:

* 1. include/linux/syscalls.h ——系统调用定义

增加新系统调用的函数定义

asmlinkage long sys\_mysyscall(int number);

* 1. arch/x86/syscalls/syscall\_64.tbl ——系统调用表

在系统调用表中为新增的系统调用分配一个系统调用号和系统调用名。

在arch中采用宏定义的方式进行系统调用的方式进行封装，故仅仅需要依据系统默认的规则进行系统调用的声明：

<number> <abi> <name> <entry point>

其中，\_\_x64为宏定义中的最终的接口形式，当使用宏定义的时候，最终一层的系统调用入口与为\_\_64\_sys。

### 4.2.3 编译Linux内核

在内核的编译过程中，arch提供了两种机制，一种是传统的内核编译机制，在内核完整编译之后修改Boot进行启动，另一种基于arch的包管理机制，从源码编译生成包，然后卸载原内核（包），安装新生成的包，此处采用传统编译方式

1. 编译内核

make

1. 编译内核模块

Make modules

1. 生成内核配置文件

make menuconfig

此处可以修改不同的驱动配置，文件系统配置，否则生成出的为裸内核，不 支持任何外界设备以及默认的集成显卡驱动，同时系统本身带有内核的配置 文件，可以将其拷贝过来。

1. 编译内核映像

make bzImage

1. 编译内核模块

make modules

1. 生成并安装模块

make modules\_install

1. 安装新的系统

make install

1. 重启，选择新修改的内核
2. 编写应用程序，测试新增系统调用

## 4.3 实验过程与结果

### 4.3.1 查看内核版本

cat /proc/version



图4-1 查看内核版本

### 4.3.2 获取内核源码

访问<https://www.kernel.org/> 下载4.18.5版本内核

### 4.3.3 添加源代码

在arch/kernel/sys.c中添加调用服务例程定义

1. 添加调用函数声明

在include/linux/syscalls.h中添加调用函数声明

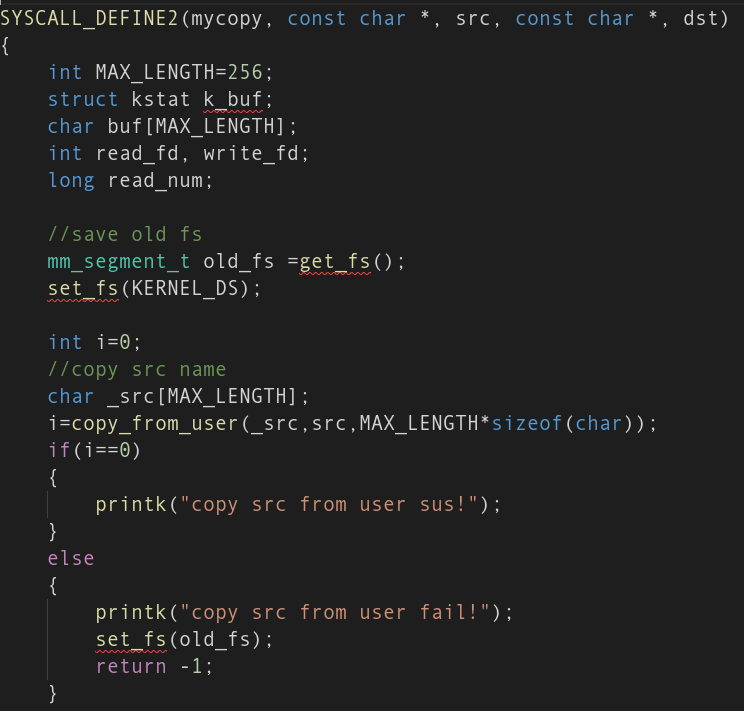


图4-2 系统调用声明截图

1. 系统调用表

在/arch/x86/entry/syscalls/syscall\_64.tbl 系统调用表中为新增的系统调用分配一个系统调用号和系统调用名。



图4-3 系统调用表截图

1. 编译内核

KERNEL\_VERSION="418"

