

华中科技大学

课程设计报告

题目： 操作系统课程设计

课程名称： 操作系统

专业班级： 信息安全 1503 班

学 号： U201514858

姓 名： 陈 薇

指导教师： 胡 侃

报告日期： 2018 年 3 月 10 日

计算机科学与技术学院

目 录

1	设计目的	1
2	设计内容	1
2.1	课设内容 1.....	1
2.2	课设内容 2.....	1
2.3	课设内容 3.....	1
2.4	课设内容 4（选做）.....	1
2.5	课设内容 5（选做）.....	2
3	设计过程	3
3.1	文件拷贝和进程并发.....	3
3.1.1	文件拷贝.....	3
3.1.2	进程并发.....	4
3.2	添加系统调用	6
3.2.1	编译内核.....	6
3.2.2	添加系统调用.....	9
3.2.3	系统调用的实现和测试.....	10
3.3	添加设备驱动程序	11
3.3.1	主设备号和次设备号.....	11
3.3.2	文件操作和进程并发.....	12
3.3.3	模块代码流程.....	13
3.3.4	测试程序.....	17
3.4	系统监控	17
3.4.1	/proc 文件系统	17
3.4.2	Qt 类	21
3.5	模拟文件系统设计	22
3.5.1	总体设计.....	22
3.5.2	文件系统的格式化和 mkfs 的实现.....	24
3.5.3	文件系统的实现.....	26
4	设计环境	35
5	实现记录	36
5.1	文件拷贝和进程并发.....	36
5.1.1	文件拷贝.....	36
5.1.2	进程并发.....	37
5.2	添加系统调用	38

5.2.1	代码编写和内核编译.....	38
5.2.2	系统调用测试.....	40
5.3	添加设备驱动程序.....	40
5.3.1	编译驱动程序.....	40
5.3.2	测试模块功能.....	41
5.4	系统监控.....	43
5.5	文件系统.....	46
6	设计心得	47

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 设计内容

2.1 课设内容 1

- 1、要求：掌握 Linux 的使用方法，熟悉和理解 Linux 编程环境。
- 2、内容
 - (1) 编写一个 C 程序，用 fread、fwrite 等库函数实现文件拷贝功能；
 - (2) 编写一个 C 程序，用 QT 或 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时间，一个窗口显示/etc/fstab 文件的内容，一个窗口显示 1 到 1000 的累加求和过程）。

2.2 课设内容 2

- 1、要求：掌握添加系统调用的方法。
- 2、内容
 - (1) 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；
 - (2) 编写一个应用程序，测试新加的系统调用。

2.3 课设内容 3

- 1、要求：掌握添加设备驱动程序的方法。
- 2、内容
 - (1) 采用模块添加的方法，添加一个新的字符设备的驱动程序，实现打开/关闭、读/写等基本操作；
 - (2) 编写一个应用程序，测试添加的驱动程序。

2.4 课设内容 4（选做）

- 1、要求：理解和分析/proc 文件。
- 2、内容
 - (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)

2.5 课设内容 5（选做）

1、要求：理解和掌握文件系统的设计方法。

2、内容

(1) 设计、实现一个模拟的文件系统；

(2) 多用户、多级目录、用户登录；

(3) 文件/目录的创建/删除，目录显示，文件打开/读/写/关闭等基本功能；

(4) 可自行扩充权限控制、读写保护等其他功能。

3 设计过程

本次课程设计已检查 1-4 题，第 5 题由于进度原因未能按时检查，将实验过程和实验结果其写入报告中，下面依次介绍各内容的设计过程。

3.1 文件拷贝和进程并发

3.1.1 文件拷贝

该题目要求利用库函数实现文件拷贝功能，具体设计思路如下图所示。

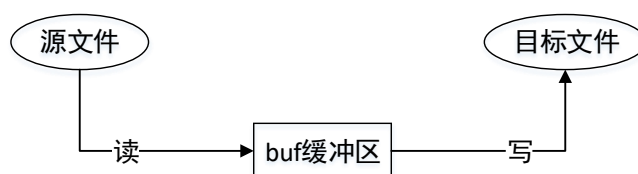


图 3.1 文件拷贝设计思路

循环进行上述读源文件和写目标文件的操作，直至读取到源文件末尾，即完成文件拷贝工作。为模拟 Linux 的 cp 命令，利用命令行参数，向 main 函数传入源文件路径和目标文件路径。main 函数使用如下 Linux API，分别实现文件的打开、读、写和关闭功能。

```
01. //头文件: #include <sys/types.h>
02. //      #include <sys/stat.h>
03. //      #include <fcntl.h>
04. //功能: 打开指定路径的文件
05. //参数: pathname文件名路径 flags打开模式
06. //返回值: 成功返回文件描述符fd 失败返回-1
07. int open(const char *pathname, int flags);
08.
09. //头文件: #include <unistd.h>
10. //功能: 读取文件内容
11. //参数: 文件描述符fd 存放读取内容的缓冲区buf 读取字节数count
12. //返回值: 成功返回实际读取的字节数 等于0表示未读入数据 失败返回-1
13. ssize_t read(int fd, void *buf, size_t count);
14.
15. //头文件: #include <unistd.h>
16. //功能: 向文件中写入数据
17. //参数: 要写入的文件描述符fd 要写入的数据存放缓冲区buf中 count表示要写入的字节个数
18. //返回值: 成功返回实际写入的字节数 等于0表示未写入任何数据 失败返回-1
19. ssize_t write(int fd, void *buf, size_t count);
20.
21. //头文件: #include <unistd.h>
22. //功能: 关闭一个被打开的文件
23. //参数: 待关闭文件的文件描述符fd
24. //返回值: 成功返回0 失败返回-1
25. int close(int fd);
```

图 3.2 文件操作 API

另外，为了保持源文件和目标文件在读写权限上的一致性，考虑将目标文件

读写权限设置为源文件权限，此时用到 API 如下。

```
01. //头文件: #include <sys/stat.h>
02. //      #include <unistd.h>
03. //功能: 获取文件信息并存放在buf所指的stat类型的结构体中
04. int stat(const char *file_name, struct stat *buf);
05.
06. //头文件: #include <sys/stat.h>
07. //功能: 改变目录或文件的读/写/执行权限为mode
08. int chmod(const char *pathname, mode_t mode);
```

图 3.3 文件权限操作 API

3.1.2 进程并发

该题目要求用图形界面分窗口显示三个并发进程的运行情况，考虑先分别实现三个图形界面窗口的功能，再利用 fork 语句实现进程并发，选择 Qt 作为图形界面的开发框架。

1、窗口一：实时显示当前时间

借助 Qt 的 QDateTime 类，调用静态函数 QDateTime::currentDateTime()，获取本地系统的时间和日期。

2、窗口二：显示/etc/fstab 文件内容

借助 Qt 的 QFile 类提供的读写文件的接口，调用如下函数成员，实现文件的打开、读取和关闭。

表 3.1 QFile 成员函数

功 能	函数原型	备 注
构造函数	QFile(const QString &name)	name 为要操作的文件路径
打开文件	bool QFile::open(OpenMode mode)	按指定模式打开文件
读文件	QByteArray QIODevice::readAll()	一次性读取全部剩余数据
文件大小	qint64 QIODevice::size() const	
关闭文件	void QIODevice::close()	关闭相应文件

3、窗口三：显示 1-1000 的累加求和过程

通过 1000 次的循环即可求得 1-1000 的累加结果。

上述 1-3 获得需要在图形界面显示的信息，下面介绍图形界面的设计。

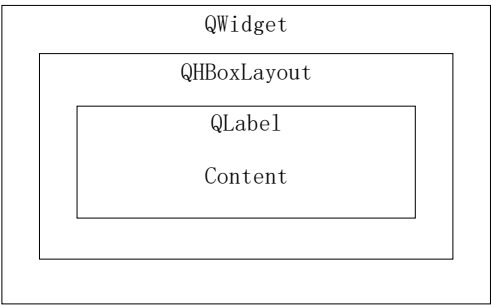


图 3.4 窗口设计

QWidget 为窗口实体，可以调用相应成员函数设置窗口的标题、初始位置、大小等属性。QHBoxLayout 为水平布局，用于放置 QLabel 类型的实例。真正显示的信息 Content 在 QLabel 类型的实例中，调用对应函数可设置文本格式。

另外，由于当前时间的实时显示是以秒为刷新单位，但文件内容的显示和 1-1000 累加的计算速度远小于 1 秒，若不加同步，则无法直观的显示出三个进程的并发。因此考虑使用定时器，每 0.5 秒进行一次 1-1000 的加法运算，每 1 秒显示部分文件内容，并刷新当前时间，以表明三个进程确实并发执行。

定时器的实现依靠 Qt 的 QTimer 类，QTimer 提供重复和单次触发信号的定时器，利用 Qt 的信号和槽机制，实现每过一个时间间隔，就执行相应操作，使用步骤如下，因此 1-3 所述操作在对应槽函数中实现即可。

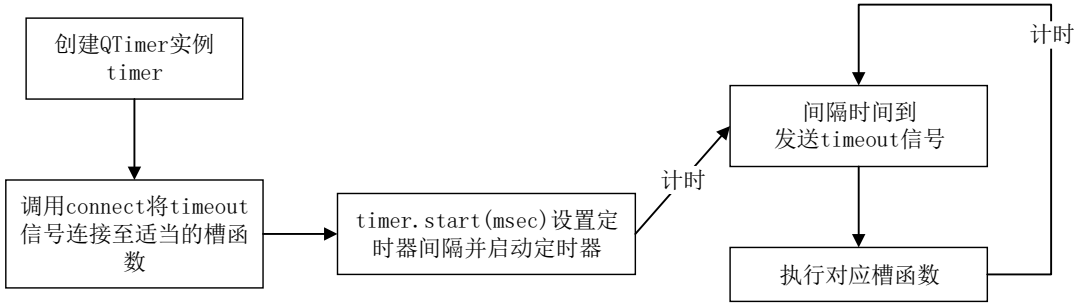


图 3.5 定时器设计

上述设计仅实现每个窗口的功能，为实现进程并发，需调用 Linux 的创建进程的函数 fork。实际上，Linux 下的所有进程都是由一个父进程调用 fork 函数创建的，fork 函数的功能如下。

表 3.2 fork 函数

头文件	#include <unistd.h>
函数原型	pid_t fork(void)
功能	创建一个新的子进程，该进程是父进程映像的一个副本
参数	无参数
返回值	若返回值大于 0，说明函数执行结束后返回父进程；等于 0，返回子进程；小于 0 说明 fork 函数执行过程中出错。

父进程调用 fork 函数创建两个子进程，此时三个进程分别显示三个窗口，即实现窗口并发执行的功能。


```

01.  main(){
02.      pid_t pid_1;
03.      pid_1 = fork();
04.      if(pid_1 > 0){
05.          pid_t pid_2;
06.          pid_2 = fork();
07.          if(pid_2 > 0){
08.              //显示窗口1
09.          }
10.          else if(pid_2 == 0){
11.              //显示窗口2
12.          }
13.      }
14.      else if(pid_1 == 0){
15.          //显示窗口3
16.      }
17.  }

```

图 3.6 并发进程创建

3.2 添加系统调用

3.2.1 编译内核

该题目要求采用编译内核的方法，添加一个新的系统调用，首先考虑在不改动源码的情况下直接编译并安装测试，验证其可用性，操作步骤如下。

1、在 <https://mirrors.edge.kernel.org/pub/linux/kernel> 上下载指定版本号的 Linux 源码，例如 linux-4.13.1.tar.xz，其中 4 为主版本号，有结构性变化时变更，13 为次版本号，新增功能时发生变化，1 为修订版本号。

2、将下载后的源码复制到 /usr/src 目录下，进入 /usr/src，解压源码。

```

→ ~ ls
Desktop    Downloads    linux-4.13.1.tar.xz  OS          Public    Videos
Documents  examples.desktop Music          Pictures    Templates
→ ~ sudo cp linux-4.13.1.tar.xz /usr/src
→ ~ cd /usr/src
→ src sudo tar xvjf linux-4.13.1.tar.xz
→ src ls
linux-4.13          linux-headers-4.10.0-40-generic
linux-4.13.1        linux-headers-4.13.0-36
linux-4.13.1.tar.xz linux-headers-4.13.0-36-generic
linux-headers-4.10.0-40

```

图 3.7 解压内核源码

3、利用 `uname -r` 命令查看当前内核版本号，复制其文件夹中的 .config 文件到 linux-4.13.1 目录下，进入 linux-4.13.1 目录，执行 `sudo make menuconfig`。

```

→ src uname -r
4.13.0-36-generic
→ src sudo cp linux-headers-4.13.0-36-generic/.config linux-4.13.1
→ src cd linux-4.13.1

```

图 3.8 复制配置文件

4、进入图形界面，选择 load->OK->Save->OK->EXIT->EXIT 后退出。

```
→ linux-4.13.1 sudo make menuconfig
HOSTCC scripts/kconfig/mconf.o
SHIPPED scripts/kconfig/zconf.tab.c
SHIPPED scripts/kconfig/zconf.lex.c
SHIPPED scripts/kconfig/zconf.hash.c
HOSTCC scripts/kconfig/zconf.tab.o
HOSTCC scripts/kconfig/lxdialog/checklist.o
HOSTCC scripts/kconfig/lxdialog/util.o
HOSTCC scripts/kconfig/lxdialog/inputbox.o
HOSTCC scripts/kconfig/lxdialog/textbox.o
HOSTCC scripts/kconfig/lxdialog/yesno.o
HOSTCC scripts/kconfig/lxdialog/menubox.o
HOSTLD scripts/kconfig/mconf
scripts/kconfig/mconf Kconfig
.config - Linux/x86 4.13.1 Kernel Configuration

Linux/x86 4.13.1 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to
exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]

[*] 64-bit kernel
    General setup --->
    [*] Enable loadable module support --->
    [*] Enable the block layer --->
        Processor type and features --->
        Power management and ACPI options --->
        Bus options (PCI etc.) --->
        Executable file formats / Emulations --->
    [*] Networking support --->
        Device Drivers --->
    ↵(+)
```

图 3.9 配置文件的装载

5、执行 `sudo make -j4` 命令进行内核编译，这里 4 为 CPU 核数*2，我是双核因此为-j4。

```
→ linux-4.13.1 sudo make -j4
HOSTCC scripts/kconfig/conf.o
HOSTLD scripts/kconfig/conf
scripts/kconfig/conf --silentoldconfig Kconfig
CHK include/config/kernel.release
HOSTCC scripts/basic/bin2c
SYSTBL arch/x86/entry/syscalls/../../include/generated/asm/syscalls_32.h
SYSHDR arch/x86/entry/syscalls/../../include/generated/asm/unistd_32_ia32.h
SYSHDR arch/x86/entry/syscalls/../../include/generated/asm/unistd_64_x32.h
CHK include/generated/uapi/linux/version.h
UPD include/generated/uapi/linux/version.h
SYSTBL arch/x86/entry/syscalls/../../include/generated/asm/syscalls_64.h
UPD include/config/kernel.release
HYPERCALLS arch/x86/entry/syscalls/../../include/generated/asm/xen-hypercalls.
h
SYSHDR arch/x86/entry/syscalls/../../include/generated/uapi/asm/unistd_32.h
SYSHDR arch/x86/entry/syscalls/../../include/generated/uapi/asm/unistd_64.h
SYSHDR arch/x86/entry/syscalls/../../include/generated/uapi/asm/unistd_x32.h
```

