



# 华中科技大学

## 操作系统课程设计报告

姓 名：张丹朱  
学 院：计算机科学与技术  
专 业：计算机科学与技术  
班 级：CS1409  
学 号：U201414797  
指导教师：谢美意

分数	
教师签名	

2017 年 03 月 19 日

# 目 录

<b>设计内容 (1)</b>	<b>1</b>
1 实验目的	1
2 实验内容	1
3 实验设计	1
3.1 平台环境	1
3.2 方案设计	1
4 实验调试	3
4.1 实验步骤	3
4.2 实验调试及结果	4
4.3 实验心得	4
<b>设计内容 (2)</b>	<b>5</b>
1 实验目的	5
2 实验内容	5
3 实验设计	5
3.1 平台环境	5
3.2 方案设计	5
4 实验调试	6
4.1 实验步骤	6
4.2 实验调试及结果	7
4.3 实验心得	8
<b>设计内容 (3)</b>	<b>9</b>
1 实验目的	9
2 实验内容	9
3 实验设计	9
3.1 平台环境	9
3.2 方案设计	9
4 实验调试	10
4.1 实验步骤	10
4.2 实验调试及结果	10
4.3 实验心得	12
<b>设计内容 (4)</b>	<b>13</b>
1 实验目的	13
2 实验内容	13
3 实验设计	13
3.1 平台环境	13
3.2 方案设计	13
4 实验调试	14
4.1 实验步骤	14
4.2 实验调试及结果	15
4.3 实验心得	17
<b>设计内容 (5)</b>	<b>18</b>

1 实验目的.....	18
2 实验内容.....	18
3 实验设计.....	18
3.1 平台环境.....	18
3.2 方案设计.....	18
4 实验调试.....	19
4.1 实验步骤.....	19
4.2 实验调试及结果.....	19
4.3 实验心得.....	20
<b>附录 实验代码.....</b>	<b>21</b>
设计内容（1）.....	21
设计内容（2）.....	24
设计内容（3）.....	25
设计内容（4）.....	28
设计内容（5）.....	35

# 设计内容 (1)

## 1 实验目的

(1) 熟悉和理解 Linux 编程环境

## 2 实验内容

- 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，使用基于文本的终端图形界面编程库或图形界面（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时间，一个窗口实时监测 CPU 的利用率，一个窗口做 1 到 100 的累加求和，刷新周期分别为 1 秒，2 秒和 3 秒）。

## 3 实验设计

### 3.1 平台环境

操作系统：Ubuntu16.04LTS

内核版本：4.4.0-64-generic

编译环境：gcc

### 3.2 方案设计

- 1) 文件拷贝程序

使用 `fopen`、`fread`、`fwrite` 等文件操作函数，打开源文件并将其内容写入新文件，借用一个固定大小的缓冲区（字符数组）来进行读写文件操作。

具体流程如下：

首先判断参数个数是否正确，正确才以只读方式打开源文件，若文件打开失败则报错并返回，否则以读写方式打开目标文件，同样判断是否打开成功。若是源文件与目标文件都成功打开则使用 `fread` 从源文件读数据到缓冲区 `buf`，再通过 `fwrite` 写入目标文件，重复复制直到 `fread` 读到的字节数为 0。

流程图如下：

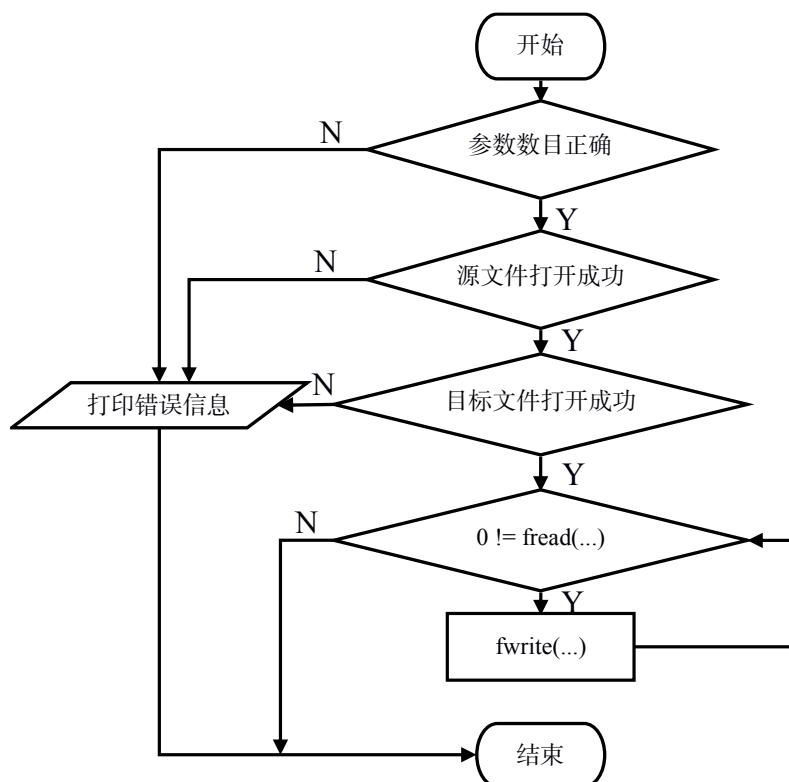


图 1.1 拷贝文件流程图

## 2) 分窗口显示程序

使用 GTK 进行图形界面的绘制，写 4 个 `main` 函数，在主文件中通过 `execv` 来 `fork` 三个进程，每个进程使用 GTK 创建各自的显示窗口并使用 GTK 中的 `Label` 组件来显示相应的信息，其中，时间通过获取当前时间并按相应格式（年-月-日