# set compile arg

kernel\_num=8

# Back edited file

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

cp kernel/sys.c ../

# make image

# make mrproper

# make menuconfig

make bzImage –j $KERNEL\_VERSION

# make modules

make modules –j $kernel\_num

make modules\_install –j $kernel\_num

# make install

make install –j $kernel\_num

# copy kernel image to boot

cp arch/x86\_64/boot/bzImage /boot/vmlinuz-linux$KERNEL\_VERSION

# build initramfs

mkinitcpio -p linux$KERNEL\_VERSION

# update-grub

grub-mkconfig -o /boot/grub/grub.cfg

## 4.4 实验结果分析

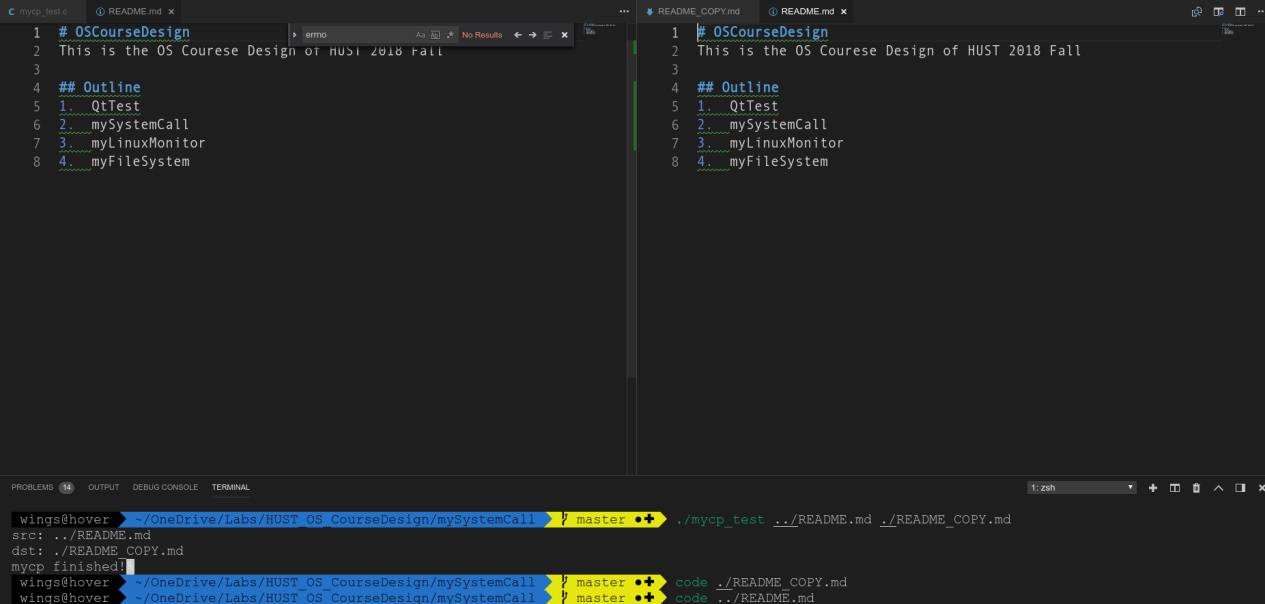


图4-4 系统调用测试

## 4.5 心得与体会

整个内核实验过程中，学习了《Linux内核分析》学习使用GDB进行内核函数的跟踪调试，而通常使用printk和errno错误代码记录也是较为有效的方式。

许多教程采用了每次编译后make clean/make mrproper的方式，导致编译时间很长，而实际上因为采用增量编译，则make仅仅需要对修改的部分进行重新编译和链接，故直接make就好，通常不会出现问题，这在一些较大的工程中是常采用的方式，同时 -j $kernel\_num在make执行过程中可以同时执行的指令数目，相当于并行，在有大量独立模块且依赖较上的时候，能够大大加快底层的编译速度。

开始进行内核调用的时候参考传统教程，修改系统sys.h，sys.c和syscall\_64.tbl。而对于ArchLinux采用宏定义的方式包装相应函数，目的是提供一定的安全性机制，包括参数检查，返回值的空间转换。

而当使用宏定义后，系统自动生成的函数调用接口为\_\_sys\_x64格式，此时对于仍然保留sys.h中函数定义，发现并不会冲突，因为函数入口的格式变化，这种机制保证了新旧函数调用的过渡和向后兼容，可以看到，在ArchLinux中仍抱有少量的旧式函数调用。

# 5 设备驱动

## 5.1 设计目的和内容

要求： 掌握添加设备驱动程序的方法

内容：

采用模块方法， 添加一个新的字符设备驱动程序， 实现打开/关闭、 读/写等基本操作

写一个应用程序， 测试添加的驱动程序

## 5.2 环境及步骤

### 5.2.1 开发环境

1. 操作系统：Arch Linux x64
2. 内核版本：4.18.5-arch1-1-ARCH
3. TextEditor: Visual Studio Code
4. 编译工具： gcc (GCC) 8.2.0

### 5.2.2 开发步骤

1. 编写设备驱动程序mydev.c
2. 设备驱动模块的编译 Makefile文件的使用
3. 加载设备驱动模块: insmod mydev.ko
4. 生成设备文件： mknod /dev/test c 254 0

## 5.3 设计实现及关键代码

### 5.3.1 编写设备驱动程序

Linux设备驱动程序利用file\_operations提供调用接口，当程序调用设备驱动程序对设备进行操作的时候，通过该结构体接口，将相应操作交给设备驱动程序的函数，完成操作。

struct file\_operations pStruct=

{

open : my\_open,

release : my\_release,

read : my\_read,

write : my\_write,

};

### 5.3.2 设备驱动Makefile

设备驱动独立于内核便于修改，但编写好的设备驱动需要build进最终的内核所在的文件夹，才能够调用系统命令进行设备驱动的装载。2

ifeq ($(KERNELRELEASE),)

KERNELDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

modules:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules\_install:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules\_install

clean:

rm -rf \*.o .depend .\*.cmd \*.ko \*.mod.c .tmp\_versions modules.\*

else

obj-m := myDevDriver.o

endif

### 5.3.3 加载设备驱动模块

insmod myDevDrive.ko

图5-1 设备驱动装载结果

### 5.3.4 设备驱动测试

图5-2 设备驱动测试结果

## 5.5 设计感想

# 6 QT系统监控器

## 6.1 设计目的

1. 了解/proc文件的特点和使用方法
2. 监控系统状态，显示系统部件的使用情况
3. 用图形界面监控系统状态，包括CPU和内存利用率、所有进程信息等(可自己补充、添加其他功能)

## 6.2 设计内容

1. 监控系统功能：通过读取proc文件系统，获取系统各种信息，并以比较容易理解的方式显示出来
2. C语言开发，图形界面直观展示
3. 参照WINDOWS的任务管理器，实现其中的部分功能

主机名、系统启动时间、系统运行时间、版本号、所有进程信息、CPU类型、 CPU的使用率、内存使用率

## 6.3 环境及步骤

### 6.3.1 开发环境

1. 操作系统： Arch Linux x64
2. 内核版本： 4.18.5-arch1-1-ARCH
3. IDE: Qt Creator 4.7.0(Based onQt5.11.1)
4. 编译工具： gcc (GCC) 8.2.0