图 3.10 内核源码的编译

此步骤需要将 linux-4.13.1 目录下的所有 C 文件编译成 O 文件，耗时较久。

6、编译完成后，安装内核，依次执行 `sudo make modules install` 和 `sudo make install` 命令，完成模块和内核的安装。

```

IHEX2FW firmware/keyspan_pda/xircom_pgs.fw
IHEX2FW firmware/keyspan_pda/keyspan_pda.fw
IHEX    firmware/cpia2/stv0672_vp4.bin
IHEX    firmware/yam/1200.bin
IHEX    firmware/yam/9600.bin
→ linux-4.13.1 sudo make modules install
[sudo] harperchen 的密码:
CHK     include/config/kernel.release
CHK     include/generated/uapi/linux/version.h
CHK     include/generated/utsrelease.h
CHK     include/generated/bounds.h
CHK     include/generated/timeconst.h
CHK     include/generated/asm-offsets.h
CALL    scripts/checksyscalls.sh
CHK     scripts/mod/devicetable-offsets.h
CHK     include/generated/compile.h

```

图 3.11 模块的安装

```

Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
→ linux-4.13.1 sudo make install
[sudo] harperchen 的密码:
sh ./arch/x86/boot/install.sh 4.13.1 arch/x86/boot/bzImage \
    System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.13.1 /boot/
z-4.13.1
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.13.1 /boot/v
-4.13.1
update-initramfs: Generating /boot/initrd.img-4.13.1

```

图 3.12 系统的安装

7、重启系统，在 grub 页面，选择进入新的内核，此时利用 `uname -r` 命令，查看内核版本，即为新安装的版本。

```

GNU GRUB  version 2.02~beta2-36ubuntu3.17

*Ubuntu, Linux 4.13.1
Ubuntu, with Linux 4.13.1 (upstart)
Ubuntu, with Linux 4.13.1 (recovery mode)
Ubuntu, Linux 4.13.1.old
Ubuntu, with Linux 4.13.1.old (upstart)
Ubuntu, with Linux 4.13.1.old (recovery mode)
Ubuntu, Linux 4.13.0-36-generic
Ubuntu, with Linux 4.13.0-36-generic (upstart)
Ubuntu, with Linux 4.13.0-36-generic (recovery mode)
Ubuntu, Linux 4.13.0
Ubuntu, with Linux 4.13.0 (upstart)
Ubuntu, with Linux 4.13.0 (recovery mode)
Ubuntu, Linux 4.10.0-40-generic
Ubuntu, with Linux 4.10.0-40-generic (upstart)
Ubuntu, with Linux 4.10.0-40-generic (recovery mode)

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the commands
before booting or 'c' for a command-line. ESC to return previous
menu.

```

图 3.13 Grub 页面

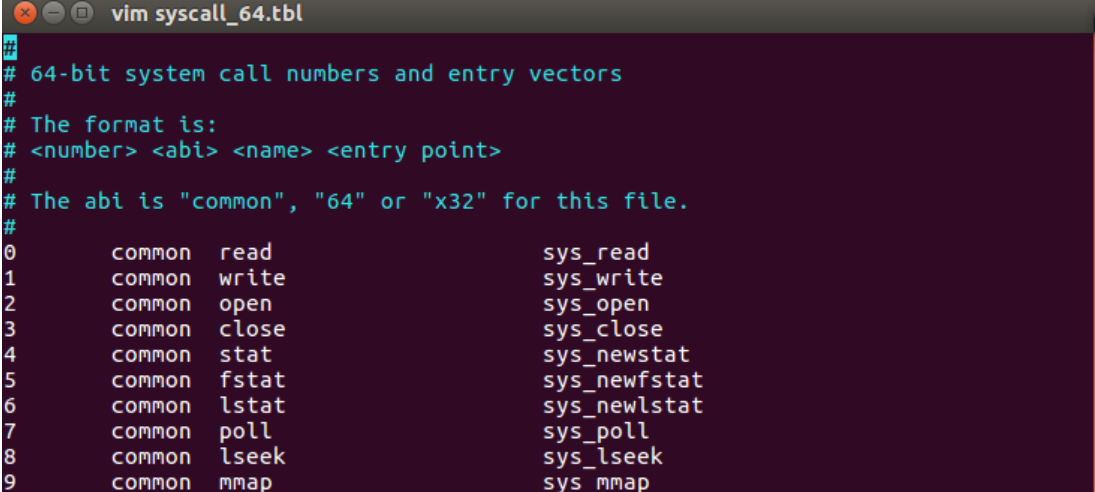
3.2.2 添加系统调用

系统调用，指的是 Linux 内核中设置的一组用于实现各种系统功能的子程序。用户可以通过函数接口调用这些系统调用命令，其调用过程类似于普通函数的调用，区别在于系统调用由操作系统核心提供，运行于核心态，而普通函数调用由函数库或用户自己提供，运行于用户态。

用户在调用系统调用时会通过寄存器向内核传递一个系统调用号，然后系统调用处理程序通过此号从系统调用表中找到相应的内核函数（系统调用服务例程）入口并执行，最后返回到用户。下面依次介绍系统调用号、系统调用表、系统调用服务例程和系统调用处理程序。

1、系统调用号

Linux 系统中有几百个系统调用，为了唯一的标识每一个系统调用，linux 为每一个系统调用分配了一个唯一的编号，此编号就是系统调用号，另外，该调用号同时作为系统调用表的下标，用于查找对应内核函数的入口地址。对于 v4.x 的内核，系统调用号的定义存放在 `/arch/x86/entry/syscalls/syscall_64.tbl` 文件中(版本不同为文件位置不同)。



```
vim syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0      COMMON  read      sys_read
1      COMMON  write     sys_write
2      COMMON  open      sys_open
3      COMMON  close     sys_close
4      COMMON  stat      sys_newstat
5      COMMON  fstat     sys_newfstat
6      COMMON  lstat     sys_newlstat
7      COMMON  poll      sys_poll
8      COMMON  lseek     sys_lseek
9      COMMON  mmap      sys_mmap
```

图 3.14 系统调用号所在文件

若需要添加新的系统调用，则首先分配一个未被使用的系统调用号，然后在该文件中按照固定格式添加新系统调用的调用号、名称、函数名称等信息即可。

2、系统调用表

为了把系统调用号与相应的服务例程（子程序）关联起来，内核利用了一个系统调用表，这个系统调用表放在 `sys_call_table` 数组中，他是一个函数指针数组，每一组指针都指向其系统调用的封装例程，有 `NR_syscalls` 个表项，第 `n` 个表项包含系统调用号为 `n` 的服务例程的地址。`NR_syscalls` 宏只是对可实现的系统调用最大个数的静态限制，并不表示实际以实现的系统调用个数。这样我们就可以利用系统调用号作为下标，找到其系统调用例程。

3、系统调用服务例程和系统调用处理程序

每一个系统调用 `bar()` 在内核都有一个对应的内核函数 `sys_bar()`，这个内核函数就是系统调用 `bar()` 的实现，也就是说在用户态调用 `bar()`，最终会有内核函数 `sys_bar()` 为用户服务，这里的 `sys_bar()` 就是系统调用的服务例程。

因为内核代码驻留在受保护的地址空间上，不允许用户进程在内核地址空间上进行读写，因此，用户空间无法直接执行内核代码。所以，应用程序应该以某种方式通知系统，告诉内核自己需要执行一个系统调用，而这种机制是通过软中断实现的，通过引发一个异常促使系统切换到内核态去执行异常处理程序。此时的异常处理程序就是所谓的系统调用处理程序。

总结来说，系统调用发生时，通过软中断，使用寄存器中系统调用号执行系统调用处理程序，该程序检查系统调用号，再查找系统调用表 `sys_call_table`，找到服务例程的函数指针，然后跳转到服务例程中执行，最后返回

综上，若要添加新的系统调用，不仅需要分配并登记其系统调用号，还需要实现对应的服务例程，对于 `v4.x` 的内核，操作步骤如下。

- (1) 在 `kernel/sys.c` 中添加系统调用函数的定义，函数名以 `sys_` 开头；
- (2) 在 `include/linux/syscalls.h` 中声明系统调用函数原型。

3.2.3 系统调用的实现和测试

要求添加实现文件拷贝功能的系统调用，可重用 3.1.1 中实现文件拷贝的代码，但使用的文件操作函数，由于内核代码和用户代码的区别，呈现出如下差异。

表 3.3 内核态文件操作 API

用户空间	内核
<code>open</code>	<code>sys_open</code>
<code>read</code>	<code>sys_read</code>
<code>write</code>	<code>sys_write</code>
<code>close</code>	<code>sys_close</code>

另外，系统调用本来是提供给用户空间的程序访问的，所以对传递给它的参数，它默认会认为来自用户空间，在 `write()` 函数中，为了保护内核空间，一般会用 `get_fs()` 得到的值来和 `USER_DS` 进行比较，从而防止用户空间程序“蓄意”破坏内核空间，因此，在进行文件操作时，由于缓冲区为内核空间，需要暂时将访问限制值设置为内核的内存访问范围，然后再修改为原来的值，使用到的函数如下。

- (1) `get_ds()`: 获取 `kernel` 的内存访问地址范围；
- (2) `get_fs()`: 获取当前的地址访问限制值；
- (3) `set_fs()`: 设置当前的地址访问限制值。

系统调用添加成功后，利用 `syscall` 函数测试系统调用，函数原型如下。

```
01. //头文件: #include <unistd.h>
02. //      #include <sys/syscall.h>
03. //参数: number为系统调用号 其后传递相应参数
04. //例如syscall(number, src_file_path, dest_file_path);
05. int syscall(int number, ...);
```

图 3.15 `syscall` 函数原型

3.3 添加设备驱动程序

Linux 设备驱动属于内核的一部分，Linux 内核的一个模块可以以两种方式被编译和加载：

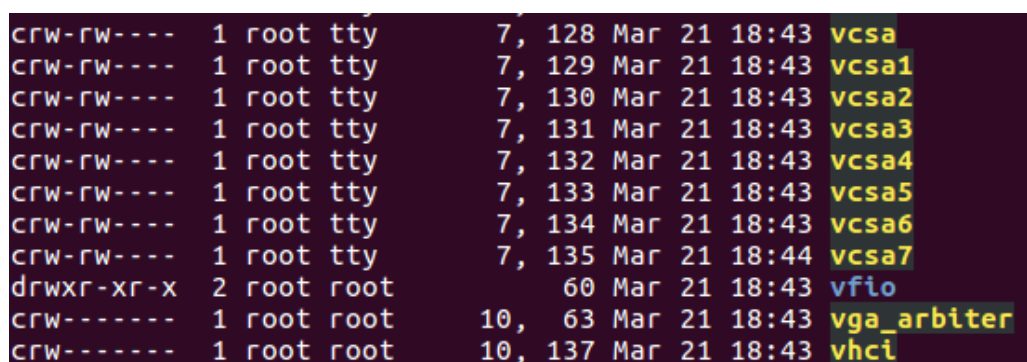
(1) 编译内核：直接编译进 Linux 内核，随同 Linux 启动时加载；

(2) 模块加载：编译成一个可加载和删除的模块，以模块的方式，使用 `insmod` 命令加载，`rmmmod` 命令删除。这种方式控制了内核的大小，而模块一旦被插入内核，它就和内核其他部分一样。

字符设备驱动是指设备发送和接受数据都是以字符的形式进行，没有缓冲且只能顺序存储。本实验考虑实现字符设备的打开、关闭、读、写和光标移动功能，即实现 `open`、`close`、`read`、`write` 和 `lseek` 函数。

3.3.1 主设备号和次设备号

当应用程序访问硬件设备时，实际上是对设备文件进行操作。设备文件位于 `/dev` 目录下，利用 `ls -l` 命令可查看有关信息，如下图。



```
crw-rw---- 1 root tty      7, 128 Mar 21 18:43 vcsa
crw-rw---- 1 root tty      7, 129 Mar 21 18:43 vcsa1
crw-rw---- 1 root tty      7, 130 Mar 21 18:43 vcsa2
crw-rw---- 1 root tty      7, 131 Mar 21 18:43 vcsa3
crw-rw---- 1 root tty      7, 132 Mar 21 18:43 vcsa4
crw-rw---- 1 root tty      7, 133 Mar 21 18:43 vcsa5
crw-rw---- 1 root tty      7, 134 Mar 21 18:43 vcsa6
crw-rw---- 1 root tty      7, 135 Mar 21 18:44 vcsa7
drwxr-xr-x 2 root root      60 Mar 21 18:43 vfio
crw----- 1 root root    10,  63 Mar 21 18:43 vga_arbiter
crw----- 1 root root    10, 137 Mar 21 18:43 vhci
```

图 3.16 `/proc/dev` 目录信息

其中 `c` 表示字符设备，对于普通文件来说，`ls -l` 会列出文件的长度，而对于设备文件来说，上面的 7、10 等代表的是对应设备的主设备号，而后面的 128、129、130 等则是对应设备的次设备号。一般情况下，主设备号标识设备对应的驱动程序，1 个主设备号对应一个驱动程序，也存在多个驱动程序共享主设备号的情况。而次设备号由内核使用，用于确定 `/dev` 下的设备文件对应的具体设备。比如，虚拟控制台和串口终端有驱动程序 4 管理，主设备号为 4，而不同的终端

分别有不同的次设备号。

因此，在构建一个字符设备之前，驱动程序首先要获得一个或者多个设备编号，注册字符设备的函数如下，该函数定义在<linux/fs.h>中。

```
1. int register_chrdev (unsigned int major, const char *name, struct file_operations*fops);
```

参数：major 是为该字符设备分配的主设备号，范围为 0-255，若为 0 则采用系统动态分配的主设备号。name 为字符设备名，fops 为 struct file_operations 类型的结构体指针，该结构体是一个函数指针的集合，里面存放了对该设备的操作的实现方法，如 open、close 等，结构体内容如下，此处只列出与本课设有关的函数。该函数若正确则返回 0，失败返回错误号。

```
1. struct file_operations {
2.     struct module *owner;
3.     loff_t (*llseek) (struct file *, loff_t, int);
4.     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5.     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6.     ...
7.     int (*open) (struct inode *, struct file *);
8.     int (*release) (struct inode *, struct file *);
9. };
```

实际上，对设备可以进行的大部分系统调用都由此结构提供。在编写模块时，若要实现打开功能，则必须编写类型为上述第 7 行所示的函数，在函数体中编写实现具体的功能的代码，最后将结构体的函数指针和本文件的函数对应即可。因此注册设备函数不仅为驱动程序分配了一个设备号，而且通过 file_operations 结构体把驱动程序的基本入口点指针存放在内核的字符设备地址表中，在用户进程对该设备执行系统调用时提供入口地址。

对应的字符设备注销函数如下，其 major 和 name 两个参数必须和注册函数一致。同样，该函数若正确则返回 0，失败返回错误号。

```
1. int unregister_chrdev(unsigned int major, const char *name);
```