时-分-秒)来显示,cpu使用率通过读取”/proc/stat”文件下相关cpu占用率来计算获得cpu使用率。

三个窗口相应进程的实现方式皆类似,故此处仅已cpu使用率的显示进行详细说明。

首先进行初始化,调用gtk\_init()函数,然后创建新窗口,设置尺寸,退出响应等。接着在窗口中添加一个label控件,使用g\_timeout\_add函数设置每隔3000毫秒调用一次回调函数get\_cpu,在get\_cpu函数中进行cpu使用率的获取与计算并更新label的text显示。

get\_cpu函数首先打开”/proc/stat”文件,依次读取第一行的2到5项内容,依次为用户模式(user)、低优先级的用户模式(nice)、内核模式(system),以及空闲的处理器时间(idle)。CPU使用率可根据公式 $100 * (user + nice + system) / (user + nice + system + idle)$ 进行计算。当计算获得CPU使用率后再调用gtk\_label\_set\_text更改label的显示值。如此做到了每隔3秒进行cpu使用率的刷新。

## 4 实验调试

### 4.1 实验步骤

- 1)根据设计方案(1)编写mycopy.c文件,编译、调试并进行测试;
- 2)根据设计方案(2)编写主文件dis\_three.c以及显示三个窗口的程序dis\_time.c,dis\_sum.c,dis\_cpu.c,分别编译,最后运行dis\_three进行调试与测试。

## 4.2 实验调试及结果

1) make 之后运行 dis\_three,获得如下图 1.2 所示结果。



图 1.2 实验一结果图

## 4.3 实验心得

通过本次实验，进一步熟悉了 Linux 下的操作，回顾了如何进行多进程的编写。对于 GTK 的使用有了一个初步的了解，明白了 GTK 画图形界面的基本操作与流程。同时对 CPU 信息的获取有了一定的了解，明白了 CPU 信息存储在何处、如何计算 CPU 的使用率等基本信息。

在如何每隔一定时间更新显示的内容这个问题上，通过查阅资料 and 与同学讨论大致有两种思路：一是像我实际使用的那样，通过 `g_timeout_add` 函数每隔一定时间调用一次回调函数并在回调函数中更新内容；另一种方法是在当前进程中开一个线程，由该线程每隔一定时间（`sleep`）来更新相应的信息。开拓了我的思维。

# 设计内容 (2)

## 1 实验目的

(1) 掌握添加系统调用的方法

## 2 实验内容

- 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

## 3 实验设计

### 3.1 平台环境

操作系统：Ubuntu16.04LTS

内核版本：4.4.4

编译环境：gcc

### 3.2 方案设计

首先从网上下载一个新的内核，在/arch/x86/entry/syscalls/syscall\_64.tbl 中添加系统调用号，接着在 include/linux/syscalls 中添加函数申明，最后在 kernel/sys.c 中添加系统调用函数的定义。



然后编译内核。

编译成功之后重启选择新的内核

## 4 实验调试

### 4.1 实验步骤

1) 在/usr/src/linux-4.4.4/arch/x86/entry/syscalls/syscall\_64.tbl 中添加系统调用号, 如下图 2.1 所示。