### 6.3.2 运行环境

基于Qt跨平台特性，可运行于基于Linux的平台

### 6.3.3 开发步骤

1. 创建窗口，进行初始化
2. CPU:
   1. 获取CPU的时间信息，利用间隔时间进行CPU利用率计算
   2. 将新的CPU信息点加入图表，进行刷新
3. Process
   1. 定时读取进程信息
   2. 维护proc内存池，进行process列表更新
   3. 进行table展示的更新
   4. 设置焦点和当前所在的页面
4. Net:
   1. 获取上一秒和当期秒当前网卡的数据包信息
   2. 进行数据负载的计算
   3. 维护chart展示，计算60s之类的峰值，进行表格的适当展示
5. Mem:
   1. 读取内存信息
   2. 维护chart图表

## 6.4 设计实现及关键代码

### 6.4.1 CPU

1. 数据结构描述
   1. cpu\_info

/\*

\* utime: user

\* stime: system time

\* ntime: nice time, the time for modefiy the priority of cpu

\* itime: idle time

\* iowtime: io waiting time

\* irqtime: interuption time

\* sirqtime: soft interuption time

\*/

struct cpu\_info

{

long unsigned utime, ntime, stime, itime;

long unsigned iowtime, irqtime, sirqtime;

};

* 1. proc\_info

struct proc\_info

{

struct proc\_info \*next;

pid\_t pid;

pid\_t tid;

uid\_t uid;

gid\_t gid;

char name[PROC\_NAME\_LEN];

char tname[THREAD\_NAME\_LEN];

char state;

long unsigned utime;

long unsigned stime;

long unsigned delta\_utime;

long unsigned delta\_stime;

long unsigned delta\_time;

long vss;

long rss;

int num\_threads;

char policy[32];

};

* 1. proc\_list

struct proc\_list

{

struct proc\_info \*\*array;

int size;

};

1. 更新过程

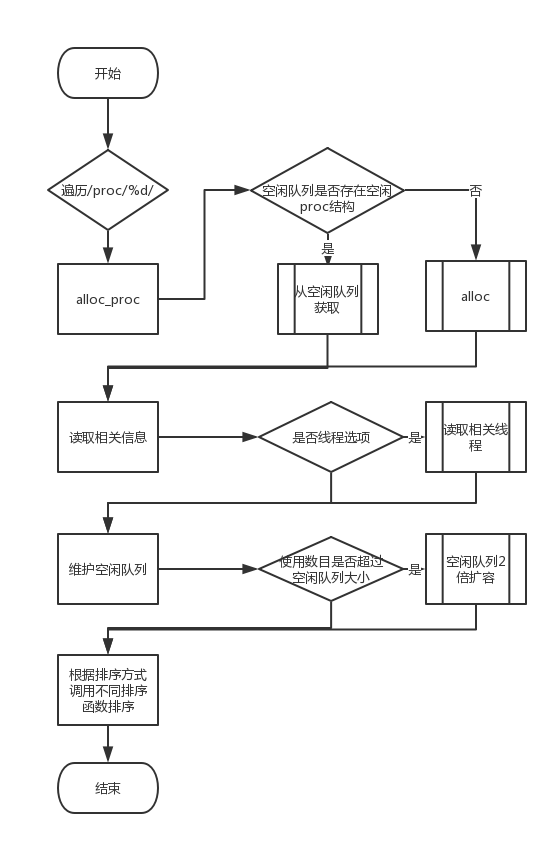


图5-1 CPU信息更新过程流程图

1. 利用率计算
   1. 两次更新之间的总时间Total\_delta\_time

total\_delta\_time =

(new\_cpu.utime + new\_cpu.ntime + new\_cpu.stime+new\_cpu.itime+ new\_cpu.iowtime + new\_cpu.irqtime + new\_cpu.sirqtime)-

(old\_cpu.utime + old\_cpu.ntime + old\_cpu.stime + old\_cpu.itime + old\_cpu.iowtime + old\_cpu.irqtime + old\_cpu.sirqtime);

* 1. 系统态和用户态时间计算

cpu\_user= ((new\_cpu.utime + new\_cpu.ntime) - (old\_cpu.utime + old\_cpu.ntime)) \* 100 / total\_delta\_time;

cpu\_sys= ((new\_cpu.stime ) - (old\_cpu.stime)) \* 100 / total\_delta\_time;

1. 图表绘制

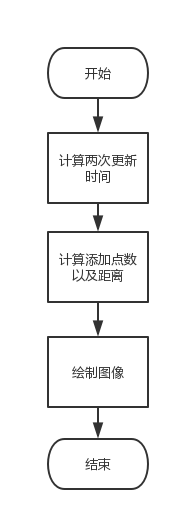


图5-2 CPU绘图过程流程图

### 6.4.2 Process

1. 更新列表
2. 列表维护
3. 列表绘制

保存更新前的焦点pid，获取更新后的表格行数，进行焦点设置，利用表格行数除以总行数获取页面相对位置，进行页面位置设置。

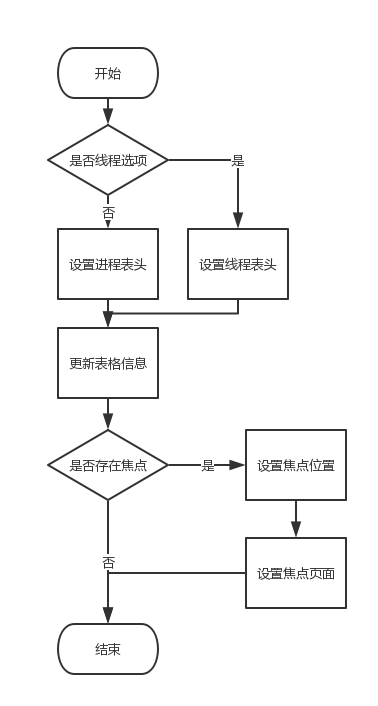


图5-3 CPU模块流程图

### 6.4.3 Memory

1. 更新内存信息
   1. 打开/proc/meminfo文件
   2. 定位不同的标识符，更新内存信息
   3. 关闭文件
2. 利用率计算

在Linux内存管理中，存在已经释放但是仍然存在与内存的缓存，故MemFree<MemAvailable,故进行计算利用率时采用MemTotal-MemAvailable作为MemUsed