3.3.2 文件操作和进程并发

1、文件操作

由于在 kernel 中操作文件没有标准库可用，需要利用 kernel 的一些函数实现对文件的操作，这些函数在 linux/fs.h 和 asm/uaccess.h 头文件中声明，列举如下。

表 3.4 内核态文件操作 API

功能	原型
打开文件	<code>struct file* filp_open(const char* filename, int open_mode, int mode);</code>
读文件	<code>ssize_t vfs_read(struct file* filp, char __user* buffer, size_t len, loff_t* pos);</code>
写文件	<code>ssize_t vfs_write(struct file* filp, const char __user* buffer, size_t len, loff_t* pos);</code>
关闭文件	<code>int filp_close(struct file* filp, fl_owner_t id);</code>
移动位置	<code>loff_t vfs_llseek(struct file *file, loff_t offset, int whence);</code>

`filp_open` 函数中, `open_mode` 为文件的打开方式, 其取值与标准库中的 `open` 相应参数类似, 可以取 `O_CREAT/O_RDWR/O_RDONLY` 等值, 在创建文件时, `mode` 参数用于设置创建文件的读写权限, 其它情况可以忽略。

文件读写函数中, 第二个参数 `buffer` 前面有 `__user` 修饰符, 要求 `buffer` 指针应该指向用户空间的内存, 如果对该参数传递 `kernel` 空间的指针, 这两个函数都会返回失败-`EFAULT`。要使这两个读写函数使用 `kernel` 空间的 `buffer` 指针也能正常工作, 需要使用 `set_fs()` 函数改变 `kernel` 对内存地址检查的处理方式。

2、进程并发

在驱动程序中, 当多个线程同时访问相同的资源时, 可能会引发"竞态", 因此我们必须对共享资源进行并发控制。Linux 内核中解决并发控制的最常用方法是自旋锁与信号量。

与信号量有关 API 主要有:

- (1) 定义信号量 `struct semaphore sem;`
- (2) 初始化信号量 `void sema_init(struct semaphore *sem, int val);`
- (3) 对信号量进行 P 操作 `void down(struct semaphore * sem);`
- (4) 对信号量进行 V 操作 `void up(struct semaphore * sem);`

与自旋锁相关的 API 主要有:

- (1) 定义自旋锁 `spinlock_t spin;`
- (2) 初始化自旋锁 `DEFINE_SPINLOCK(spin);`
- (3) 获得自旋锁 `spin_lock(lock);`
- (4) 释放自旋锁 `spin_unlock(lock);`

3.3.3 模块代码流程

模块代码编写主要框架如下:

1. `MODULE_LICENSE("GPL");` // 声明模块的许可证
2. `#define MAJOR_NUM 244`
- 3.


```

4. static int chr_open(struct inode *inode, struct file *filp){
5.     //设备打开对应操作
6.     return 0;
7. }
8. static int chr_release(struct inode *inode, struct file *filp){
9.     //设备关闭对应操作
10.    return 0;
11.}
12.static ssize_t chr_read(struct file *filp, char *buf, size_t count,
loff_t *f_pos){
13.    //读文件操作
14.    return count;
15.}
16.static ssize_t chr_write(struct file *filp, const char *buf, size_t
count, loff_t *f_pos){
17.    //写文件操作
18.    return count;
19.}
20.static loff_t chr_llseek(struct file *filp, loff_t off, int whence){
21.    //移动光标操作
22.}
23.struct file_operations chr_fops = {
24.    .open    =  chr_open,
25.    .release =  chr_release,
26.    .read    =  chr_read,
27.    .write   =  chr__write,
28.    .llseek  =  chr_llseek,
29.};
30.static int chr_init(void)
31.{
32.    int ret;
33.    ret = register_chrdev(MAJOR_NUM, "mychr", &chr_fops);
34.    if (ret){
35.        printk("Fail to register char device!\n");
36.        return ret;

```

```

37.     }
38.     else{
39.         //设备注册成功后操作
40.     }
41.     return 0;
42.}
43.
44.static void chr_exit(void) {
45.    unregister_chrdev(MAJOR_NUM, "mychr");
46.}
47.
48.module_init(chr_init);
49.module_exit(chr_exit);

```

本实验考虑将所有对设备文件的操作映射到/tmp 目录下的临时文件 mychr，相当于开辟一部分磁盘空间作为设备。对设备文件的打开/关闭或读写操作均为对临时文件的操作，需要注意的是，对于文件读写位置，两个文件始终要保持一致。实际上，设备文件相当于傀儡文件，本次实验实现的驱动将内核态的文件操作函数进行封装，并加以读写和打开控制。

为增加程序的健壮性，在模块初始化函数中创建/tmp/mychr 函数时不关闭，直接使文件处于打开状态，而打开文件的函数 chr_open 只实现对执行该函数进程数的控制，防止临时文件在内核态被多次打开。类似的，在模块卸载时关闭临时文件，文件关闭 chr_release 函数只将全局变量 count 进行减一操作，防止内核态多次关闭文件。

该框架包括如下几个部分。

1、模块初始化函数 chr_init

该函数在执行 insmod 命令时运行，功能如下：

- (1) 注册字符设备
- (2) 创建临时文件/tmp/mychr 并打开
- (3) 初始化信号量 sem 的值为 1，实现读写操作的互斥

2、模块卸载函数 chr_exit

该函数在执行 rmmod 命令时运行，功能如下：

- (1) 关闭临时文件
- (2) 注销字符设备

3、设备打开函数 chr_open

应用程序对设备文件进行 open() 系统调用时，将调用驱动程序的 chr_open() 函数，其中参数 inode 为设备文件的 inode(索引结点)结构的指针，参数 file

是指向这一设备文件结构的指针。函数根据执行情况返回状态码，该函数功能如下：

- (1) 利用自旋锁保护全局变量 `count`，`count` 变量控制使用该设备的进程数，防止多个进程同时打开文件

- (2) 上锁后对 `count` 执行加一操作

- (3) 文件光标置 0，指向文件头

- (4) 释放自旋锁

4、设备关闭函数 `chr_release`

当最后一个打开设备的用户进程执行 `close()` 系统调用时，内核将调用驱动程序程序的 `chr_release()` 函数，参数和返回值含义同 `open` 函数，功能如下：

- (1) 对 `count` 执行减一操作，表示打开设备的进程减少

- (2) 文件光标置 0，指向文件头

5、`chr_read` 函数

当对设备文件进行 `read()` 系统调用时，将调用驱动程序 `chr_read()` 函数，该函数传入参数均为设备文件的相关信息，需要同步操作 `/tmp/mychr` 临时文件。函数返回非负值表示成功读取的字节数，函数功能如下：

- (1) 对信号量 `sem` 进行 P 操作表示占用设备，防止读写同时进行

- (2) 调用 `vfs_read` 函数，从 `/tmp/mychr` 文件读取数据

- (3) 移动设备文件的 `f_pos`，使其和临时文件一致

- (4) 对信号量 `sem` 进行 V 操作表示释放设备

- (5) 返回成功写入的字节数

6、`chr_write` 函数

当对设备文件进行 `write()` 系统调用时，将调用驱动程序的 `chr_write()` 函数，功能如下：

- (1) 对信号量 `sem` 的值进行 P 操作

- (2) 调用 `vfs_write` 函数，向 `/tmp/mychr` 文件写入数据

- (3) 移动设备文件的 `f_pos`，使其和临时文件一致

- (4) 对信号量 `sem` 进行 V 操作表示释放设备

- (5) 返回成功写入的字节数

7、`chr_llseek` 函数

当对设备文件进行 `lseek()` 系统调用时，将调用驱动程序的 `chr_llseek()` 函数，该函数调用 `vfs_llseek` 函数修改 `/tmp/mychr` 文件的读写位置，并同步更新设备文件的读写位置，返回新的读写位置。

3.3.4 测试程序

驱动设备程序完成后，加载模块并测试模块功能，实际上是测试对设备文件的打开/关闭等操作是否能正常运行。

首先需要在/dev 目录下建立设备文件，其次在测试程序中分别对设备文件调用 open/write/lseek/read/close 函数，并测试多进程打开设备文件的结果，查看系统日志确认功能是否正确。

3.4 系统监控

3.4.1 /proc 文件系统

/proc 是一个伪文件系统，是进程文件系统和内核文件系统组成的复合体，被用作内核数据结构的接口，该文件系统是一个实时的、常驻内存的文件系统，跟踪进程在机器上的运行情况和系统的状态。大部分的/proc 文件系统信息被实时更新来与当前操作系统的状态一致。/proc 文件系统的内容能被任何有相应权限的人读取。但/proc 文件系统的特定部分只能被这个进程的拥有者和 root 用户读取。列举/proc 文件夹内容如下，可以看到有很多以数字命名的文件夹，也有类似 cpuinfo 等的文件。

下面依次介绍重要文件/文件夹的信息。

```
→ /proc ls
1      187    215    229    244    2566    322    45    99      locks
10     188    216    2294   245    257     350    46    acpi    mdstat
100    19     2163   2296   246    2575    35418  47    asound  meminfo
101    195    217    230    2460   258     35423  48    buddyinfo  misc
102    1953   2171   2306   2468   259     35638  49    bus     modules
1022   1954    218    231    2469   2596    35639  50    cgroups mounts
1025   1955   2183   2317   247    26      35692  53    cmdline mpt
1047   198     219    232    2473   2600    360     54    consoles mtrr
108    199     22     233    2474   2629    36158  549   cpuinfo net
11     2       220    234    2475   2665    36208  55    crypto  pagetypeinfo
1136   20      2207   2347   248     27     36263  6     devices partitions
12     200     221    2350   249    2711    36277  7     diskstats sched_debug
13     201     2219   2351   2492   2719    36278  8     dma     schedstat
```

图 3.17 /proc 目录结构

A. /proc/pid

实际上，数字目录代表进程，其数字为进程号，这些进程在我们对/proc 文件系统作快照时正运行在机器上。进入 2596 文件夹，有下列文件，下面介绍和系统监控相关文件信息。

```

→ /proc cd 2596
→ 2596 ls
attr          cwd          map_files    oom_adj      schedstat    task
autogroup     environ     maps         oom_score    sessionid    timers
auxv          exe         mem          oom_score_adj setgroups    timerslack_ns
cgroup        fd          mountinfo    pagemap      smaps        uid_map
clear_refs    fdinfo      mounts       patch_state   stack        wchan
cmdline       gid_map     mountstats   personality   stat
comm          io          net          projid_map    statm
coredump_filter limits      ns           root          status
cpuset        loginuid    numa_maps    sched         syscall

```

图 3.18 /proc/pid 目录结构

a) status

该文件包含进程的状态信息，可以从中获得进程名/进程状态/进程所有者 ID/使用内存大小等信息。

```

→ 2596 sudo cat status
[sudo] harperchen 的密码:
Name:   gvfsd-trash
Umask:  0002
State:  S (sleeping)
Tgid:   2596
Ngid:   0
Pid:    2596
PPid:   2088
TracerPid: 0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 64
Groups: 4 24 27 30 46 113 128 1000

```

图 3.19 /proc/pid/status 文件

b) stat

该文件包含了该进程所有 CPU 活跃的信息，文件中的所有值都是从系统启动开始累计到当前时刻，文件内容如下。

```

→ 2596 sudo cat stat
2596 (gvfsd-trash) S 2088 2171 2171 0 -1 4194304 446 0 1 0 4 8 0 0 20 0 3 0 5406
0 361074688 1180 18446744073709551615 4194304 4236380 140726503007760 0 0 0 0 40
96 2048 0 0 0 17 2 0 0 10 0 0 6335904 6338040 36216832 140726503016338 140726503
016405 140726503016405 140726503018462 0

```

图 3.20 /proc/pid/stat 文件

文件包含 pid 进程号、comm 应用程序或命令名称和进程优先级等信息，其中大部分信息和 status 文件重复。

B. /proc/meminfo

该文件存放操作系统内存的使用状态。

MemTotal 是可使用内存的总量，是物理内存减去一些保留内存和内核二进制代码占用的内存，MemAvailable 为可使用的内存大小，SwapTotal 为交换总量，SwapFree 为空闲 swap 总量。

```
→ /proc cat meminfo
MemTotal:      2018084 kB
MemFree:       77060 kB
MemAvailable:  1050996 kB
Buffers:       69008 kB
Cached:        956544 kB
SwapCached:    0 kB
Active:        1046176 kB
Inactive:      508108 kB
Active(anon):  529832 kB
Inactive(anon): 10504 kB
Active(file):  516344 kB
Inactive(file): 497604 kB
Unevictable:   48 kB
Mlocked:       48 kB
SwapTotal:     0 kB
SwapFree:      0 kB
```

图 3.21 /proc/meminfo 文件

C. /proc/cpuinfo

该文件中存放了 cpu 的相关信息，包括型号、缓存大小和性能等。

```
processor      : 1
vendor_id     : GenuineIntel
cpu family    : 6
model         : 61
model name    : Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
stepping      : 4
microcode     : 0x1f
cpu MHz       : 2401.000
cache size    : 4096 KB
physical id   : 0
siblings      : 4
core id       : 1
cpu cores     : 4
apicid        : 1
initial apicid : 1
fpu           : yes
fpu_exception : yes
cpuid level   : 20
wp            : yes
```

图 3.22 /proc/cpuinfo 文件

processor 为 cpu 的一个物理标识,可以根据 processor 出现的次数统计 cpu 的逻辑个数(包括多核、超线程), model name 为 CPU 型号信息。

D. /proc/stat

该文件包含了所有 CPU 活动的信息，该文件中的所有值都是从系统启动开始累计到当前时刻。下面通过实例来说明数据该文件中各字段的含义。

```
→ /proc cat stat
cpu 106658 462 78960 6981275 33286 0 2692 0 0 0
cpu0 27452 10 19713 1747051 7698 0 468 0 0 0
cpu1 20336 14 16641 1756871 5486 0 537 0 0 0
cpu2 21042 45 15247 1760536 5115 0 394 0 0 0
cpu3 37828 392 27358 1716816 14985 0 1292 0 0 0
```

图 3.23 /proc/stat 文件

表 3.5 /proc/stat 文件结构

参数	解析（单位：jiffies）
user(106658)	处于用户态的运行时间
nice(462)	nice 值为负的进程所占用的 CPU 时间
system(78960)	处于核心态的运行时间
idle(6981275)	除 IO 等待时间以外的其它等待时间
iowait(33286)	IO 等待时间
irq(0)	硬中断时间
softirq(2692)	软中断时间
steal(0)	
guest(0)	
guest_nice(0)	

因此，总的 cpu 时间 $\text{totalCpuTime} = \text{user} + \text{nice} + \text{system} + \text{idle} + \text{iowait} + \text{irq} + \text{softirq} + \text{steal} + \text{guest} + \text{guest_nice}$ 。

若要计算 CPU 利用率，需要采样两个足够短的时间间隔的 Cpu 快照，计算步骤如下。

- (1) 计算两次快照的总的 Cpu 时间 totalCpuTime 得到 cpuTotal_1 和 cpuTotal_2 ;
- (2) $\text{total} = \text{cpuTotal_2} - \text{cpuTotal_1}$ 得到这个时间间隔内的 cpu 时间片;
- (3) 获取两次快照的 idle 等待时间得到 idle_1 和 idle_2 ;
- (4) $\text{idle} = \text{idle_2} - \text{idle_1}$ 得到这个时间间隔内的等待时间;
- (4) 计算 cpu 使用率 $\text{pcpu} = 100 * (\text{total} - \text{idle}) / \text{total}$ 。