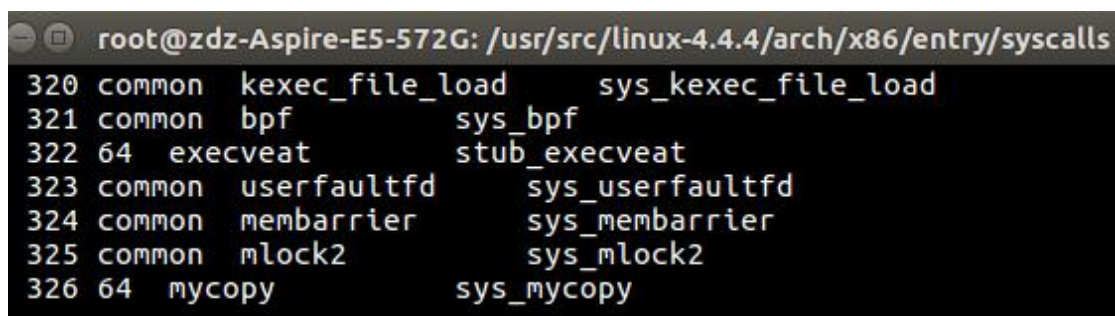


图 2.1 添加系统调用号

2) 在/usr/src/linux-4.4.4/include/linux/syscalls 中添加函数申明, 如下图 2.2 所示。

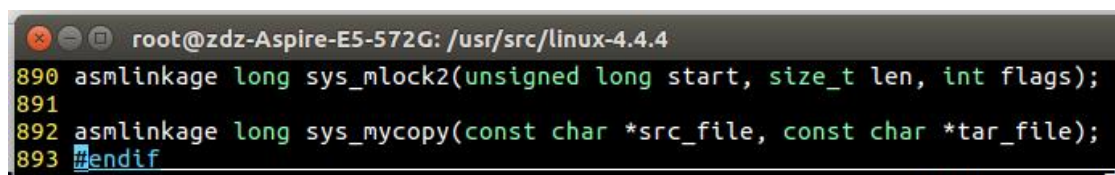


图 2.2 添加系统调用函数声明

3) 在/usr/src/linux-4.4.4/kernel/sys.c 中添加系统调用函数的定义。

4) 编译内核

sudo menuconfig 直接选择 save, 然后退出即可, 生成.config

sudo make

sudo make modules\_install 安装模块

sudo make install 安装内核

5) 重启, 选择进入新的内核。

6) 编写测试文件进行测试。

## 4.2 实验调试及结果

1) 在安装内核的时候提示 No space left on device;安装失败, 如下图 2.3 所示。

```
zdz@zdz-Aspire-E5-572G:/usr/src/linux-4.4.4$ sudo make install
sh ./arch/x86/boot/install.sh 4.4.4 arch/x86/boot/bzImage \
    System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.4.4 /boot/vmlinuz-4.4.4
run-parts: executing /etc/kernel/postinst.d/dkms 4.4.4 /boot/vmlinuz-4.4.4
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.4.4 /boot/vmlinuz-4.4.4
update-initramfs: Generating /boot/initrd.img-4.4.4

gzip: stdout: No space left on device
E: mkinitramfs failure cpio 141 gzip 1
update-initramfs: failed for /boot/initrd.img-4.4.4 with 1.
run-parts: /etc/kernel/postinst.d/initramfs-tools exited with return code 1
arch/x86/boot/Makefile:188: recipe for target 'install' failed
make[1]: *** [install] Error 1
arch/x86/Makefile:260: recipe for target 'install' failed
make: *** [install] Error 2
```

图 2.3 内核安装失败

发现是/boot 空间不够, 于是删去了该目录下旧的内核文件, 再次 make install, 成功。

2) 编写测试文件, 通过系统调用将 test0.c 拷贝到 test1.c.结果如下图 2.4 所示。

```
zdz@zdz-Aspire-E5-572G:~/zdz/os_design/lab2$ ls
makefile  syscall_test  syscall_test.c  test0.c
zdz@zdz-Aspire-E5-572G:~/zdz/os_design/lab2$ ./syscall_test
zdz@zdz-Aspire-E5-572G:~/zdz/os_design/lab2$ cat test0.c test1.c
sdasdadasdadsdasdasdasdasd
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
lllsldlsldslldslldaldasldlasdla
sadasdasdasdsadasdaswidqwodqo
o

qjwlkdjqlk;djl;kdjwldkjaslkdja;ldj
sdasdadasdadsdasdasdasdasd
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
lllsldlsldslldslldaldasldlasdla
sadasdasdasdsadasdaswidqwodqo
o

qjwlkdjqlk;djl;kdjwldkjaslkdja;ldj
```

图 2.4 系统调用测试

## 4.3 实验心得

通过本次实验，掌握了添加系统调用的方法，对 linux 的内核结构有了一个初步的了解。明白了内核态与用户态下的运行权限是不同的，在编写系统调用时要注意保护内核空间，不能再像在用户空间下写程序那样随意。

通过实验，对系统调用程序的大致结构有了一个了解。

同时，通过本次实验也明白了编译内核是一件极为耗时的事情，故在编译内核之前需要尽量确认各步骤无误，系统调用程序能够进行编译，否则会浪费大量时间。

# 设计内容 (3)

## 1 实验目的

(1) 掌握添加设备驱动程序的方法

## 2 实验内容

- 1) 采用模块方法，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

## 3 实验设计

### 3.1 平台环境

操作系统：Ubuntu16.04LTS

内核版本：4.4.4

编译环境：gcc

### 3.2 方案设计

- 1) 首先编写设备驱动程序 `mydev.c`。主要是通过文件位置指针来实现读、写时可以顺着上一次的位置继续读/写。每次读写都是通过设定好的缓冲区来进行，读

写时都要先判断读写指针是否已经到缓冲区头。

## 4 实验调试

### 4.1 实验步骤

- 1) 根据设计方案编写设备驱动程序 mydev.c (详见附录);
- 2) 将设备驱动源文件复制到内核源代码目录/usr/src/linux-4.4.4/drivers/misc 下;
- 3) 修改 makefile 文件, 只需增加: obj-m += mydev.o;
- 4) 在当前目录下进行文件编译:  

```
make -C /usr/src/linux-4.4.4/ SUBDIRS=$PWD modules
```

得到 mydev.ko 文件
- 5) 挂载内核中的模块:  

```
insmod ./mydev.ko
```
- 6) 创建新的虚拟设备文件:  

```
mknod /dev/my_dev c 168 0
```
- 7) 编写测试程序测试新的设备驱动。
- 8) 最后执行卸载操作

删除模块: `rmmmod mydev`

删除新增的设备文件: `rm /dev/mydev`

### 4.2 实验调试及结果

- 1) 编写测试程序, 打开新增的设备后连续进行三次写操作, 再通过三次读操作读取内容看是否能够成功写入和读出。测试结果如下图 3.1 和图 3.2 所示



```
zdz@zdz-Aspire-E5-572G:~/zdz/os_design/lab3$ sudo ./test_mydev
This is 1 line
this is 2 line
this is 3 line
zdz@zdz-Aspire-E5-572G:~/zdz/os_design/lab3$
```

图 3.1 设备驱动测试

```
[18146.868916] Register my_dev to system successfully!
[18247.745057] mydev open successfully!
[18247.745061] write *ppos : 0
[18247.745062] mydev write successfully
[18247.745063] write *ppos : 16
[18247.745064] mydev write successfully
[18247.745064] write *ppos : 32
[18247.745065] mydev write successfully
[18247.745068] mydev open successfully!
[18247.745069] read *ppos : 0
[18247.745070] n = 16
[18247.745071] mydev read successfully
[18247.745114] read *ppos : 16
[18247.745115] n = 16
[18247.745116] mydev read successfully
[18247.745121] read *ppos : 32
[18247.745122] n = 16
[18247.745123] mydev read successfully
[18250.715570] mydev open successfully!
[18250.715574] write *ppos : 0
[18250.715575] mydev write successfully
[18250.715576] write *ppos : 16
[18250.715576] mydev write successfully
[18250.715577] write *ppos : 32
[18250.715577] mydev write successfully
[18250.715579] mydev open successfully!
[18250.715581] read *ppos : 0
[18250.715581] n = 16
[18250.715582] mydev read successfully
[18250.715630] read *ppos : 16
[18250.715631] n = 16
[18250.715644] mydev read successfully
[18250.715648] read *ppos : 32
[18250.715649] n = 16
[18250.715649] mydev read successfully
```