1. 图表绘制

绘制内存曲线图和饼图

1. 双击触发

当点击表格，使用nautiulus系统调用打开对应的文件系统。

### 6.4.4 Net

1. 读取网络信息

读取/proc/net/dev中的网卡信息

1. 利用率计算

计算两个时间差之间的包数量，从而近似计算当前网络的利用率

1. 图表绘制

需要较为合适的绘制速率曲线，计算60s内大于峰值的最小模512整数作为 y轴最高值进行曲线绘制

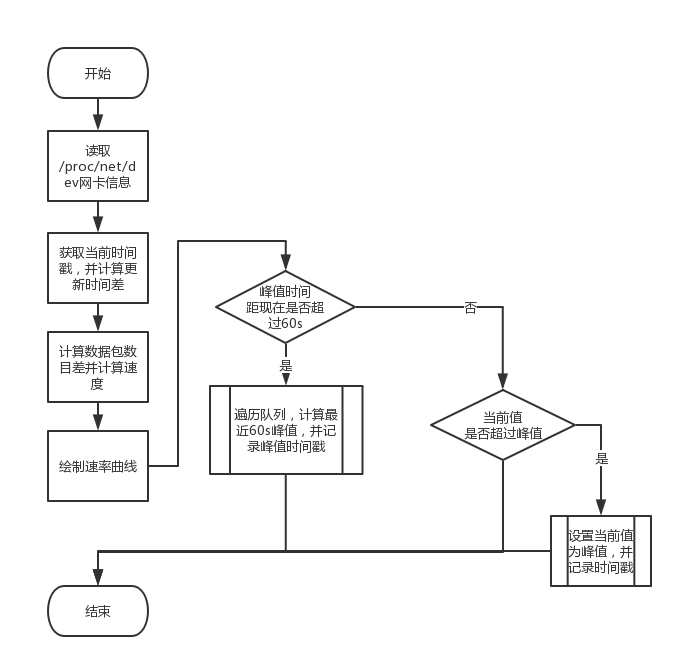


图5-4 Net模块流程图

### 6.4.5 FileSystem

1. 读取/etc/mtab信息，并判断非交换文件和临时文件系统
2. 更新FileSystablelist，并计算利用率
   1. 硬盘整个的空间大小不等于blocks数目，还含有inode节点和前面的空余信息，故整个硬盘的大小为disktotal=used+avail
   2. blocks\_percent\_used=blocks\_used/disktotal;
3. 调用绘图绘制利用率曲线图