E. /proc/net/dev

该文件记录网络传输过程中接收和发送数据的信息。

```

→ /proc cat net/dev
Inter-|   Receive
face |bytes   packets errs drop fifo frame compressed multicast|
ens33: 291334    1486    0    0    0    0          0          0
lo:    50703     282    0    0    0    0          0          0

| Transmit
|bytes   packets errs drop fifo colls carrier compressed
112604    1085    0    0    0    0          0          0
50703     282    0    0    0    0          0          0

```

图 3.24 /proc/net/dev 文件

F. /proc/modules

该文件保存了当前系统中被加载模块的相关信息，截图如下。每一个模块占用一行，从左至右分别为模块的名字、内存大小（单位 bytes）、被 load 的次数、依赖的第三方 module、模块的状态（Live/Loading/Unloading）和模块当

前的内核内存偏移位置。

```
→ ~ cat /proc/modules
btrfs 1101824 0 - Live 0x0000000000000000
xor 24576 1 btrfs, Live 0x0000000000000000
raid6_pq 118784 1 btrfs, Live 0x0000000000000000
ufs 73728 0 - Live 0x0000000000000000
qnx4 16384 0 - Live 0x0000000000000000
hfsplus 106496 0 - Live 0x0000000000000000
```

图 3.25 /proc/modules 文件

3.4.2 Qt 类

为展示图形界面，本程序考虑用 Qt Design 设计大致框架后，再编写代码向其中添加表格、进度条、标签等。

Qt Design 设计界面如下，利用 QTabWidget 类，分出五个页面，分别显示系统的五个方面的信息。

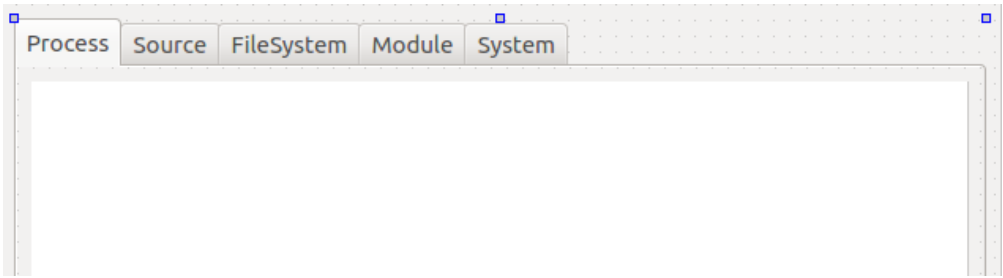


图 3.26 系统监控 ui 设计

而在 Process、FileSystem、Module 界面，设计利用 QTableWidgetItem 类，以表格形式展示信息，而在 Source 页面，考虑利用进度条显示内存比例信息，截图如下。

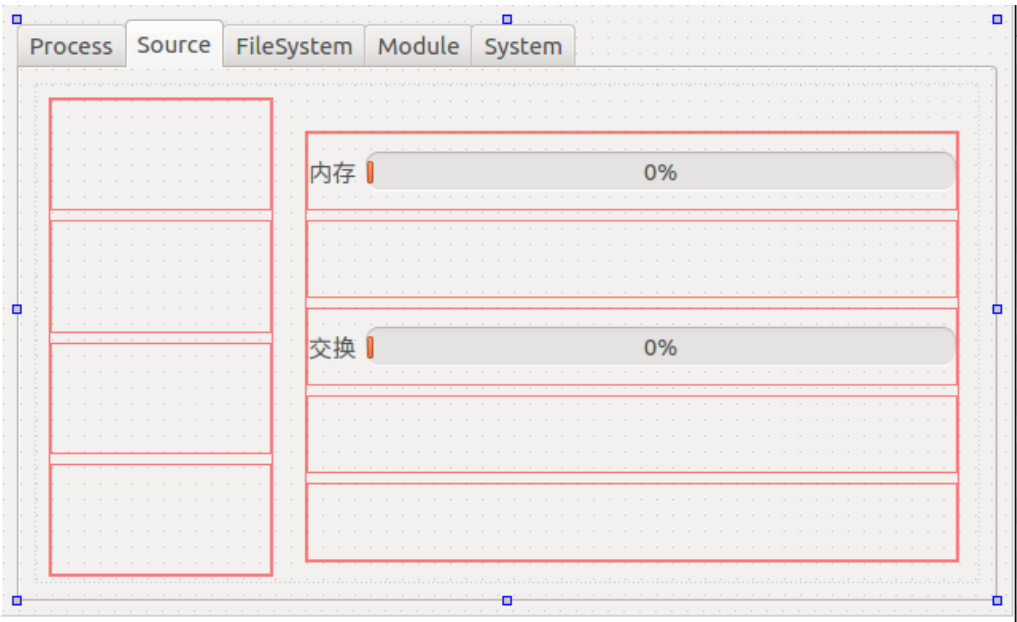


图 3.27 Source 页面设计

具体每个页面的内容，则通过读取相应文件获取有关信息，利用 Qt 提供 API 显示在表格或标签中。另外，为实现动态监控，设置定时器，每隔一段时间重新执行函数刷新页面。

3.5 模拟文件系统设计

该题目要求设计和实现一个简单的文件系统。由于 VFS 的存在，文件系统的设计变得简单许多，所以本题采用 VFS 来写，实现了一个运行在内核里面的简单文件系统，而不是模拟实现了文件系统。

3.5.1 总体设计

本文件系统的磁盘结构参考 minix 的文件系统实现。但是自举块（或称引导块）中没有数据。切不采用二级或者多级索引，其结构如下图。

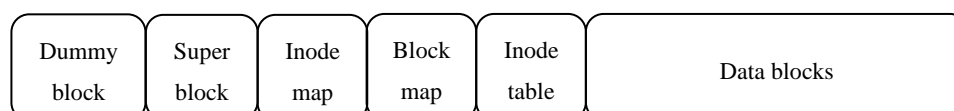


图 3.28 磁盘结构

其中每个块的大小定义为 4096bytes，每个 Inode 含有 10 个块，所以单个文件最大为 40KB，支持的最小磁盘大小为 24K。下面详细阐述文件系统中所需要的三个基本数据结构。

1. 超级块结构：

```
2. struct HUST_fs_super_block {
3.     uint64_t version;
4.     uint64_t magic;
5.     uint64_t block_size;
6.     uint64_t inodes_count;
7.     uint64_t free_blocks;
8.     uint64_t blocks_count;
9.     uint64_t bmap_block;
10.    uint64_t imap_block;
11.    uint64_t inode_table_block;
12.    uint64_t data_block_number;
13.    char padding[4016];
14.};
```

超级块中的 padding 数组，是为了使超级块的大小为 4096bytes，以简化后期的工作；“Magic”为 1314522；inode_count 记录文件系统所支持的 inode

个数，这个值在格式化时就已经计算并写入超级块了。Bmap_block 记录 bmap 开始的数据块索引，同理，imap_block、inode_table_block 和 data_block_number 记录索引是为了简化文件块的定位操作。

2. HUST_inode 结构:

```
15. struct HUST_inode {
16.     mode_t mode; //sizeof(mode_t) is 4
17.     uint64_t inode_no;
18.     uint64_t blocks;
19.     uint64_t block[HUST_N_BLOCKS];
20.     union {
21.         uint64_t file_size;
22.         uint64_t dir_children_count;
23.     };
24.     int32_t i_uid;
25.     int32_t i_gid;
26.     int32_t i_nlink;
27.     int64_t i_atime;
28.     int64_t i_mtime;
29.     int64_t i_ctime;
30.     char padding[112];
31.};
```

HUST_inode 对应着磁盘上的 inode 结构，在后文会描述它是如何转换为 VFS 中的 inode。在上述结构体中，mode 代表该 inode 是文件还是目录，blocks 代表该 inode 的大小（所占块的数目），i_uid 和 i_gid 用于后面的多用户管理。Padding 数组是为了让 HUST_inode 结构体能够被 4096 整除；宏 HUST_N_BLOCK 被定义为 10，意味着每个文件（目录）最大的大小为 10 个块；block 数组存储着每个块的索引，用于定位文件。

3. 文件系统的目录记录:

```
1. struct HUST_dir_record {
2.     char filename[HUST_FILENAME_MAX_LEN];
3.     uint64_t inode_no;
4.};
```

文件记录是为了储存目录项，其中 HUST_FILENAME_MAX_LEN 定义为 256，也就是说，文件名最大长度 256。

编写文件系统除了设计文件系统的磁盘结构，定义文件系统支持的操作也是十分重要的，这个直接影响了文件系统的功能。本文件系统支持基本的文件的增

删改查，多用户等功能，但是不支持文件的移动，软硬链接等操作。

3.5.2 文件系统的格式化和 mkfs 的实现

要使用这个文件系统，首先必须创建一个符合磁盘布局的映像文件，所以需要实现一个格式化程序，这个程序按照惯例叫做 mkfs。本节详细描述 mkfs 的实现。

mkfs 的作用是将一个文件改写成对应于我们文件系统的结构，其主要功能为写入超级块、写入 imap 和 bmap、写入 inode table，以及创建一个根目录和测试文件。

超级块包含了文件系统的基本信息，其信息在上文中有详细描述。写入超级块信息，需要计算整个磁盘的大小，然后计算 imap、bmap 以及 inode table 的大小，这样才能确定各个区域在磁盘中的位置。这些工作都是在 init_disk 这个函数中完成的。基本逻辑为读取需要格式化的文件大小，计算出整个磁盘中的块的个数，简单的将块的个数与 inode 的个数等同起来；然后通过块数以及 inode 个数计算 imap 和 bmap 的大小。其中 bmap 的计算公式如下 (imap 与 bmap 一致)：

$$\text{bmap}_{\text{size}} = \frac{\text{blockcount}}{\text{HUST_BLOCKSIZE} * 8}$$

关键代码如下：

```
1. static int init_disk(int fd, const char* path)
2. {
3.     //获取基本信息
4.     //... ...
5.     //计算 bmap
6.     bmap_size = super_block.blocks_count/(8*HUST_BLOCKSIZE);
7.     super_block.bmap_block = RESERVE_BLOCKS;
8.
9.     if (super_block.blocks_count%(8*HUST_BLOCKSIZE) != 0) {
10.         bmap_size += 1;
11.     }
12.     bmap = (uint8_t *)malloc(bmap_size*HUST_BLOCKSIZE);
13.     memset(bmap,0,bmap_size*HUST_BLOCKSIZE);
14.
15.     //计算 imap
16.     imap_size = super_block.inodes_count/(8*HUST_BLOCKSIZE);
```

```

17.    super_block.imap_block = super_block.bmap_block + bmap_size;

18.

19.    if(super_block.inodes_count%(8*HUST_BLOCKSIZE) != 0) {
20.        imap_size += 1;
21.    }
22.    imap = (uint8_t *)malloc(imap_size*HUST_BLOCKSIZE);
23.    memset(imap,0,imap_size*HUST_BLOCKSIZE);
24.
25.    //计算 inode_table
26.    inode_table_size = super_block.inodes_count/(HUST_BLOCKSIZE/H
        UST_INODE_SIZE);
27.    super_block.inode_table_block = super_block.imap_block + imap
        _size;
28.    super_block.data_block_number = RESERVE_BLOCKS + bmap_size +
        imap_size + inode_table_size;
29.    super_block.free_blocks = super_block.blocks_count - super_bl
        ock.data_block_number - 1;
30.
31.    //设置 bmap 以及 imap
32.    //... ...
33.}

```

其中，imap 和 bmap 为 uint8_t 的全局数组。

计算完基本信息之后，需要将其写入文件并创建根目录和测试文件。文件创建的基本步骤如下：

- 1、检测（获取）磁盘（文件）大小，确认是否有足够的空间；
- 2、找的空闲的 inode 和 block，并标记 imap 和 bmap；
- 3、生成相应的数据，并写入对应的块中。对于根目录来讲，写入的数据为三个目录项，目录项的内容为文件（目录）名以及对应的 inode 编号。第一个目录项为当前目录和对应的 inode 编号 0，第二个目录项为上一级目录和对应的 inode 编号 0，第三个目录项为欢迎文件，内容为文件名“file”和对应的 inode 编号 1；
- 4、设置对应的 inode 信息，如是文件还是目录（mode 信息），则创建时间修改时间(i_ctime 和 i_mtime)、用户 id 和组 id 信息（i_uid 和 i_gid）等；
- 5、更新超级块信息。

在我们的文件系统写完之前，我们可以新建一个文件来测试我们的 mkfs 是

否能正常运行，通过 16 进制编辑器来查看是否功能正常。具体步骤如下：

- 1、运行下列命令创建文件。
- 2、`dd bs=4096 count=100 if=/dev/zero of=image`
- 3、编译 mkfs.c 文件 `gcc mkfs.c -o mkfs`
- 4、格式化 image 文件

```
HUST_OS_fs_experiment git:(master) ./mkfs ./image
Disk size is 409600
blocks count is 100
Super block written successfully!
Current seek is 16912 and rootdir at 40960
Create root dir successfully!
HUST_OS_fs_experiment git:(master) |
```

图 3.29 格式化 image 文件

5、通过 hexdum0 来查看文件的结构，结果如下图。检查发现，image 文件结构写入正确无误。

```
0000a210 66 69 6c 65 00 00 00 00 00 00 00 00 00 00 00 00 |file.....|
0000a220 00 00 00 00 00 00 00 00 ed 11 75 5a 31 7f 00 00 |.....uZl...|
0000a230 c0 a5 a8 5a 31 7f 00 00 68 0d 00 00 00 00 00 00 |..Zl...h....|
0000a240 40 62 a8 5a 31 7f 00 00 60 82 5a 1b d2 55 00 00 |eb.Zl...Z..U..|
0000a250 00 57 a8 5a 31 7f 00 00 cf 04 75 5a 31 7f 00 00 |.W.Zl...uZl...|
0000a260 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00 |.....!.....|
0000a270 0a 00 00 00 00 00 00 00 60 56 8b 19 d2 55 00 00 |.....V...U..|
0000a280 40 62 a8 5a 31 7f 00 00 00 00 00 00 00 00 00 00 |eb.Zl.....|
0000a290 00 00 00 00 00 00 00 00 e9 23 75 5a 31 7f 00 00 |.....#uZl...|
```

图 3.30 磁盘文件检查

3.5.3 文件系统的实现

一个通常意义上的文件系统驱动可以单独被编译成模块动态加载，也可以被直接编译到内核中，为了调试的方便，本文中的文件系统采用动态加载的方式实现。实现一个文件系统必须遵照内核的一些“规则”，以下将以递进的顺序阐述文件系统的实现过程。

一、文件系统的加载与卸载

首先为了能够成功加载文件系统，文件系统需要提供文件系统的名字，超级块的加载和删除方法。这些东西反应在 `file_system_type` 中。

```
1. struct file_system_type HUST_fs_type = {
2.     .owner = THIS_MODULE,
3.     .name = "HUST_fs",
4.     .mount = HUST_fs_mount,
5.     .kill_sb = HUST_fs_kill_superblock, /* unmount */
6. };
```