图 3.2 设备驱动测试 dmesg

结果符合预期。

- 2) 在测试的时候发现若不加 sudo 直接运行测试程序是得不到正确的结果的。
- 3) 测试时发现只用了一个文件位置指针，导致写完设备后直接读取的话会使得读不出数据，因为此时文件位置指针已经到达文件内容的末尾。尝试使用两个指针分别进行读、写位置的定位，但是失败了，最终是在写完之后先关闭设备再打

开后进行读的操作。

### 4.3 实验心得

通过本次实验，对字符设备驱动有了初步的了解，对其注册等函数进行了具体的实现，对设备驱动的编写、编译、挂载等流程有了具体的体会。

读写设备意味着要在内核地址空间与用户地址空间之间传递数据，要注意对内核地址空间的保护。结合上一个实验深深地体悟到在内核中编写的系统调用、模块等程序都应该要考虑到内核地址空间的保护。

同时也通过此次实验对 Linux 的内核结构有了更深入的了解。

# 设计内容 (4)

## 1 实验目的

(1) 理解和分析/proc 文件

## 2 实验内容

- 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)

## 3 实验设计

### 3.1 平台环境

操作系统：Ubuntu16.04LTS

内核版本：4.4.4

编译环境：gcc

### 3.2 方案设计

- 1) 主要采用笔记本构件（notebook）来实现各信息（进程信息、cpu 信息、内存



信息)的分页面显示,每个标签页(用 frame 实现)显示对应的信息。由此总的方案便是先创建窗口,然后添加笔记本控件,再在其中添加各标签页信息,每个标签页通过一个函数进行创建,将笔记本构件作为参数传递给每个函数,各函数获得相应信息后生成新标签页并添加到笔记本构件中。

其中,对于进程信息的获取是通过读取/proc 目录下的数字命名的文件夹中 status 文件来获取 NAME、STATE(睡眠、运行等)、PID、RSS(实际占用物理内存)等信息。每个进程的信息获取后添加进分栏列表(clist),一行为一个进程的信息(由于进程数过多,故将分栏列表放入滚动窗 scroll 以显示全部进程)。

进程信息的刷新借鉴设计内容(1)的思路,采用 g\_timeout\_add 函数每隔一秒的时间调用回调函数对整个分栏列表进行一次清除、重写,以此达到自动刷新的目的。

对于 CPU 信息与内存信息,直接通过 label 构件简单地显示相应信息,信息分别从/proc/cpuinfo 与/proc/meminfo 中读取,同样是自动刷新,整体思路与设计内容一类似,在此不多加赘述。

## 4 实验调试

### 4.1 实验步骤

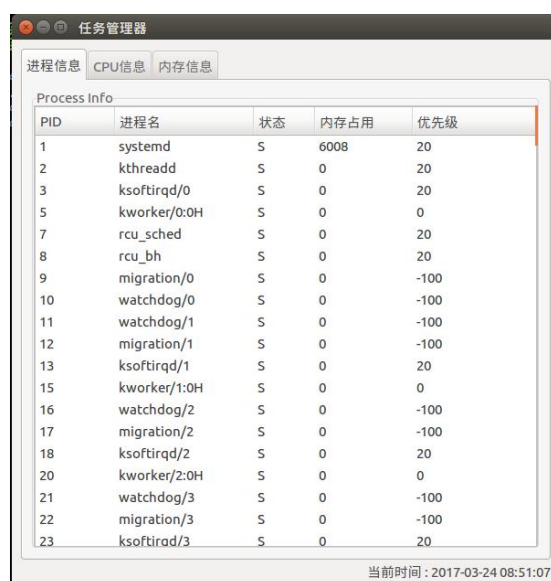
- 1) 完成界面的总体布局,包括主窗口、笔记本构件等的创建;
- 2) 按照设计方案分别完成三个函数用于相应信息的获取、显示与刷新;
- 3) 编译、调试。

## 4.2 实验调试及结果

1) 在进行进程信息的刷新时,发现进程刷新时会发生很严重的频闪,并且过一会儿整个进程就会崩溃。通过上网查阅资料,发现在对分栏列表进行刷新时可以通过 `gtk_clist_freeze` (冻结) 与 `gtk_clist_thaw` (解冻) 的配合使用 (即在刷新前先冻结列表,全部更改完后再解冻显示) 来消除频繁的大量内容修改带来的卡顿、闪烁与崩溃,提高内容更新的效率。

2) 本想在进程这一块增添选择进程并杀死进程的功能,但由于更新进程信息列表时采用的是先全部清除而后进行重新添加信息的方式,每次选完后一旦刷新便丢失了原来的选择,于是考虑通过 `gtk_signal_connect` 为分栏列表的“选择行”这一事件绑定一个回调函数,每次选择一行时就将该进程名记录下来,当更新的时候一旦发现同名的进程便调用 `gtk_clist_select_row` 对此时改进程所在行进行选择。实际测试后发现的确是会选择该行,但在图形界面上并没有标记选择,而且还有一些其他的小问题。由于时间问题,最终放弃了这一功能的实现。