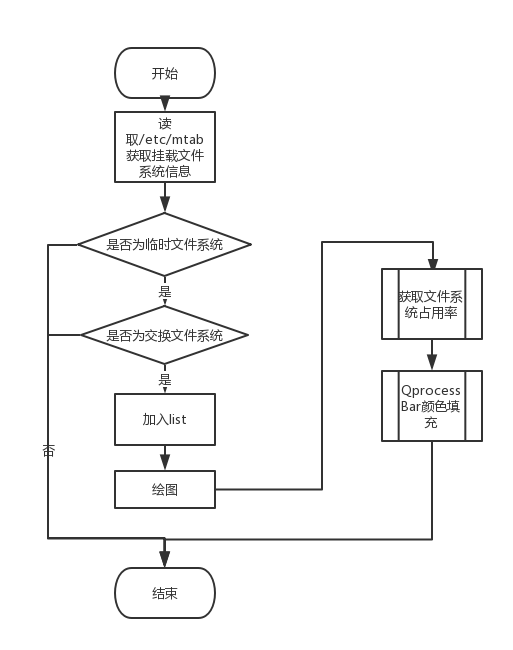


图5-5 FileSystem模块流程图

## 6.6 调试记录及运行结果

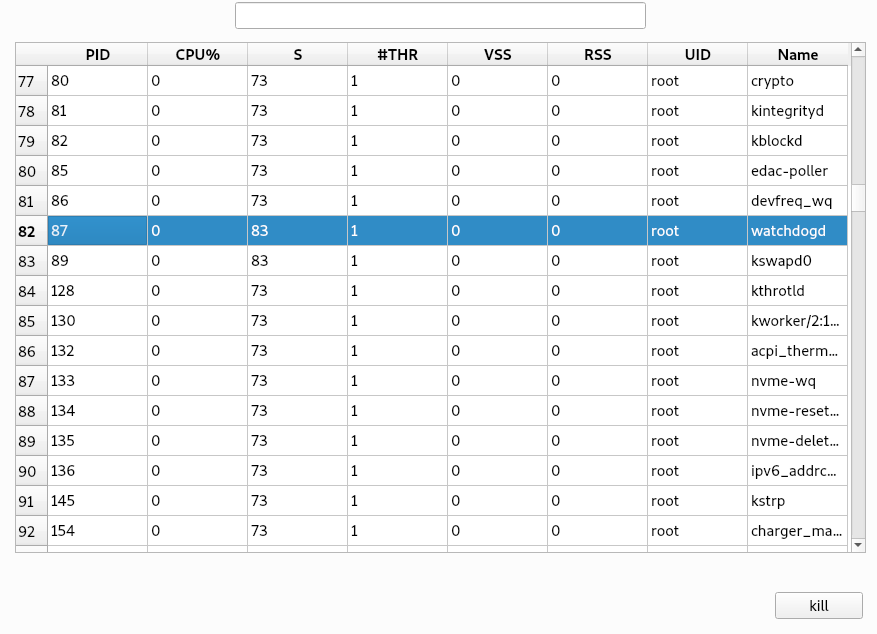


图5-6 Process模块排序测试图

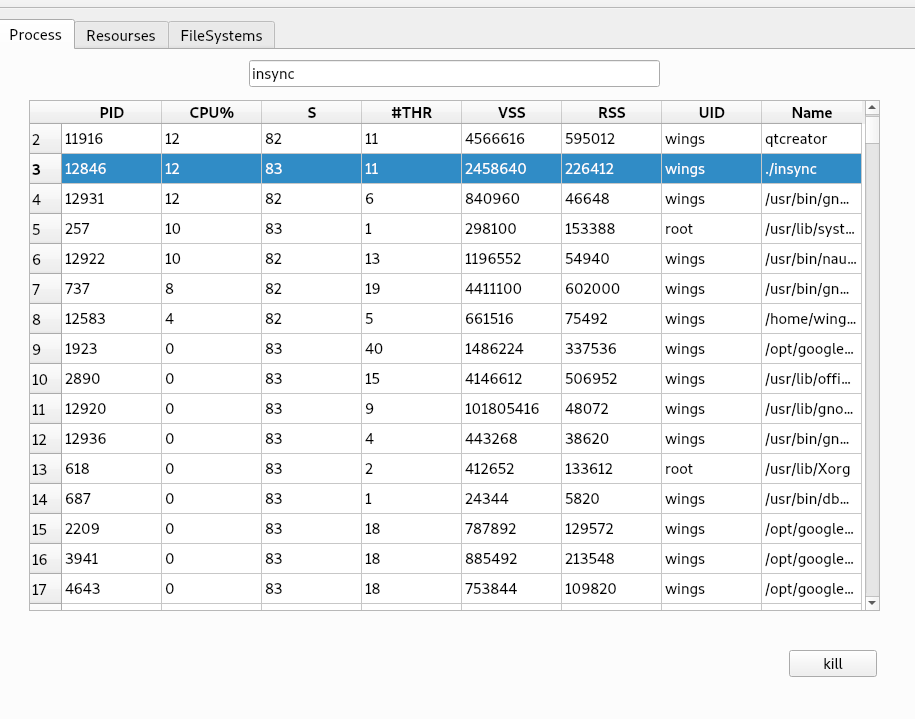


图5-7 Process模块搜索测试图

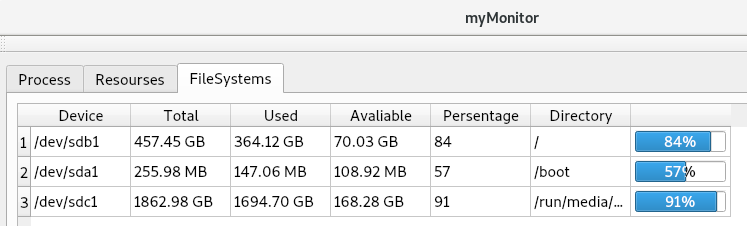


图5-7 FileSystem模块外接设备测试图

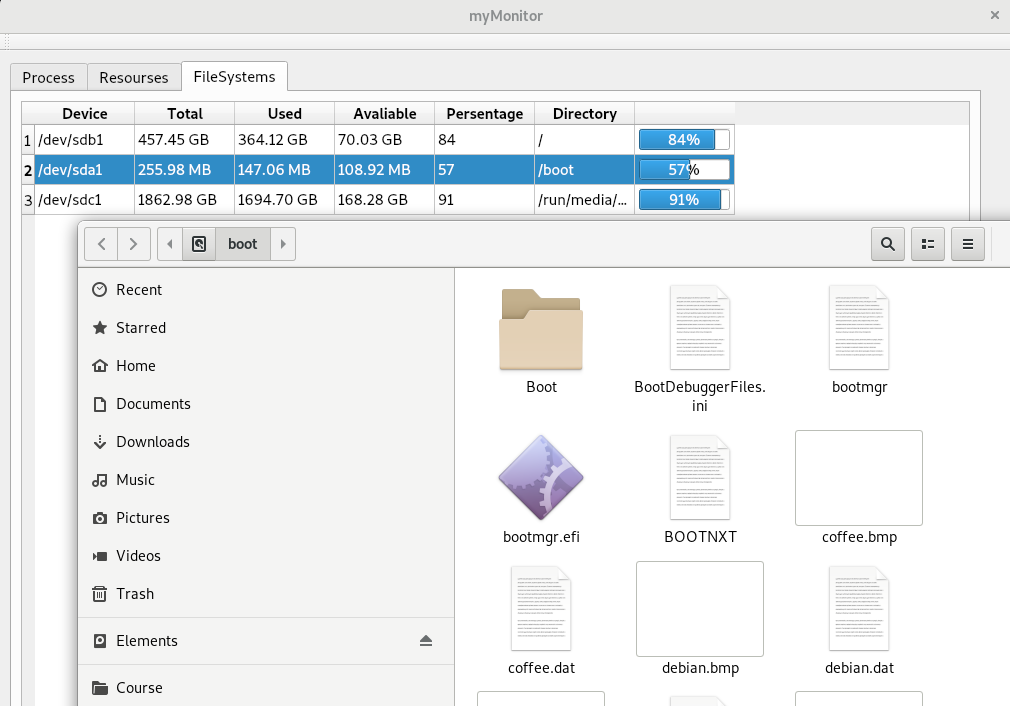


图5-7 FileSystem模块双击打开测试图



图5-7 资源管理器模块测试图

## 6.7 设计感想

设计过程中，参考GNOME-MONITOR，使用Qt的定时器机制进行操作，定时更新，因为Process和绘制图像的即时性不同，故采用不同的定时时长，进行更行。

更新机制上，获取数据部分和绘图部分分离，可以采用不同的定时器时长，在绘图的间隙中间进行数据的获取，减少卡顿。

Process表格维护上，采用了内存池机制，避免频繁的内存申请和释放，同时尝试了原生的model和自己手动实现了焦点追踪进行比较，因为原生的model进行的实现的时候，对于添加项的数据如果已经存在，仅仅进行update操作，故所在的item并不会有构造和销毁操作，焦点不会丢失，同时数字不会闪烁，尝试手动实现，计算焦点所在的列数

Net表格绘制，为了维护适当的坐标轴上的最大刻度，维护60s内的峰值，并设置过时机制，使能够尽可能的合理显示网络曲线。

FileSystem模块，因为QProcessBar为widget的子类，故不能嵌套进item之下，故采用自己继承原UI类进行重载，手写绘图函数进行块填充，完成图像的绘制。

# 7 模拟文件系统设计

## 7.1 设计目的

熟悉Linux文件系统

## 7.2 设计内容

1. 用磁盘中的一个文件（大小事先指定）来模拟一个磁盘