文件系统作为一种块设备驱动，需要实现 `module_init` 以及 `module_exit`。代码如下：

```

1. /* Called when the module is loaded. */
2. int HUST_fs_init(void)
3. {
4.     int ret;
5.     ret = register_filesystem(&HUST_fs_type);
6.     if (ret == 0)
7.         printk(KERN_INFO "Sucessfully registered HUST_fs\n");
8.     else
9.         printk(KERN_ERR "Failed to register HUST_fs. Error: [%d]\n",
10.             ret);
11.     return ret;
12.}
13.
14./* Called when the module is unloaded. */
15.void HUST_fs_exit(void)
16.{
17.    int ret;
18.    ret = unregister_filesystem(&HUST_fs_type);
19.    if (ret == 0)
20.        printk(KERN_INFO "Sucessfully unregistered HUST_fs\n");
21.    else
22.        printk(KERN_ERR "Failed to unregister HUST_fs. Error: [%d]\n",
23.            ret);
24.}
25.module_init(HUST_fs_init);
26.module_exit(HUST_fs_exit);
27.
28.MODULE_LICENSE("MIT");
29.MODULE_AUTHOR("cv");

```

我们可以看到，设备驱动加载的时候，驱动向内核注册了文件系统，而驱动卸载的时候，文件系统的信息也被删除。文件系统加载时调用的函数为 `HUST_fs_mount`，实际上，这个函数向内核注册了一个回调：

```
1. int HUST_fs_fill_super(struct super_block *sb, void *data, int si
    lent)
```

该函数用来与 VFS 交互从而生成 VFS 超级块的。在 HUST fs 中，超级块在磁盘的第二个 4096 字节上，即块号为 1。这个函数执行时会从磁盘中读取信息，填充到 VFS 提供的超级块结构体中，下列为部分关键代码。

```
1. int HUST_fs_fill_super(struct super_block *sb, void *data, int si
    lent) {
2.     struct buffer_head *bh;
3.     bh = sb_bread(sb, 1);
4.     struct HUST_fs_super_block *sb_disk;
5.     sb_disk = (struct HUST_fs_super_block *)bh->b_data;
6.     struct inode *root_inode;
7.     if (sb_disk->block_size != 4096) {
8.         printk(KERN_ERR "HUST_fs expects a blocksize of %d\n", 40
            96);
9.         ret = -EFAULT;
10.        goto release;
11.    }
12.    //fill vfs super block
13.    sb->s_magic = sb_disk->magic;
14.    sb->s_fs_info = sb_disk;
15.    sb->s_maxbytes = HUST_BLOCKSIZE * HUST_N_BLOCKS; /* Max file
        size */
16.    sb->s_op = &HUST_fs_super_ops;
17.}
```

从上述代码可以看出，首先用 `sb_read` 来读取磁盘上的内容，然后填充 `super_block` 结构体。值得注意的是，有关超级块的操作函数即 `superblock_operations` 也是在此处赋值的。由于 `super_block* sb` 在文件系统卸载之前是一直存在于内存中的，所以可以使用 `s_fs_info` 来存储原始的超级块信息，避免后期交互时再次读取磁盘。

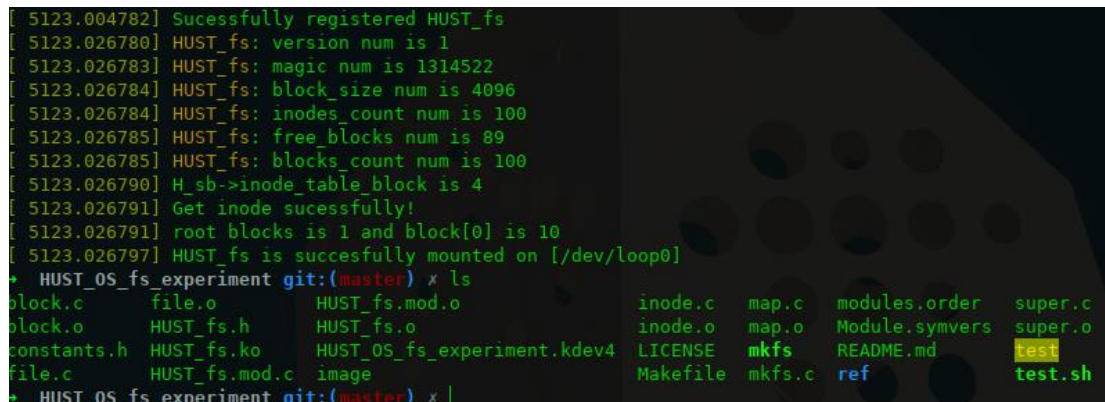
文件系统卸载的时候超级块信息需要被删除，因此 `HUST_fs_kill_super block` 的作用是释放该超级块，并通知 VFS 该挂载点已经卸载。

实现基本函数后，可以对文件系统进行挂载操作，挂载操作的脚本内容如下：

```
1. sudo umount ./test
2. sudo rmmod HUST_fs
3. dd bs=4096 count=100 if=/dev/zero of=image
```

4. ./mkfs image
5. insmod HUST_fs.ko
6. mount -o loop -t HUST_fs image ./test
7. dmesg

上述脚本将项目下的 test 文件夹作为文件系统的挂载点，并在挂载之后答应出了内核调试目录。成功执行该脚本的截图如下：



```

[ 5123.004782] Sucessfully registered HUST_fs
[ 5123.026780] HUST_fs: version num is 1
[ 5123.026783] HUST_fs: magic num is 1314522
[ 5123.026784] HUST_fs: block_size num is 4096
[ 5123.026784] HUST_fs: inodes_count num is 100
[ 5123.026785] HUST_fs: free_blocks num is 89
[ 5123.026785] HUST_fs: blocks_count num is 100
[ 5123.026790] H_sb->inode_table_block is 4
[ 5123.026791] Get inode sucessfully!
[ 5123.026791] root blocks is 1 and block[0] is 10
[ 5123.026797] HUST_fs is succesfully mounted on [/dev/loop0]
+ HUST_OS_fs_experiment git:(master) x ls
block.c      file.o      HUST_fs.mod.o  inode.c     map.c      modules.order  super.c
block.o      HUST_fs.h  HUST_fs.o      inode.o     map.o      Module.symvers  super.o
constants.h  HUST_fs.ko HUST_OS_fs_experiment.kdev4 LICENSE      mkfs        README.md      test
file.c       HUST_fs.mod.c image         Makefile    mkfs.c      ref            test.sh
+ HUST_OS_fs_experiment git:(master) x |

```

图 3.31 脚本运行情况

可以看到 test 目录已经挂载成功且内核调试信息显示文件系统挂载成功。

二、ls 命令的实现

加载文件系统之后第一个要实现的功能是读取文件系统中的数据，所以选择实现文件夹读取操作，这一操作在 2.x 内核中是 .readdir 函数指针，在最新版本中是 .iterate 函数指针。这个指针在保存在 file_operation 中，如下所示。

1. const struct file_operations HUST_fs_dir_ops = {
2. .owner = THIS_MODULE,
3. .iterate = HUST_fs_iterate,
4. };

HUST_fs_iterate 函数主要功能逻辑是读取 inode 的块数据，并将块数据中的 inode 和文件名通过 dir_emit 函数传输到 VFS 层。以根目录为例，根目录的包含三个数据项，分别是父目录，当前目录和欢迎文件，所以该函数会执行以下三个语句。

1. //参数分别表示上下文，文件/目录名，文件/目录名长度，inode 号，文件类型
2. dir_emit(ctx, ".", 1, 0, DT_DIR);
3. dir_emit(ctx, "..", 2, 0, DT_DIR);
4. dir_emit(ctx, "file", 4, 1, DT_REG);

完成该函数后，在填充根目录 inode 时将 HUST_fs_dir_ops 指针赋值，即可在挂在文件系统后执行 ls 命令。


```
→ HUST_OS_fs_experiment git:(master) x ls ./test
file
→ HUST_OS_fs_experiment git:(master) x |
```

图 3.32 ls 命令

如上图所示，成功看到了欢迎文件。但是此时还不能对文件进行任何操作，因为还没有实现其他的接口。

三、磁盘管理相关逻辑的实现

这个磁盘管理的内涵包括向磁盘写入和从磁盘取出读取 inode、更新 inode 信息和维护 imap、bmap、inode table 等操作。为了使磁盘上的内容有序的组合起来，磁盘空间的管理十分的重要，后续的文件读写操作都与此相关。

写入和删除 inode 的操作存放在 super_operations 这个结构体中。

```
1. const struct super_operations HUST_fs_super_ops = {
2.     .evict_inode = HUST_evict_inode,
3.     .write_inode = HUST_write_inode,
4. };
```

HUST_fs_super_ops 需要在填充超级块时赋值到 super_block 的 s_ops 字段中。HUST_write_inode 函数的功能是将内存中的 inode 保存在磁盘上。关键代码如下。

```
1. int HUST_write_inode(struct inode *inode, struct writeback_control *wbc)
2. {
3.     struct buffer_head * bh;
4.     struct HUST_inode * raw_inode = NULL;
5.     HUST_fs_get_inode(inode->i_sb, inode->i_ino, raw_inode);
6.     if (!raw_inode)
7.         return -EFAULT;
8.     raw_inode->mode = inode->i_mode;
9.     raw_inode->i_uid = fs_high2lowuid(i_uid_read(inode));
10.    raw_inode->i_gid = fs_high2lowgid(i_gid_read(inode));
11.    raw_inode->i_nlink = inode->i_nlink;
12.    raw_inode->file_size = inode->i_size;
13.    raw_inode->i_atime = (inode->i_atime.tv_sec);
14.    raw_inode->i_mtime = (inode->i_mtime.tv_sec);
15.    raw_inode->i_ctime = (inode->i_ctime.tv_sec);
16.    mark_buffer_dirty(bh);
17.    brelse(bh);
```

```

18.     return 0;
19. }

```

可以看到，该函数的将 vfs inode 中的相关信息存储到 HUST_inode 结构体中，然后写入磁盘。这是个单独的写入磁盘操作。事实上，当申请 inode 时，imap 也是需要检查刷新的，需要把相应位置标记为 1。同理，evict_inode 函数的作用时删除 inode，删除成功后需要刷新 imap 的值，把相应位置标记为 0。

设置和写入 map 的操作都在 map.c 中，以下以 imap 为例。对于 imap 来讲，申请 inode 的时候需要检查第一个空闲的 inode 编号，当 inode 被释放的时候也要及时清零对应的 imap。与此相关的函数如下。

```

1. //从磁盘中读取数据并存在 imap 数组中
2. int get_imap(struct super_block* sb, uint8_t* imap, ssize_t imap_
   size);
3. //在 vaddr 数组中找到第一个为 0 的 bit，这个函数用于定位空 inode 或者
   block
4. int HUST_find_first_zero_bit(const void *vaddr, unsigned size);
5. //将 imap 的某一位置 0 或者 1，并保存在磁盘上
6. int set_and_save_imap(struct super_block* sb, uint64_t inode_num,
   uint8_t value);
7. //定义的位操作宏如下
8. #define setbit(number,x) number |= 1UL << x
9. #define clearbit(number, x) number &= ~(1UL << x)

```

由于本文件系统并不是为了实际使用，所以上述的操作都没有考虑性能以及准确性问题。事实上，能够加上校验或者冗余备份是最好的。

四、读写文件内容

为了能够快速看到文件系统在正常工作，接下来需要实现文件的读写操作。文件读写操作按照一般处理，应该是实现在 struct file_operations 这个结构体中的。事实上，最开始是实现在这个结构体中的 read_iter 函数指针中的。但是比较有趣的一点是，如果实现了 struct address_space_operations 结构体中的函数，那么 struct file_operations 结构体中的函数则可以交由 VFS 实现。代码如下：

```

1. const struct file_operations HUST_fs_file_ops = {
2.     .owner = THIS_MODULE,
3.     .llseek = generic_file_llseek,
4.     .mmap = generic_file_mmap,
5.     .fsync = generic_file_fsync,
6.     .read_iter = generic_file_read_iter,

```

```

7.     .write_iter = generic_file_write_iter,
8. };
9. const struct address_space_operations HUST_fs_aops = {
10.     .readpage = HUST_fs_readpage,
11.     .writepage = HUST_fs_writepage,
12.     .write_begin = HUST_fs_write_begin,
13.     .write_end = generic_write_end,
14. };

```

上述的 generic 开头的函数是不需要我们手动实现的。address_space_operations 操作其实是实现了页高速缓存的一些操作。页高速缓存是 linux 内核实现的一种主要磁盘缓存，它主要用来减少对磁盘的 IO 操作，具体地讲，是通过把磁盘中的数据缓存到物理内存中，把对磁盘的访问变为对物理内存的访问。这些接口一旦实现，那么对文件的操作就可以转移到内存中，这就是为什么可以使用 generic 开头的这些函数来代替手写。

HUST_fs_readpage、HUST_fs_writepage 以及 HUST_fs_write_begin 都被注册回调到同一个函数 HUST_fs_get_block。HUST_fs_get_block 主要返回内核请求长度的数据。至于读写操作，内核调用 __bwrite 函数最终调用块设备驱动执行。因为未采用二级或者多级索引，故而 HUST_fs_get_block 函数逻辑比较简单，部分代码如下：

```

1. int HUST_fs_get_block(struct inode *inode, sector_t block,
2.     struct buffer_head *bh, int create)
3. {
4.     struct super_block *sb = inode->i_sb;
5.     if (block > HUST_N_BLOCKS)
6.         return -ENOSPC;
7.     struct HUST_inode H_inode;
8.     if (-1 == HUST_fs_get_inode(sb, inode->i_ino, &H_inode))
9.         return -EFAULT;
10.    if (H_inode.blocks == 0)
11.        if(alloc_block_for_inode(sb, &H_inode, 1))
12.            return -EFAULT;
13.    map_bh(bh, sb, H_inode.block[block]);
14.    return 0;
15.}

```

如上所示，该函数判断传入的 block 的大小，并将磁盘内容映射到 bh 中。后续的读写操作将有 VFS 帮我们完成。

五、inode 操作

Inode 操作涉及文件(夹)的创建删除,将 HUST_inode 映射到 VFS 中的 inode 等操作。具体实现的函数如下。

```
1. const struct inode_operations HUST_fs_inode_ops = {
2.     .lookup = HUST_fs_lookup,
3.     .mkdir = HUST_fs_mkdir,
4.     .create = HUST_fs_create,
5.     .unlink = HUST_fs_unlink,
6. };
```