3) 最终结果如下图 4.1、图 4.2 和图 4.3 所示



PID	进程名	状态	内存占用	优先级
1	systemd	S	6008	20
2	kthreadd	S	0	20
3	ksoftirqd/0	S	0	20
5	kworker/0:0H	S	0	0
7	rcu_sched	S	0	20
8	rcu_bh	S	0	20
9	migration/0	S	0	-100
10	watchdog/0	S	0	-100
11	watchdog/1	S	0	-100
12	migration/1	S	0	-100
13	ksoftirqd/1	S	0	20
15	kworker/1:0H	S	0	0
16	watchdog/2	S	0	-100
17	migration/2	S	0	-100
18	ksoftirqd/2	S	0	20
20	kworker/2:0H	S	0	0
21	watchdog/3	S	0	-100
22	migration/3	S	0	-100
23	ksoftirqd/3	S	0	20

图 4.1 进程信息显示



图 4.2 cpu 信息显示



图 4.3 内存信息显示

结果符合预期。

### 4.3 实验心得

通过此次实验，更深入地了解了 GTK 的使用，掌握了笔记本、分栏列表、滚动窗等控件的使用。对于进程在 Linux 文件系统中的存储方式也有了初步的了解，对 Linux 的文件系统的结构有了一个浅显的认识。

尤其是对于进程信息的存放有了一个新的认识，原来存放在以数字命名的文件夹中，各信息存于 status 文件中，存储结构清晰，获取信息方便。

# 设计内容 (5)

## 1 实验目的

(1) 理解和掌握文件系统的设计方法

## 2 实验内容

- 1) 设计、实现一个模拟的文件系统
- 2) 包含文件/目录创建/删除，目录显示等基本功能(可自行扩充文件读/写、用户登录、权限控制、读写保护等其他功能)

## 3 实验设计

### 3.1 平台环境

操作系统: Ubuntu16.04LTS

内核版本: 4.4.4

编译环境: gcc

### 3.2 方案设计

- 1) 总体采用树的结构(为方便,使用数组来实现树,以数组下标为索引构建父子结点之间的联系;

- 2) 每个目录结点下可以挂目录和文件, 每个文件内容存放到几块内存空间(模拟磁盘块)中;
- 3) 当前路径的表示可以直接使用当前目录对应的目录结点在总目录结点数组中的下标, 显示时则从当前路径通过父结点层层返回至根结点, 再从根结点开始层层往下输出目录名称直至当前目录, 由此打印出当前路径;
- 4) 在显示当前目录下的内容时, 首先遍历当前目录结点的子文件结点信息, 然后遍历子目录结点, 通过递归调用, 显示各子目录结点的信息;
- 5) 更改当前目录时可以采用绝对路径也可以采用相对路径, 只需首先判断是否以‘/’开头即可;
- 6) 创建文件和目录时首先从数组中找空闲的结点并返回其下标, 删除时释放对该数组元素的占用(另外使用一个数组进行占用与否标记的存储), 若找不到空闲则创建失败。

## 4 实验调试

### 4.1 实验步骤

- 1) 根据设计方案编写程序并进行调试与测试。

### 4.2 实验调试及结果

- 1) 测试创建和删除目录、创建和删除文件、更改目录、读写文件、显示当前目录下内容的功能, 在此仅列出一种测试的结果图, 该测试首先在根目录下创建文件 file1 与文件 mulu1, 然后进入 mulu1 创建文件 file2, 对文件 file2 进行读写操

作，最后回到根目录进行 ls 操作查看信息。

结果如图 5.1 所示。

```
/mkdir mulu1
create dir 'mulu1' successfully!
/vim file1
create file 'file1' successfully!
/ls
<file>  name: file1      len: 0
<dir>   name: mulu1
1 files
1 dirs
/cd mulu1
/mulu1/vim file2
create file 'file2' successfully!
/mulu1/echo file2 this is file 2
write successfully!
/mulu1/cat file2
this is file 2
/mulu1/cd ..
/ls
<file>  name: file1      len: 0
<dir>   name: mulu1
        <file>  name: file2      len: 14
1 files
1 dirs
/
```

图 5.1 文件系统测试结果图

测试成功。

### 4.3 实验心得

此次实验，总体架构（数据结构的设计）部分是参考的别人的设计思路，该设计使用数组来实现树，极大地方便了编程，操作简单，结构清晰，值得借鉴。缺点是所有数组大小需要事先给定，可扩展性不强，同时也会带来空间不足或浪费空间的现象。

总体上对于文件系统的组织架构有了更深入的了解。

## 附录 实验代码

### 设计内容（1）

#### [dis\_three.c]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
    int pid1,pid2,pid3;
    if ((pid1 = fork()) < 0)
    {
        printf("fork 1 error!\n");
    }
    else if (0 == pid1)
    {
        execv("./dis_time", argv);
    }
    else
    {
        if ((pid2 = fork()) < 0)
        {
            printf("fork 2 error!\n");
        }
        else if (0 == pid2)
        {
            execv("./dis_cpu", argv);
        }
        else
        {
            if ((pid3 = fork()) < 0)
            {
                printf("fork 3 error!\n");
            }
            else if (0 == pid3)
            {
                execv("./dis_sum", argv);
            }
            else
            {
                waitpid(pid1,NULL,0);
                waitpid(pid2,NULL,0);
                waitpid(pid3,NULL,0);
            }
        }
    }
}
return 0;
```



```

}
[dis_time.c]
#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>
#include <time.h>

static char buffer[256];
GtkWidget *label;

static gboolean get_time(GtkWidget *widget)
{
    time_t cur_time;
    struct tm *loc_time;

    cur_time = time(NULL);
    loc_time = localtime(&cur_time);
    sprintf(buffer,"now time : %04d-%02d-%02d %02d:%02d:%02d\n",loc_time->tm_year+1900,
loc_time->tm_mon+1,loc_time->tm_mday,loc_time->tm_hour, loc_time->tm_min, loc_time->tm_sec);
    gtk_label_set_text(GTK_LABEL(label), buffer);
    return TRUE;
}

int main(int argc, char **argv)
{
    gtk_init(&argc,&argv);