2. 确定文件目录项的结构
3. 空闲块的管理（每个块＝连续的Ｎ个文件字节）
4. 扩充系统调用命令实现文件的操作：open、 close、 read、 write、 cp、 rm等

## 7.3 环境及步骤

### 7.3.1开发环境

1. 操作系统： Arch Linux x64
2. 内核版本： 4.18.5-arch1-1-ARCH
3. 编译器： gcc (GCC) 8.2.0
4. 编译工具： CMake 3.12.1
5. 编辑器: Visual Studio Code

### 7.3.2开发步骤

1. 初始化文件系统，并对相应的类
2. 载入superblock，同时输出界面信息
3. 对于相应的操作，进行相应的读写操作
4. 函数退出，将缓存写回

### 7.3.2系统结构

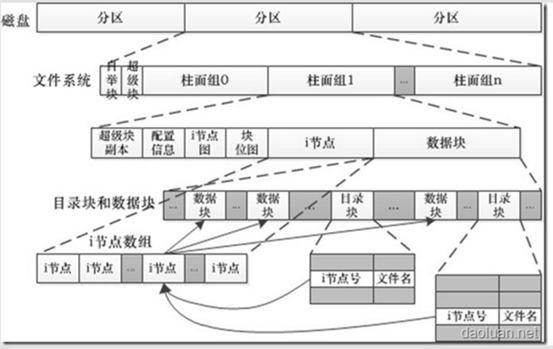


图7-1 系统结构图

## 7.4 内存版本设计实现

### 7.4.1 FreeNode

1. 类描述

所删除的文件之后，其inode和数据块需要同时被释放，此时将其打包为FreeNode，故当下一次需要创建文件，先从FreeNodeList中进行寻找，减少了inode类构造和销毁的开销，同时减少创建文件需要遍历位图的开销

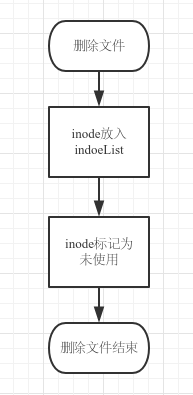


图7-2 删除文件操作

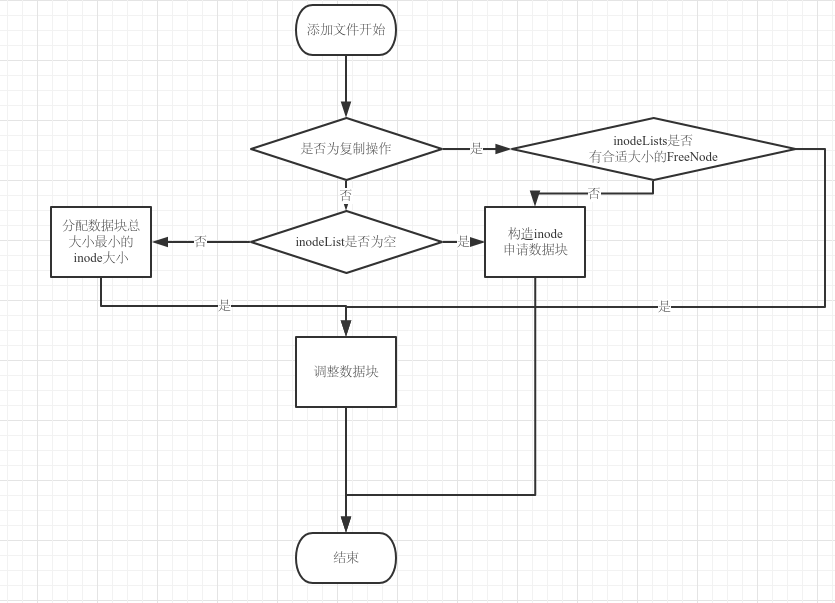


图7-3 添加文件操作

1. 关键成员

数据块个数： uint block\_num;        //free node num

起始数据块个数：uint pos;            //start pos

### 7.4.2 Inode

1. 类描述

Inode类，索引节点，记录文件相关信息，包括inode号，创建时间，文件类型等。文件的数据块有两种形式，一种是采用数据块串联链表形式，因为流式文件多为顺序读取，另一种直接在ionde中存储指针数组，此处采用后者。