HUST_fs_lookup 是其中比较复杂的一个函数,它负责将一个目录下的 inode 信息交由 VFS 管理。首先, HUST_fs_lookup 读取文件夹的内容,然后遍历文件夹下面的 HUST_inode,找到想要的 HUST_inode,根据不同的文件属性,申请 vfs_inode,并对不同的 vfs_inode 设置不同的操作。假设 vfs_inode 对应的是一个文件,那么就设置 vfs_inode->mapping->a_ops,如果 vfs_inode 对应的是文件夹,那么就设置 vfs_inode->f_ops = &HUST_fs_dir_ops;最后将 vfs_inode 注册到 VFS 中。这部分的关键代码如下:

```
1. struct dentry *HUST_fs_lookup(struct inode *parent_inode,
2. struct dentry *child_dentry, unsigned int flags)
3. {
4.     struct super_block *sb = parent_inode->i_sb;
5.     struct HUST_inode H_inode;
6.     //省略代码
7.     for (i = 0; i < H_inode.dir_children_count; i++) {
8.         if (strncmp
9.             (child_dentry->d_name.name, dtptr[i].filename,
10.             HUST_FILENAME_MAX_LEN) == 0){
11.             inode = iget_locked(sb, dtptr[i].inode_no);
12.             if (inode->i_state & I_NEW) {
13.                 inode_init_owner(inode, parent_inode, 0);
14.                 struct HUST_inode H_child_inode;
15.                 if (-1 == HUST_fs_get_inode(sb, dtptr[i].inode_no
16.                 , &H_child_inode))
17.                     return ERR_PTR(-EFAULT);
18.                 HUST_fs_convert_inode(&H_child_inode, inode);
19.                 inode->i_op = &HUST_fs_inode_ops;
```

```

19.         if (S_ISDIR(H_child_inode.mode)) {
20.             inode->i_fop = &HUST_fs_dir_ops;
21.         } else if (S_ISREG(H_child_inode.mode)) {
22.             inode->i_fop = &HUST_fs_file_ops;;
23.             inode->i_mapping->a_ops = &HUST_fs_aops;
24.         }
25.         inode->i_mode = H_child_inode.mode;
26.         inode->i_size = H_child_inode.file_size;
27.         insert_inode_hash(inode);
28.         unlock_new_inode(inode);
29.     }
30. }
31. }
32. //省略代码
33. }

```

只有在这里注册了相关函数，系统调用才能正常执行。不然就会出现不支持的操作这种报错信息。

.create 与.mkdir 都是对应了 inode 的创建，只是 inode 的属性不同而已，.create 创建普通文件而.mkdir 创建文件夹。所以这两个函数的功能被函数 HUST_fs_create_obj 所处理。这个函数接受新建文件（夹）的请求，检查磁盘的大小，检查是否有空余的 indoe，并且分配 inode 号，然后更新 imap 信息，最后更新超级块信息。由于该函数逻辑简单但是代码量比较大，故而不在此展示其具体实现。

在完成上述工作之后，文件系统基本已经完成了，这个系统采用线性（区别于 minix1 二级索引树来管理）的方式管理磁盘空间，支持基本的增删改查文件操作，支持文件权限，支持多用户。

4 设计环境

为实验方便，且防止编译内核对系统造成破坏，本实验选择 Windows+VMWare 虚拟机，前 1-4 个实验所用虚拟机的具体信息如下。

1、虚拟机硬件信息

由于编译内核需要较大的磁盘空间，并且为提高编译速度，将内存扩充至 2G，处理器数量增至 4。



图 4.1 Ubuntu 硬件信息

2、虚拟机操作系统

Linux 操作系统选择 Ubuntu，内核版本为 4.13.0，具体信息如下。



图 4.2 Ubuntu 操作系统信息

3、虚拟机编程环境

编辑器: Visual Studio Code/Vim

编译器: g++/gdb

图形编程: Qt 5.6.0

对于第 5 个文件系统的实验，由于系统功能复杂，涉及到很多内存操作，又运行在内核态，经常导致系统崩溃。为实验进度，选用更加轻量级的、开机加载速度更快的 arch linux 虚拟机系统，具体信息如下。

- 1、开发环境：KDevelop
- 2、内核版本：Linux-4.14
- 3、操作系统：arch Linux

5 实现记录

5.1 文件拷贝和进程并发

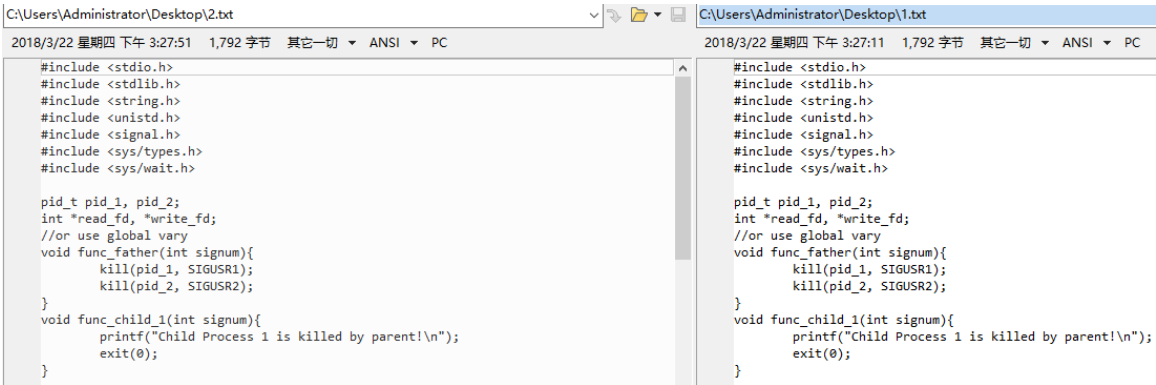
5.1.1 文件拷贝

按照设计过程，以 1024 字节数的字符数组作为 buf 缓冲区，协助读源文件、写目标文件操作的进行。采用命令行参数，第一个参数为可执行文件名，第二个参数为源文件路径，第三个参数为目标文件路径。为增强程序鲁棒性，检验参数个数是否为 3，并在源文件路径和目标文件路径出错时提示错误信息。编写程序位于 Harper/copy/copy.c 中，编辑器为 VS Code，编译并运行，结果如下。

```
→ copy ls
1.txt copy.c
→ copy gcc -o copy.out copy.c
→ copy ./copy.out 1.txt 2.txt
→ copy ls -ll
总用量 24
-rwxrw-rw- 1 harperchen harperchen 1792 Mar 22 15:27 1.txt
-rwxrw-rw- 1 harperchen harperchen 1792 Mar 22 15:27 2.txt
-rwxrw-rw- 1 harperchen harperchen 874 Mar 2 04:23 copy.c
-rwxrwxr-x 1 harperchen harperchen 9072 Mar 22 15:27 copy.ou
t
```

图 5.1 文件拷贝执行情况

生成 copy.out 可执行文件，执行，复制 1.txt 文件为 2.txt，将两份文件复制到 Windows 中利用 Beyond Compare 进行文本对比，结果如下，两个文件内容完全一致，且权限一致。



The image shows a side-by-side comparison of two C source files in the Beyond Compare application. Both files are located at C:\Users\Administrator\Desktop\ and are 1,792 bytes in size. The code in both files is identical and includes standard headers like <stdio.h>, <stdlib.h>, <string.h>, <unistd.h>, <signal.h>, <sys/types.h>, and <sys/wait.h>. The functions defined are func_father and func_child_1, which use kill() to send signals to child processes.

图 5.2 文本对比

5.1.2 进程并发

该实验设计图形编辑，利用 Qt 实现图形界面。编写如下三个函数，分别实现显示三个窗口的功能，每个函数使用一个定时器 timer，控制本窗口的信息更新，槽函数定义为 Qt 中的匿名函数，在 main 函数中调用三个函数，一个进程执行一个函数，实现进程并发。为显示整齐，调整窗口的大小和初始位置。

1. `int left(int argc, char *argv[]);`
2. `int mid(int argc, char *argv[]);`
3. `int right(int argc, char *argv[]);`

另外，对于显示文件，由于匿名函数传入参数的限制，考虑一次性读完所有文件内容后，不断增多显示内容，直至全部显示后不再刷新。运行截图如下。



图 5.3 进程并发-1

一分钟后截图如下，三个进程并发执行。

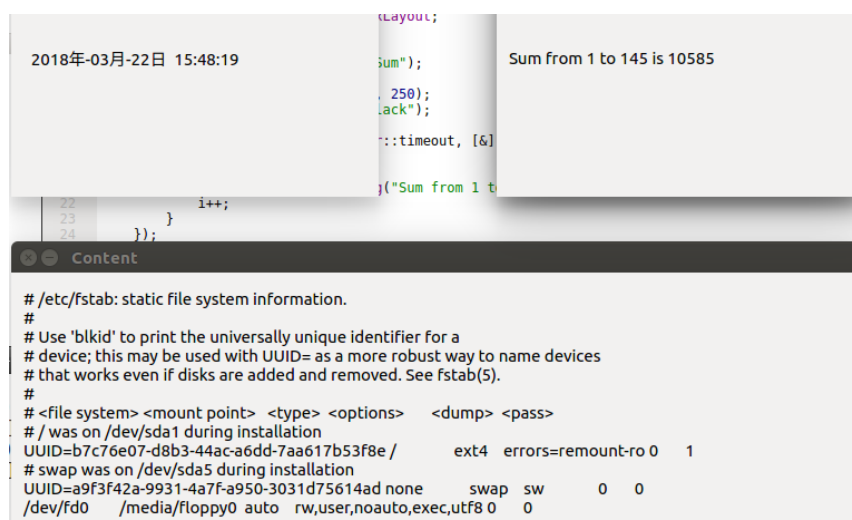


图 5.4 进程并发-2

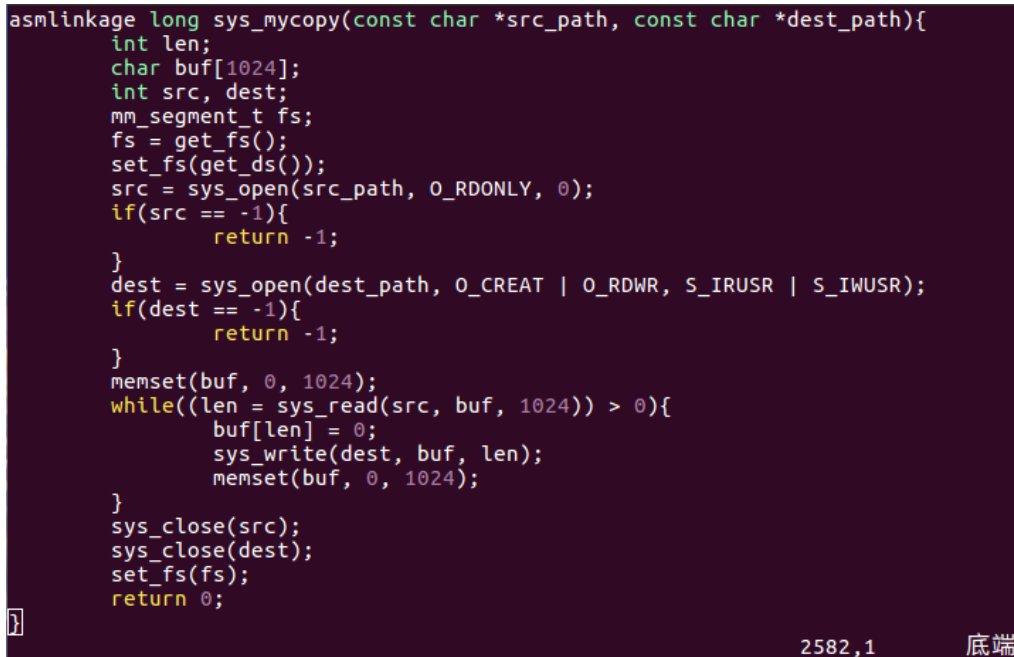
5.2 添加系统调用

5.2.1 代码编写和内核编译

本实验选择和源内核版本一致的源码 linux-4.13，未经改动的情况下编译安装后成功进入系统，说明代码可用。

按照设计过程，操作步骤如下：

1、将 kernel/sys.c 文件拷贝至 Harper/sys_copy 目录下，利用 vim 打开文件并在文件尾添加系统调用函数的定义，函数名为 sys_mycopy，参数 src_path 和 dest_path 分别为源文件和目标文件的路径。



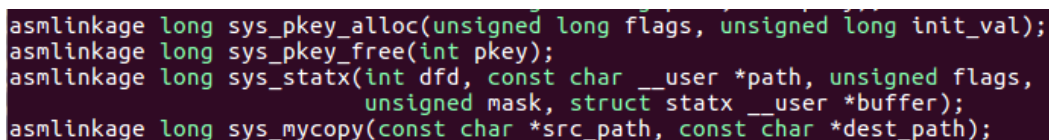
```
asmlinkage long sys_mycopy(const char *src_path, const char *dest_path){
    int len;
    char buf[1024];
    int src, dest;
    mm_segment_t fs;
    fs = get_fs();
    set_fs(get_ds());
    src = sys_open(src_path, O_RDONLY, 0);
    if(src == -1){
        return -1;
    }
    dest = sys_open(dest_path, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if(dest == -1){
        return -1;
    }
    memset(buf, 0, 1024);
    while((len = sys_read(src, buf, 1024)) > 0){
        buf[len] = 0;
        sys_write(dest, buf, len);
        memset(buf, 0, 1024);
    }
    sys_close(src);
    sys_close(dest);
    set_fs(fs);
    return 0;
}
```

图 5.5 复制文件内核代码

其中的 asmlinkage 用在大多数的系统调用中，某些情况下是需要明确的告诉编译器函数使用 stack 来传递参数，比如 X86 中的系统调用，是先将参数压入 stack 以后调用 sys_*函数的，所以所有的 sys_*函数都有 asmlinkage 来告诉编译器不要使用寄存器来编译。

内核代码使用 sys_open/sys_read/sys_write/sys_close 代替用户代码中的函数。操作成功函数返回 0，否则返回-1。

2、将 include/linux/syscalls.h 文件拷贝至 Harper/sys_copy 目录下，声明文件拷贝函数的原型，原型和定义保持一致。



```
asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);
asmlinkage long sys_statx(int dfd, const char __user *path, unsigned flags,
                          unsigned mask, struct statx __user *buffer);
asmlinkage long sys_mycopy(const char *src_path, const char *dest_path);
```

图 5.6 系统调用原型声明

3、将/arch/x86/entry/syscalls/syscall_64.tbl 文件拷贝至 Harper/sys_copy 目录下，为新系统调用分配一未使用的系统调用号，在该文件末尾登记系统调用号，显示如下。

327	64	preadv2	sys_preadv2
328	64	pwritev2	sys_pwritev2
329	common	pkey_mprotect	sys_pkey_mprotect
330	common	pkey_alloc	sys_pkey_alloc
331	common	pkey_free	sys_pkey_free
332	common	statx	sys_statx
333	64	mycopy	sys_mycopy

图 5.7 系统调用号

系统调用号为 333 号，mycopy 系统调用名，sys_mycopy 为系统调用函数名。

4、用三个文件覆盖原来的文件，进入/usr/src/linux-4.13 目录下,按照内核编译流程编译，过程如下。

```

→ sys_copy cp sys.c /usr/src/linux-4.13/kernel/sys.c
→ sys_copy cp syscalls.h /usr/src/linux-4.13/include/linux/syscalls.h
→ sys_copy cp syscall_64.tbl /usr/src/linux-4.13/arch/x86/entry/syscalls/syscall_64.tbl
→ sys_copy cd /usr/src/linux-4.13
→ linux-4.13 sudo make menuconfig
[sudo] harperchen 的密码:
scripts/kconfig/mconf Kconfig

*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.

→ linux-4.13 sudo make -j4
scripts/kconfig/conf --silentoldconfig Kconfig
CHK include/config/kernel.release
SYSHDR arch/x86/entry/syscalls/../../../../include/generated/asm/unistd_64_x32.h
SYSTBL arch/x86/entry/syscalls/../../../../include/generated/asm/syscalls_64.h