    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    label = gtk_label_new("time");
    g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL);
    // GtkWidget *box = gtk_vbox_new(TRUE,100);

    // gtk_container_add(GTK_CONTAINER(window),box);

    gtk_window_set_title(GTK_WINDOW(window),"Now Time");
    gtk_widget_set_usize(window, 300, 300); // 设置窗口大小

    gtk_container_add(GTK_CONTAINER(window),label);
    g_timeout_add(1000, (GSourceFunc) get_time, (gpointer) window);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}

```

```

[dis_sum.c]
#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>
#include <time.h>

static char buffer[256];
GtkWidget *label;

static gboolean get_sum(GtkWidget *widget)
{
    static int sum = 0;

```

```

static int i = 1;

if (i <= 100)
{
    sum = sum+i;
    i++;
    sprintf(buffer,"累加和: %d",sum);
    gtk_label_set_text(GTK_LABEL(label), buffer);
}
return TRUE;
}

int main(int argc, char **argv)
{
    gtk_init(&argc,&argv);

    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    label = gtk_label_new("sum");
    g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL);

    gtk_window_set_title(GTK_WINDOW(window),"Sum");
    gtk_widget_set_usize(window, 300, 300); // 设置窗口大小

    gtk_container_add(GTK_CONTAINER(window),label);
    g_timeout_add(3000, (GSourceFunc) get_sum, (gpointer) window);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}

```

### **[dis\_cpu.c]**

```

#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>
#include <time.h>

static char buffer[256];
GtkWidget *label;

static gboolean get_cpu(GtkWidget *widget)
{
    FILE *fp;
    char buf[128];
    char name[5];
    long int user,nice,system,idle;
    float ratio;
    fp = fopen("/proc/stat","r");
    fgets(buf,sizeof(buf),fp);
    sscanf(buf,"%s%ld%ld%ld%ld",name,&user,&nice,&system,&idle);
    ratio = 100*(float)(user+nice+system)/(float)(user+nice+system+idle);
    sprintf(buffer,"cpu 使用率: %%%.6f",ratio);
    gtk_label_set_text(GTK_LABEL(label), buffer);
    return TRUE;
}

int main(int argc, char **argv)

```

```

{
    gtk_init(&argc,&argv);

    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    label = gtk_label_new("cpu");
    g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL);

    gtk_window_set_title(GTK_WINDOW(window),"Cpu");
    gtk_widget_set_usize(window, 300, 300); // 设置窗口大小

    gtk_container_add(GTK_CONTAINER(window),label);
    g_timeout_add(3000, (GSourceFunc) get_cpu, (gpointer) window);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}

```

### **[makefile]**

CC = gcc

lab1:

```

    ${CC} -o dis_time dis_time.c `pkg-config --cflags --libs gtk+-2.0`
    ${CC} -o dis_cpu dis_cpu.c `pkg-config --cflags --libs gtk+-2.0`
    ${CC} -o dis_sum dis_sum.c `pkg-config --cflags --libs gtk+-2.0`
    ${CC} -o dis_three dis_three.c

```

clean:

```
rm -f *.o
```

rebuild: clean

## 设计内容（2）

### **[系统调用函数定义]**

asm linkage long sys\_mycopy(const char \*src\_file, const char \*tar\_file)

```

{
    int infd,outfd,count;
    char buf[256];
    mm_segment_t fs;
    fs = get_fs(); // 取得当前的地址访问限制值
    set_fs(get_ds()); // 获得 kernel 的内存访问地址范围
    if ((infd = sys_open(src_file, O_RDONLY, 0)) == -1)
    {
        return 1;
    }
    if ((outfd=sys_open(tar_file, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) == -1)
    {
        return 2;
    }
    while ((count = sys_read(infd, buf, 256)) > 0)
    {
        if (sys_write(outfd, buf, count) != count)

```

```

        return 3;
    }
    if (-1 == count)
        return 4;
    sys_close(infd);
    sys_close(outfd);
    set_fs(fs); // 恢复原来的值
    return 0;
}

```

## 设计内容（3）

### [mydev.c]

```

#include <linux/fs.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/module.h>
#include <linux/device.h>
#include <asm/uaccess.h>

#define MAJOR_NUM 168

int BUSY_SIG = 0;

struct mydev
{
    int len;
    char buffer[50];
    struct cdev cdev;
    loff_t *r_ppos;
}my_dev0;

static ssize_t mydev_open(struct inode *inode,struct file *file)
{
    struct mydev *my_dev;
    /*if (BUSY_SIG)
    {
        return EBUSY;
    }
    BUSY_SIG++;*/
    my_dev0.r_ppos = 0;
    printk("mydev open successfully!\n");
    my_dev = container_of(inode->i_cdev,struct mydev, cdev);
    file->private_data = my_dev;

    return 0;
}

static ssize_t mydev_release(struct inode *inode, struct file *file)
{
    /*if (!BUSY_SIG)
    {
        return -1;
    }

```

```

        BUSY_SIG = 0;*/
        return 0;
    }

static ssize_t mydev_read(struct file *file, char __user *buf_usr, size_t size, loff_t *ppos)
{
    int n, ret;
    char *buf_ker;

    struct mydev *my_dev = file->private_data;

    printk("read *ppos : %lld\n", *ppos);

    if (*ppos == my_dev->len)
    {
        return 0;
    }

    if (size > (my_dev->len - *ppos))
    {
        n = my_dev->len - *ppos;
    }
    else
    {
        n = size;
    }

    printk("n = %d\n", n);

    buf_ker = my_dev->buffer + *ppos;

    ret = copy_to_user(buf_usr, buf_ker, n);
    if (ret != 0)
    {
        return EFAULT;
    }