1. 关键成员

Inode号： uint inode\_num;

起始数据块位置：uint sec\_num;

### 7.4.3 DirEntry

1. 类描述

文件的数据块类，包含文件的类型（数据/目录）及相应的构造函数接口，利用c++的lock机制提供访问保护。

1. 关键成员
   1. 目录项/数据项构造函数

static shared\_ptr<DirEntry>

make\_de\_dir(const string name,const shared\_ptr<DirEntry> parent);

static shared\_ptr<DirEntry>

make\_de\_file(const string name,const shared\_ptr<DirEntry> parent,

const shared\_ptr<Inode> &inode=nullptr);

* 1. 目录入口

weak\_ptr<DirEntry> parent; // .

weak\_ptr<DirEntry> self; // ..

list<shared\_ptr<DirEntry>> contents; // dir entry

* 1. 锁变量

bool is\_locked; // lock

* 1. 添加子项操作

shared\_ptr<DirEntry> find\_child(const string name) const;

shared\_ptr<DirEntry> add\_dir(const string name);

shared\_ptr<DirEntry> add\_file(const string name);

### 7.4.4 myFs(file system operation)

1. 类描述

相应的界面输出函数以及参数解析，同时利用底层类的接口实现相应的常用文件操作，同时维护当前打开的文件及其描述信息。

1. 关键成员
   1. 打开文件描符

struct Descriptor

{

Mode mode; // open mode

uint byte\_pos; //now pos

weak\_ptr<Inode> inode;

weak\_ptr<DirEntry> from;

uint fd;

};

* 1. 当前位置描述符

struct PathRet

{

bool invalid\_path = false;

string final\_name;

shared\_ptr<DirEntry> parent\_node;

shared\_ptr<DirEntry> final\_node;

};

* 1. 基础读/写操作

真正的读写操作，向上提供服务接口，不提供保护功能

bool basic\_open(Descriptor \*d, vector <string> args);

unique\_ptr<string> basic\_read(Descriptor &desc, const uint size);

uint basic\_write(Descriptor &desc, const string data);

bool basic\_close(uint fd);

1. 操作函数

所有操作函数接口包括：

文件操作：open,read,write,seek.close,

文件夹操作：mkdir,rmdir

文件系统信息查看：cd,stat,ls,cat,pwd,tree,

本地文件系统交互：import,FS\_export

文件拷贝：link,unlink,cp

## 7.5 硬盘版本设计实现

### 7.5.1 Buffer

1. 类描述

提供cache机制,向上提供虚拟读写接口，供功能函数进行调用，在进行调用时，维护bufferList，每一次调用将更新当前读写块的优先级，然后调用底层的读写接口进行真正的磁盘读写操作

1. 关键成员
   1. 函数接口

bool write\_disk(const BufferNode& node);

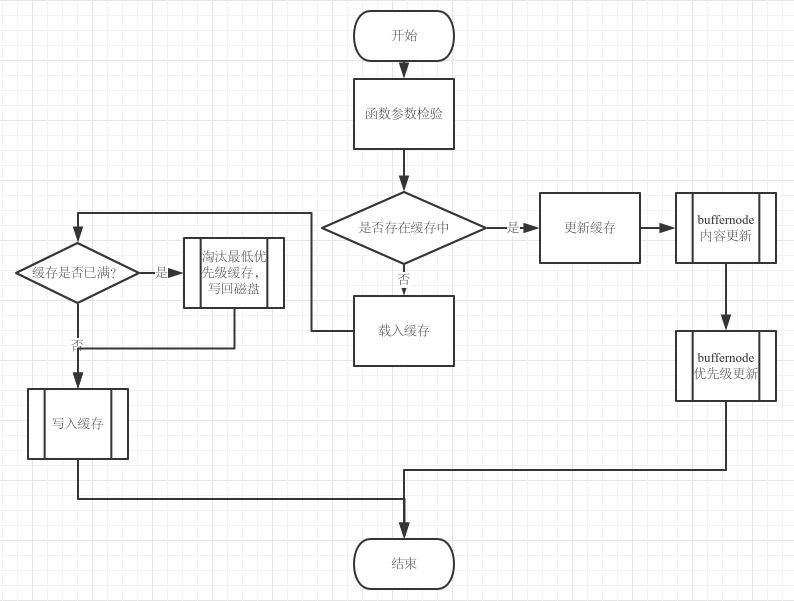


图7-3 写磁盘操作流程图

bool read\_disk(int sec\_num, BufferNode& node);

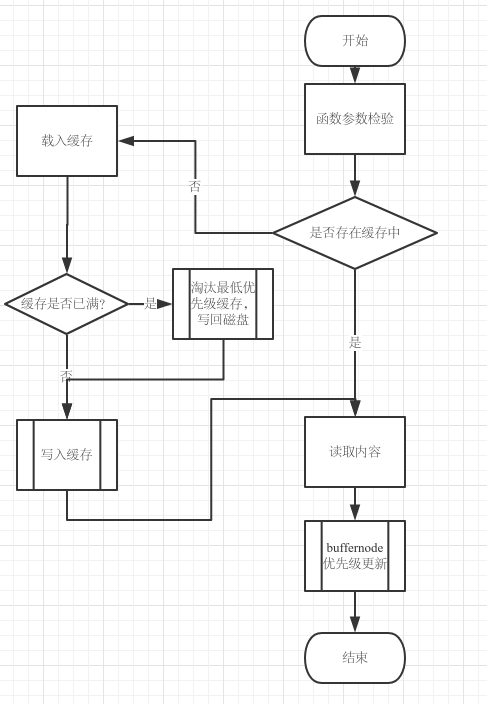


图7-4 读磁盘操作流程图

void all\_write\_to\_disk();