```

图 5.8 内核编译

5、重启系统，进入 grub 页面，选择进入新系统 linux 4.13.0，成功进入。

```

*Ubuntu, Linux 4.13.0-36-generic
Ubuntu, with Linux 4.13.0-36-generic (upstart)
Ubuntu, with Linux 4.13.0-36-generic (recovery mode)
Ubuntu, Linux 4.13.0
Ubuntu, with Linux 4.13.0 (upstart)
Ubuntu, with Linux 4.13.0 (recovery mode)
Ubuntu, Linux 4.13.0.old
Ubuntu, with Linux 4.13.0.old (upstart)
Ubuntu, with Linux 4.13.0.old (recovery mode)

```

图 5.9 Grub 页面

此时执行 uname -r 命令，查看内核版本，显示为 4.13.0，内核编译成功。

```

→ ~ uname -r
4.13.0

```

图 5.10 内核版本

5.2.2 系统调用测试

进入新系统后，编写测试程序，代码路径为 Harper/test_mycopy.c。测试文件同样采用命令行参数，调用 syscall 命令，系统调用号为 333，传入源文件路径和目标文件路径，编译和运行测试程序如下。

```
→ sys_copy ls
1.txt sys.c syscall_64.tbl syscalls.h test_mycopy.c
→ sys_copy gcc -o mycopy.out test_mycopy.c
→ sys_copy ./mycopy.out 1.txt 2.txt
```

图 5.11 系统调用测试

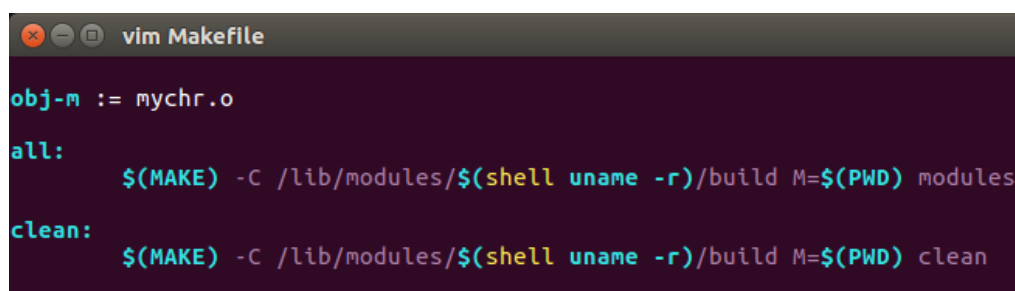
1.txt 为源文件，2.txt 为目标文件，打开比较两份文件完全一致，系统调用添加成功。

5.3 添加设备驱动程序

字符设备代码存放在 Harper/dev_char 目录下。模块代码命名为 mychr.c，按照设计过程的框架编写，填充相应函数，选择 MAJOR_NUM 即主设备号为 244，临时文件路径为/tmp/mychr。在函数中，使用 printk 函数输出调试信息，便于查错和修改。

5.3.1 编译驱动程序

采用 kbuild 标准编写 Makefile 文件如下。



```
vim Makefile

obj-m := mychr.o

all:
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

图 5.12 Makefile 文件

obj-m := mychr.o 表示会编译一个模块（-m），生成的 mychr.o 文件来自于 mychr.c 文件。

uname -r 命令返回的实际当前正在使用的内核版本，例如 4.13.0-36-generic，则-C 选项后面完整的路径是：/lib/modules/ 4.13.0-36-generic/build。

执行 make 命令编译产生了许多中间文件，其中.o 文件是对象文件，.ko 文件是 kernel object，得到.ko 文件后，利用 insmod 命令加载生成的模块。

```
→ dev_char ls
Makefile mychr.c test_mychr.c
→ dev_char make
make -C /lib/modules/4.13.0-36-generic/build M=/home/harperchen/Harper/dev_char
modules
make[1]: Entering directory '/usr/src/linux-headers-4.13.0-36-generic'
CC [M] /home/harperchen/Harper/dev_char/mychr.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/harperchen/Harper/dev_char/mychr.mod.o
LD [M] /home/harperchen/Harper/dev_char/mychr.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.13.0-36-generic'
→ dev_char ls
Makefile Module.symvers mychr.ko mychr.mod.o test_mychr.c
modules.order mychr.c mychr.mod.c mychr.o
→ dev_char sudo insmod ./mychr.ko
[sudo] harperchen 的密码:
```

图 5.13 模块加载

利用 lsmod 查看是否加载成功，并利用 tail /var/log/kern.log 命令查看系统日志，结果如下图，模块已加载。

```
→ dev_char lsmod
Module              Size  Used by
mychr                16384  0
btrfs               1101824  0
xor                  24576  1 btrfs
```

图 5.14 已加载模块

```
Mar 22 19:45:13 ubuntu kernel: [35727.334595] mychr: loading out-of-tree module
taints kernel.
Mar 22 19:45:13 ubuntu kernel: [35727.364266] mychr: module verification failed:
signature and/or required key missing - tainting kernel
Mar 22 19:45:13 ubuntu kernel: [35727.428629] I will init my char device module!
Mar 22 19:45:13 ubuntu kernel: [35727.428650] Success to register char device!
Mar 22 19:45:13 ubuntu kernel: [35727.428753] Success to create file!
```

图 5.15 系统日志

5.3.2 测试模块功能

编写测试程序 test_mychr.c，用户进程通过设备文件同硬件打交道，对设备文件进行打开、关闭、读写等操作，测试其功能就是测试编写的设备驱动程序中的子函数的功能。

首先查看设备的主设备号，模块加载成功后，在文件 /proc/devices 中能看看到新增加的设备，包括设备名 mychr 和主设备号 244。然后根据主设备号生成设备文件，命令执行情况如下。

```
→ dev_char cat /proc/devices | grep mychr
244 mychr
→ dev_char sudo mknod /dev/mychr0 c 244 0
→ dev_char
```

图 5.16 生成设备文件

其中，mychr0 为设备文件名，244 为主设备号，0 为从设备号，c 表示字符设备。在 test_mychr.c 中，对设备文件 mychr0 调用 open/write/lseek/read/close 函数，代码逻辑如下，根据返回值输出函数执行情况，并查看内核日志，判断驱动程序是否执行正确。

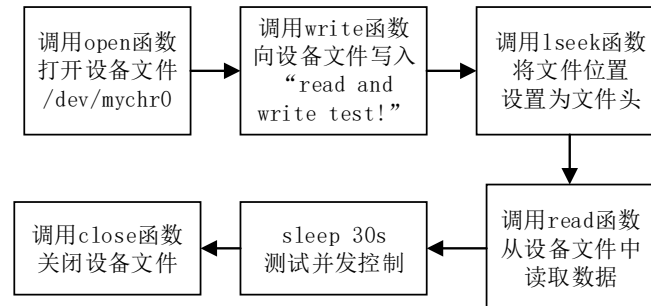


图 5.17 测试程序流程

编译并运行测试程序，结果如下。

```

→ dev_char gcc -o test.out test_mychr.c
→ dev_char sudo ./test.out
[sudo] harperchen 的密码:
open /dev/mychr0 successfully!
write data is read and write test!
User write size is 20
User read size is 20
read data is read and write test!
close /dev/mychr0 successfully!
  
```

图 5.18 测试系统调用

输出文件打开/关闭、读写成功的信息，打开/tmp/mychr 文件，发现数据写入成功。

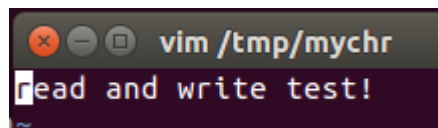


图 5.19 /tmp/mychr 文件内容