    *ppos += n;

    printk("mydev read successfully\n");
    // return n;
}

static ssize_t mydev_write(struct file *file, const char __user *buf_usr, size_t size, loff_t *ppos)
{
    int n, ret;
    char *buf_ker;
    struct mydev *my_dev = file->private_data;

    printk("write *ppos : %lld\n", *ppos);

    // 已经到 buf 尾部
    if (*ppos == sizeof(my_dev->buffer))
    {
        return -1;
    }

    if (size > (sizeof(my_dev->buffer) - *ppos))
    {
        n = sizeof(my_dev->buffer) - *ppos;
    }

```

```

    }
    else
    {
        n = size;
    }

    buf_ker = my_dev->buffer + *ppos;

    ret = copy_from_user(buf_ker, buf_usr, n);
    if (ret != 0)
    {
        return EFAULT;
    }
    // 更新文件位置指针的值
    *ppos += n;

    // 更新 my_dev.len
    my_dev->len += n;
    printk("mydev write successfully\n");
}

static const struct file_operations mydev_fops = {
    .owner = THIS_MODULE,
    .read = mydev_read,
    .write = mydev_write,
    // .ioctl = mydev_ioctl,
    .open = mydev_open,
    .release = mydev_release
};

int __init mydev_init(void)
{
    int ret;
    dev_t dev_num;

    // 初始化字符设备
    cdev_init(&my_dev0.cdev, &mydev_fops);
    // 设备号(主设备号 12bit, 次设备号 20bit)
    dev_num = MKDEV(MAJOR_NUM,0);

    // 注册设备号
    ret = register_chrdev_region(dev_num,1,"my_dev");
    if (ret < 0) // 若静态注册失败则改用动态注册
    {
        ret = alloc_chrdev_region(&dev_num, 0, 1, "my_dev");
        if (ret < 0)
        {
            printk("Fail to register_chrdev_region!\n");
            return EIO;
        }
    }

    // 添加设备到操作系统
    ret = cdev_add(&my_dev0.cdev, dev_num, 1);
    if (ret < 0)
    {
        printk("Fail to cdev_add");
        goto unregister_chrdev;
    }
}

```

```

    }

    printk("Register my_dev to system successfully!\n");
    return 0;

    unregister_chrdev:
    unregister_chrdev_region(dev_num, 1);
}

void __exit mydev_exit(void)
{
    // 从系统中删除设备的字符设备
    cdev_del(&my_dev0.cdev);
    // 释放设备申请号
    unregister_chrdev_region(MKDEV(MAJOR_NUM,0), 1);
    printk("Exit my_dev successfully!\n");
    return ;
}

module_init(mydev_init);
module_exit(mydev_exit);

```

## 设计内容（4）

### [System\_Explorer.c]

```

#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>
#include <math.h>

#define MAX_PROC 100// 能显示的最大进程数量
#define PATH_LEN_MAX 50 // 路径名最大长度
#define NAME_LEN_MAX 256 // status 中项名最大长度
// status 中各信息行
#define LINE_NAME 1
#define LINE_STATE 2
#define LINE_PID 5
#define LINE_RSS 21

GtkWidget *window; // 主窗口
GtkWidget *time_label; // 时间显示

static gchar temp_sel_name[256] = {0};
static gchar *old_sel_name = NULL;

typedef struct _proc_info_st{
    char name[256];
    char state[20];

```

```

    char pid[20];
    char rss[20];
    char priority[20];
}proc_info;

void CreateMenuBar(GtkWidget *vbox);
void Add_notebook(GtkWidget *vbox);
static gboolean get_time(GtkWidget *widget);
void CreateProPage(GtkWidget *notebook);
void CreateCpuPage(GtkWidget *notebook);
void CreateMemPage(GtkWidget *notebook);
static gboolean pro_ref(GtkWidget *clist);
static gboolean cpu_ref(GtkWidget *cpu_label);
static gboolean mem_ref(GtkWidget *mem_label);
void ReadLine(FILE *fp, char *buff, int buflen, int line);
void sel_func(GtkWidget *clist, gint row, gint column, GdkEventButton *event, gpointer data);

int main(int argc, char **argv)
{
    GtkWidget *vscrollbar, *hscrollbar; // 定义滚动条
    GtkWidget *main_vbox;
    GtkWidget *status_bar, *menu_bar;
    GtkWidget *file_menu;
    GtkWidget *notebook;

    gtk_init(&argc, &argv); // 初始化在任何构件生成之前完成

    // 构建主窗口
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL); // 创建主窗口
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL); // 设置退出
响应
    gtk_window_set_title(GTK_WINDOW(window), "任务管理器"); // 设置窗口标题
    gtk_widget_set_usize(window, 600, 600); // 设置窗口大小
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER); //
窗口居中显示
    gtk_window_set_resizable(GTK_WINDOW(window), TRUE); // 设置窗口大小可改变
    gtk_container_set_border_width(GTK_CONTAINER(window), 5); // 设置窗口边框宽度
    gtk_widget_show(window);

    main_vbox = gtk_vbox_new(FALSE, 1); // 不强制使用相同大小, 构件间隔 1 像素
    gtk_container_set_border_width(GTK_CONTAINER(main_vbox), 5);
    gtk_container_add(GTK_CONTAINER(window), main_vbox);
    gtk_widget_show(main_vbox);