* 1. 底层操作函数

bool real\_disk\_write(const BufferNode& node);

bool real\_disk\_read(int sec\_num, BufferNode& node);

int has\_sec(int sec\_number);

int is\_full();

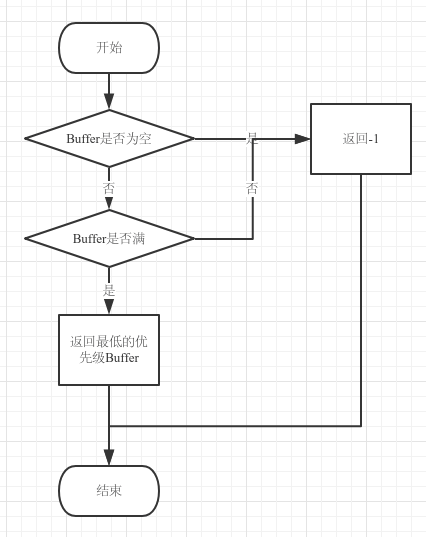


图7-5 查找最低优先级缓冲流程图

### 7.5.2 SuperBlock

1. 类描述

维护当前文件系统inode位图和block位图，同时提供inode和block的检索函数，获取到空闲的块

1. 关键成员

Inode位图：bool inode\_bitmap[INODE\_NUM];

Block位图：bool block\_bitmap[BLOCK\_NUM];

磁盘文件： fstream disk;

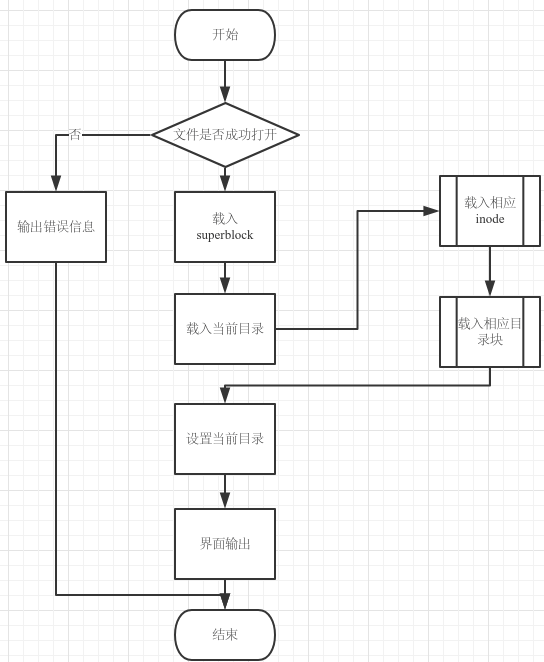


图7-6 系统初始化流程图

### 7.5.3 myFs(file system operation)

1. 类描述

相应的界面输出函数以及参数解析，同时利用底层类的接口实现相应的常用文件操作，同时维护当前打开的文件及其描述信息。

1. 关键成员

Vim:使用vim编辑器编辑文件：

图7-7 vim系统调用流程图

## 7.5 实验总结

开始了解到关于内核编译文件系统模块，所参考的教程较老，且内核编译较为复杂，故不采用，后学习FUSE，但调用的为系统底层API，上层加壳实现其他的目的（如网络文件系统、音乐库文件系统），故最后采用手动编写文件系统。

模拟文件系统类似资源管理工具，如游戏引擎中的资源管理器，本质为磁盘和内存之间信息的交换，故文件系统中遇到的问题也大多相似：脏数据，减少读写次数和读写频率，内存空间限制，涉及到操作系统中关于虚拟存储的常见算法：如缓存调度，其设计思想终究为目录结构，仅仅对于特定的目录内容进行载入。

在编写过程中，考虑参数传入，使用C++ STL模板类进行传参，同时对于参数的的个数进行动态的解析，在每一个函数的入口进行参数个数判断，最少参数数目及最多参数数目，并对操作对象和操作方法之间的合法性进行检验，确保程序的鲁棒性。

对象的持久化过程中，尝试了两种思路，一种是使用Boost库进行对象持久化，但因为在Class中采用变长的vector容器进行数据的存储，故仅仅在文件系统读取和最终关闭的时候进行IO,同时需要占用较大的内存空间，故采用定长的分块管理机制。

性能优化上，采用了FreeNodeList对释放的Node节点进行缓存，类似于内存池机制，便于下一次malloc，实现了自己的malloc机制。在硬盘I/O上，采用Buffer队列进行缓冲，对外提供read/write接口，对硬盘实现read\_disk/write\_disk操作，完成了缓冲机制，减少了硬盘I/O，提高了读写效率。

# 参考文献

1. Qt官方文档
2. Unix环境高级编程
3. Linux内核分析[MOOC]中国科技大学
4. Writing-a-linux-kernel-module

<http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/>

1. Boost库Tutorial

<https://www.boost.org/doc/libs/1_57_0/libs/filesystem/doc/tutorial.html>

# 附录

## QTTest

## mycp

## mySystemCall

## myDevDriver

## myLinuxMonitor

## myFileSystem

## myFileSystem\_mem