```

[ 1182.028803] open filp is mychr0
[ 1182.028808] Success to open file!
[ 1182.029045] write filp is mychr0 f_pos is 0
[ 1182.029047] tmp file pos is 0
[ 1182.029051] Content of buf is read and write test!
[ 1182.029102] Success to write buf!
[ 1182.029104] Kernel write size is 20
[ 1182.029106] tmp file pos is 20
[ 1182.029239] read filp is mychr0 f_pos is 0
[ 1182.029241] tmp file pos is 0
[ 1182.029251] Content of buf is read and write test!
[ 1182.029252] Success to read content of file!
[ 1182.029253] Kernel read size is 20
[ 1182.029255] tmp file pos is 20
[ 1205.746991] open filp is mychr0
[ 1212.029855] release filp is mychr0
[ 1212.029859] Success to close file!
  
```

查看/var/log/kern.log 文件，查看日志信息如下。

图 5.20 日志文件内容

在编写模块代码时，对子函数的参数进行输出，发现传入参数不论是 inode 节点还是 file 结构的指针确实都是设备文件的信息。

另外，由于在编写驱动时，对打开设备文件的进程数加以控制，因此测试程序中 sleep30s，并在这 30s 内使用另一终端执行测试文件打开设备，模拟多进程同时打开设备的过程，运行情况如下，两个设备同时打开设备文件时，第二个进程会打开失败，实验成功。

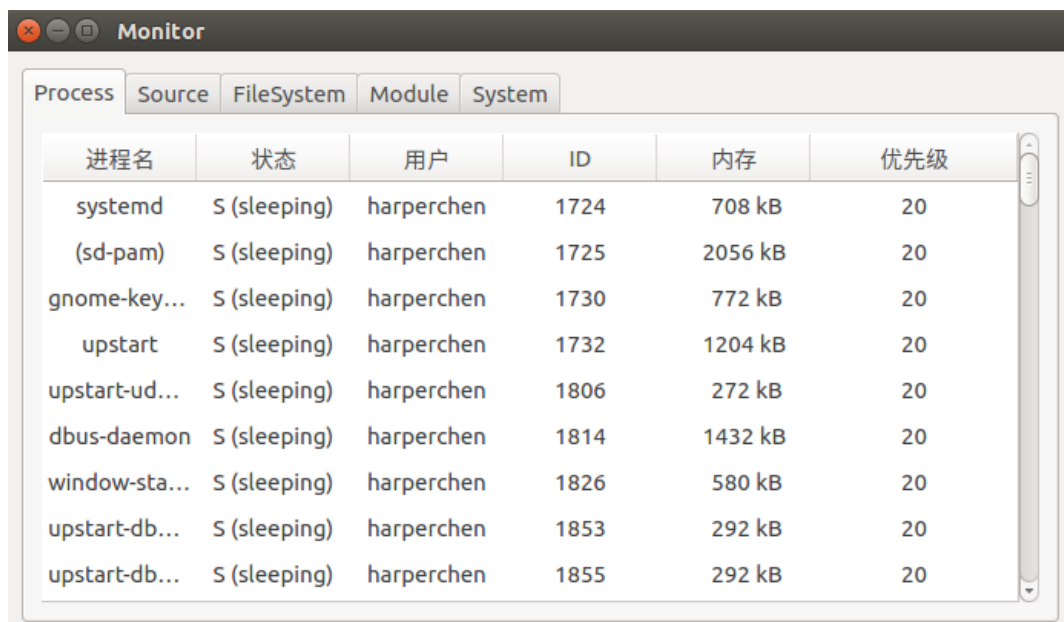
```
→ dev_char sudo mknod /dev/mychr0 c 244 0
→ dev_char ls
Makefile      Module.symvers  mychr.ko
modules.order mychr.c          mychr.mod.c
→ dev_char sudo ./test.out
open /dev/mychr0 successfully!
write data is read and write test!
User write size is 20
User read size is 20
read data is read and write test!
→ ~ cd Harper/dev_char
→ dev_char sudo ./test.out
[sudo] harperchen 的密码:
error open!
: Device or resource busy
→ dev_char
```

图 5.21 并发控制测试

5.4 系统监控

按框架设计完成后，编译运行，该功能模仿 Linux 下 monitor 程序的界面，分别监控系统的进程、模块、CPU、内存、网络 and 文件系统等信息，截图如下。

1、Process 页面



进程名	状态	用户	ID	内存	优先级
systemd	S (sleeping)	harperchen	1724	708 kB	20
(sd-pam)	S (sleeping)	harperchen	1725	2056 kB	20
gnome-key...	S (sleeping)	harperchen	1730	772 kB	20
upstart	S (sleeping)	harperchen	1732	1204 kB	20
upstart-ud...	S (sleeping)	harperchen	1806	272 kB	20
dbus-daemon	S (sleeping)	harperchen	1814	1432 kB	20
window-sta...	S (sleeping)	harperchen	1826	580 kB	20
upstart-db...	S (sleeping)	harperchen	1853	292 kB	20
upstart-db...	S (sleeping)	harperchen	1855	292 kB	20

图 5.22 Process 页面

该页面通过读取 /proc/pid/stat 文件和 /proc/pid/status 文件，获取需要的信息后，以表格的形式展示到该页面。由于设置定时器，所以每隔一段时间，

重新读取文件数据，刷新显示内容，具体实现函数为 `void setProcessContent()`。

2、Source 界面

该界面显示 CPU 利用率、内存利用率和网络数据。CPU 利用率通过读取 `/proc/stat` 文件，间隔一小段时间获取两次 CPU 快照，按照公式计算而得。内存信息则是读取 `/proc/meminfo` 文件，获取相应信息后，以进度条形式显示百分比，网络信息通过 `/proc/net/dev` 文件，获取已接受和已发送数据的总量并显示。

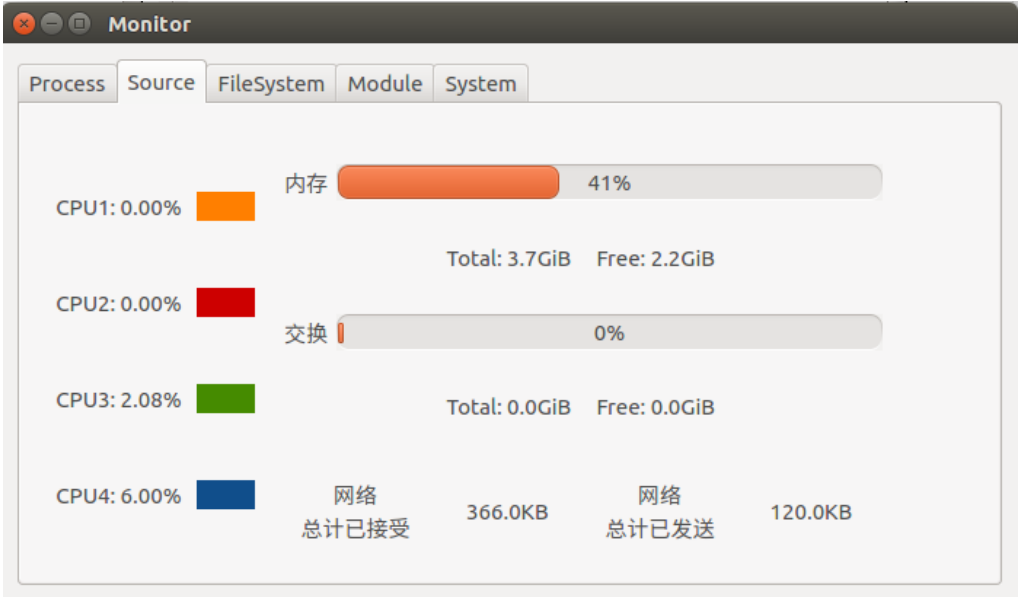


图 5.23 Source 页面

同样的，该界面也是定时刷新，所有数据均为动态，具体实现在 `void setResourceContent();` 函数和 `void setCpuContent();` 函数中，后者用于专门计算 CPU 利用率。

3、FileSystem 界面

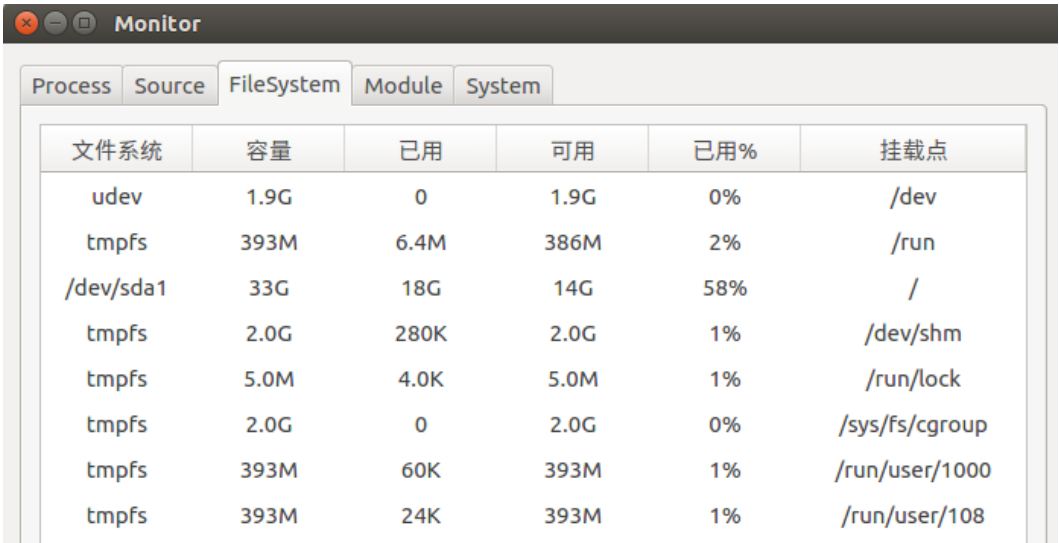
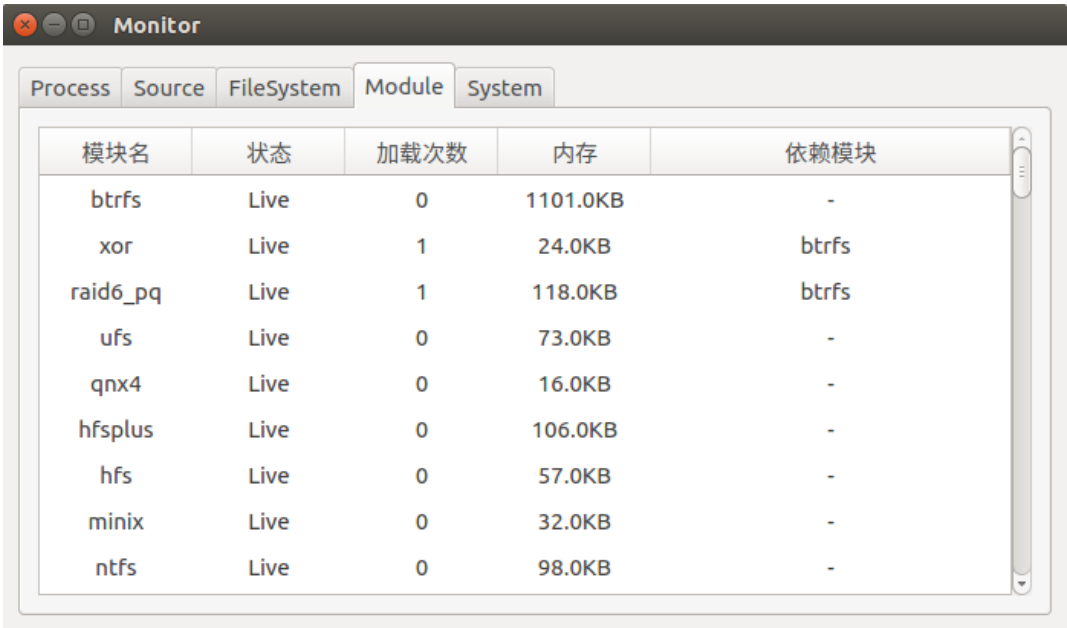


图 5.24 FileSystem 页面

该界面动态显示已挂载的文件系统的信息，实现在 `void setFileContent();`

函数中。

4、Module 界面

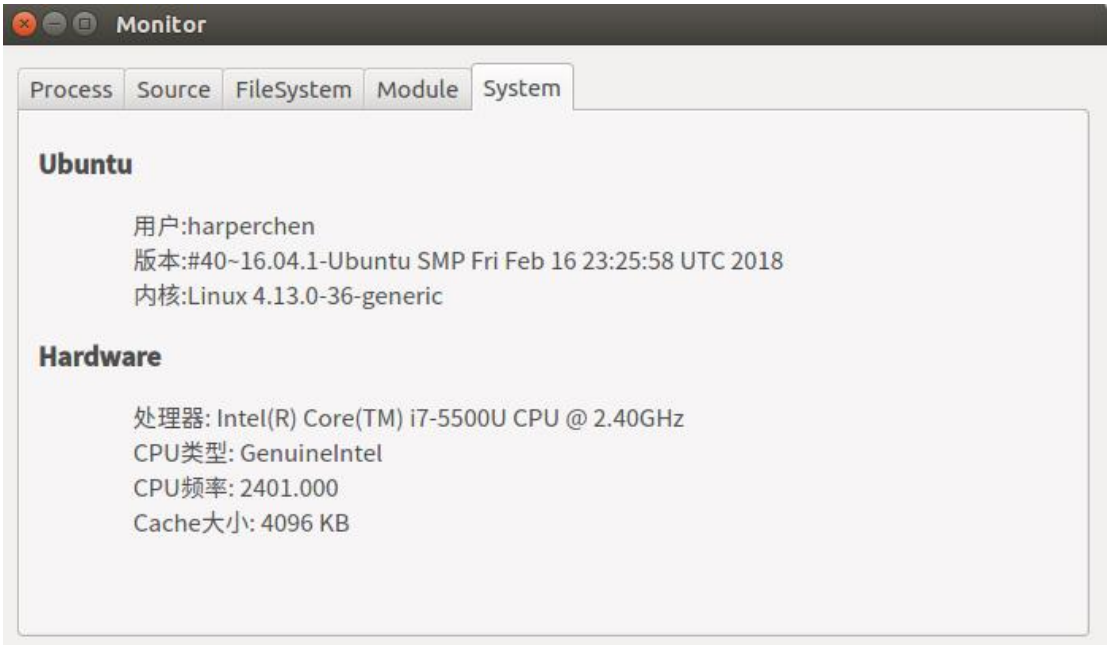


模块名	状态	加载次数	内存	依赖模块
btrfs	Live	0	1101.0KB	-
xor	Live	1	24.0KB	btrfs
raid6_pq	Live	1	118.0KB	btrfs
ufs	Live	0	73.0KB	-
qnx4	Live	0	16.0KB	-
hfsplus	Live	0	106.0KB	-
hfs	Live	0	57.0KB	-
minix	Live	0	32.0KB	-
ntfs	Live	0	98.0KB	-

图 5.25 Module 页面

该界面动态显示已安装的模块信息，通过读取/proc/modules 文件获取有关信息，具体实现在 void setModulesContent();函数中。

5、System 界面



Ubuntu
用户:harperchen
版本:#40~16.04.1-Ubuntu SMP Fri Feb 16 23:25:58 UTC 2018
内核:Linux 4.13.0-36-generic
Hardware
处理器: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
CPU类型: GenuineIntel
CPU频率: 2401.000
Cache大小: 4096 KB

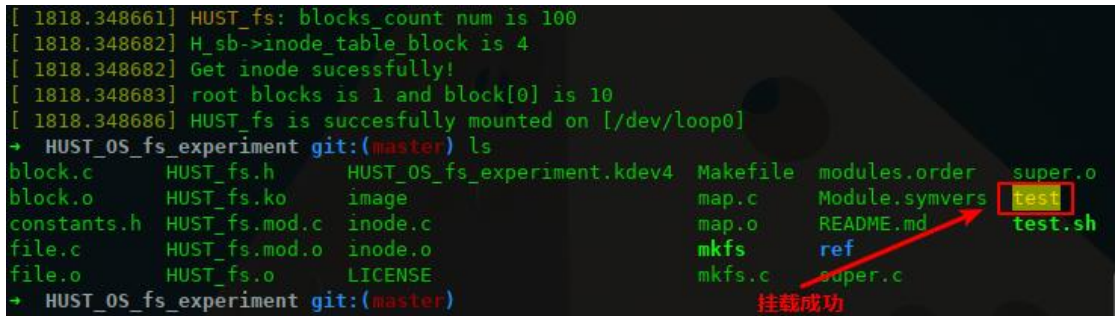
图 5.26 System 页面

该界面显示虚拟机的操作系统信息、内核版本和硬件信息，由于信息固定，因此不动态刷新该界面。CPU 有关信息通过读取/proc/cpuinfo 文件获得，具体实现在函数 void setSystemContent();中。

5.5 文件系统

一、编译并且挂载

执行 test.sh，编译并挂载 image 文件到 test 目录。下图中，test 文件夹以绿色高亮，表示挂载成功，而且可以看到 dmesg 也输出了挂载成功的调试信息。



```
[ 1818.348661] HUST_fs: blocks_count num is 100
[ 1818.348682] H_sb->inode_table_block is 4
[ 1818.348682] Get inode sucessfully!
[ 1818.348683] root blocks is 1 and block[0] is 10
[ 1818.348686] HUST_fs is sucessfully mounted on [/dev/loop0]
+ HUST_OS_fs_experiment git:(master) ls
block.c      HUST_fs.h      HUST_OS_fs_experiment.kdev4  Makefile  modules.order  super.o
block.o      HUST_fs.ko      image                        map.c      Module.symvers  test
constants.h  HUST_fs.mod.c   inode.c                      map.o      README.md       test.sh
file.c       HUST_fs.mod.o   inode.o                      mkfs       ref
file.o       HUST_fs.o       LICENSE                      mkfs.c     super.c
+ HUST_OS_fs_experiment git:(master)
```

图 5.27 编译并挂载

二、测试多级目录，多用户功能

进入 test 文件夹，用不同的用户创建文件，然后用 ls -l 查看，可以看到文件的所属是不同的。



```
+ test ls -l
总用量 1
-rw-r--r-- 1 chen chen 0 3月 22 15:20 chen
-rwxrwxrwx 1 root root 0 3月 22 15:14 file
drwxr-xr-x 1 chen chen 1 3月 22 15:20 wei
```

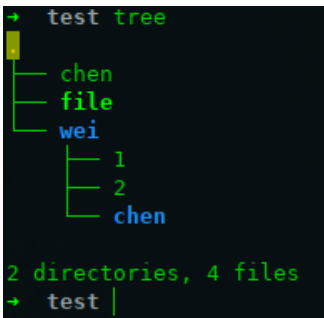
图 5.28 测试多用户

上图中可以看到，file 文件是文件系统的欢迎文件，所以所属着是 root，后面在用户 chen 创建的文件的所有组和用户都变成了“chen”。这说明是支持多用户的。

接下来建立多级目录以及多个文件，然后执行

yaourt -S tree

安装 tree，tree 类似于 windows 上的 tree 命令，能够树形来显示目录结构，可以很好的展示多级目录。结果如下图所示。



```
+ test tree
.
├── chen
├── file
└── wei
    ├── 1
    ├── 2
    └── chen

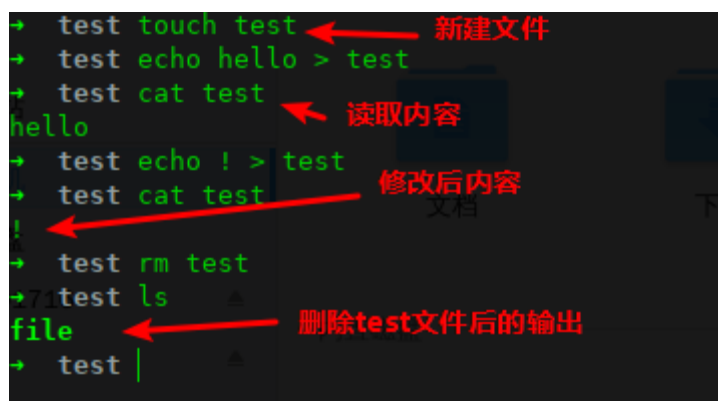
2 directories, 4 files
+ test
```

图 5.29 树形结构

其中 wei 和 chen 都是目录，而 1 和 2 都是文件。

三、测试文件的增删查改

建立 test 文件，初始为空，然后查看内容；向 test 文件中写如内容，观察是否写入成功；修改 test 文件内容，然后看是否修改成功；最后删除文件，再次查看 ls 的输出。结果如下。



```
→ test touch test
→ test echo hello > test
→ test cat test
hello
→ test echo ! > test
→ test cat test
!
→ test rm test
→ test ls
file
→ test
```

新建文件

读取内容

修改后内容

删除test文件后的输出

图 5.30 文件增删测试

6 设计心得

本次课程设计相比之前的实验，更注重实现，不论是编译内核还是模块加载，某种程度上说都是比较陌生的概念，从网上查询很多博客辅以实践才明白具体原理。同时，由于图形界面的要求，对 Qt 的使用也更加熟悉，包括定时器、匿名函数和信号和槽。但是，实验过程中也遇到了很多问题，列举如下。

1、匿名函数传参数的问题：

在并发程序中，由于函数体本身较小，槽函数用匿名函数编写，但参数传递上出现问题。

2、内核编译过程：

内核编译时，由于是虚拟机，处理器起初数量为 1，内存不足 1G，磁盘为 20G，编译过程中显示空间不足且时间较慢，因此更改处理器数量为 4，内存增至 2G，磁盘增至 35G。另外，初次编译后，重启并未显示 Grub 界面，查询资料改写 grub 配置文件后才成功。

3、模块加载

驱动程序编写格式很严格，函数原型固定，对于模块初始化函数和卸载函数，就算无参数也要加 void，否则编译失败。

4、系统监控

系统监控由于不断的刷新读文件，起初反应很慢，后期经优化速度才处于正常范围。

5、文件系统

文件系统编写过程中，由于有 bug，内存操作失误，导致驱动模块崩溃，进而使得操作系统无法运行，给调试开发工作带来很大困扰。

经查询资料和询问同学，上述问题得以解决，在解决过程的问题中感觉收获颇多，进步很快。尤其是文件系统，从最开始设计到最终的实现，每一步都让我感叹 Linux 内核设计的精妙与优雅。特别 VFS，巧妙的避开了不同文件系统中磁盘布局的差异，给上层的系统调用提供了统一的接口，同时又给下层的文件系统的时间提供了一个简洁的框架。对于文件系统层面来说，文件系统的编写者也不需要了解磁盘究竟是如何读取和刷新信息的，也不需要实现缓存系统，这些都有 VFS 和块设备驱动代劳。