    // 添加笔记本构件
    notebook = gtk_notebook_new();
    gtk_box_pack_start(GTK_BOX(main_vbox), notebook, TRUE, TRUE, 0);
    gtk_widget_show(notebook);
    CreateProPage(notebook);
    CreateCpuPage(notebook);
    CreateMemPage(notebook);

    // 底部显示时间
    time_label = gtk_label_new("当前时间");
    gtk_box_pack_end(GTK_BOX(main_vbox), time_label, FALSE, FALSE, 0);

```



```

g_timeout_add(1000, (GSourceFunc)get_time, (gpointer>window);

gtk_widget_show_all(window);
gtk_main();

}

/*void sel_func(GtkWidget *clist, gint row, gint column, GdkEventButton *event, gpointer data)
{
    printf("****\n");
    gtk_clist_get_text(GTK_CLIST(clist), row, 1, &old_sel_name);
    strcpy(temp_sel_name, old_sel_name);
    printf("%s\n", temp_sel_name);
}*/

void CreateProPage(GtkWidget *notebook)
{
    GtkWidget *frame = gtk_frame_new("Process Info");
    gchar *col_name[6] = {"PID", "进程名", "状态", "内存占用", "优先级"}; // 分栏列表列名
    GtkWidget *clist = gtk_clist_new_with_titles(5, col_name); // 创建分栏列表
    GtkWidget *scroll = gtk_scrolled_window_new(NULL, NULL); // 创建滚动窗

    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scroll),
    GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
    // gtk_signal_connect(GTK_OBJECT(clist), "select_row", GTK_SIGNAL_FUNC(sel_func),
    NULL); // 选中行时做处理

    gtk_clist_set_column_width(GTK_CLIST(clist), 0, 80);
    gtk_clist_set_column_width(GTK_CLIST(clist), 1, 145);
    gtk_clist_set_column_width(GTK_CLIST(clist), 2, 60);
    gtk_clist_set_column_width(GTK_CLIST(clist), 3, 100);
    gtk_clist_set_column_width(GTK_CLIST(clist), 4, 100);

    gtk_scrolled_window_add_with_viewport(GTK_SCROLLED_WINDOW(scroll), clist); // 分栏
    列表放入滚动窗

    g_timeout_add(1000, (GSourceFunc)pro_ref, (gpointer)clist);

    gtk_container_set_border_width(GTK_CONTAINER(frame), 10); // frame 设置
    gtk_widget_set_size_request(frame, 100, 335);
    gtk_container_add(GTK_CONTAINER(frame), scroll);

    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), frame, gtk_label_new("进程信息"));

}

static gboolean pro_ref(GtkWidget *clist)
{
    int i;
    DIR *dir = NULL;
    FILE *fp = NULL;
    struct dirent *ptr;
    char path[PATH_LEN_MAX];
    char name[NAME_LEN_MAX];
    proc_info pro_inf;
    char file[512] = {0};
    char buff[1024] = {0};
    //int cur_row = 0;

```



```

        fclose(fp);

        gchar *list[1][6] = {{pro_inf.pid, pro_inf.name, pro_inf.state, pro_inf.rss,
pro_inf.priority}};
        gtk_clist_append((GtkCList*) clist, list[0]);
    }
}
gtk_clist_thaw((GtkCList *) clist);
closedir(dir);
return TRUE;
}

```

```

void GetInfo(char *path, char *name, char info[])
{
    int fd = open(path, O_RDONLY);
    char temp[1000];
    int k = 0;
    read(fd, temp, sizeof(temp));
    close(fd);
    char *p = NULL;
    p = strstr(temp, name);
    while(*p != ':')
        p++;
    //p = p + 2;
    p++;
    while(*p == ' ')
        p++;
    while(*p != '\n')
    {
        info[k] = *p;
        p++;
        k++;
    }
    info[k] = '\n';
    info[k+1] = '\0';
}

```

```

static gboolean cpu_ref(GtkWidget *label)
{

```

```

    FILE *fp;
    char buf[128];
    char name[5];
    char cpuBuffer[2000];
    char buffer[50];
    long int user,nice,system,idle;
    float ratio;
    fp = fopen("/proc/stat","r");
    fgets(buf,sizeof(buf),fp);
    sscanf(buf,"%s%ld%ld%ld%ld",name,&user,&nice,&system,&idle);
    ratio = 100*(float)(user+nice+system)/(float)(user+nice+system+idle);
    sprintf(buffer,"cpu 使用率: %%%.6f",ratio);

```

```

    char modeName[50], cpuMHz[20], cacheSize[20], cpuCores[20], addrSize[50];
    char *p;
    GetInfo("/proc/cpuinfo", "model name", modeName);
    GetInfo("/proc/cpuinfo", "cache size", cacheSize);

```

```

GetInfo("/proc/cpuinfo", "cpu MHz", cpuMHz);
GetInfo("/proc/cpuinfo", "cpu cores", cpuCores);
GetInfo("/proc/cpuinfo", "address sizes", addrSize);
strcpy(cpuBuffer, "\nCPU 型号和主频:\n");
p = strtok(modeName, "\n");
strcat(cpuBuffer, p);
strcat(cpuBuffer, "\n\n 寻址位数:");
p = strtok(addrSize, "\n");
strcat(cpuBuffer, p);
strcat(cpuBuffer, "\n\nncpu 主频:");
p = strtok(cpuMHz, "\n");
strcat(cpuBuffer, p);
strcat(cpuBuffer, " MHz");
strcat(cpuBuffer, "\n\nCache 大小:");
p = strtok(cacheSize, "\n");
strcat(cpuBuffer, p);
strcat(cpuBuffer, "\nncpu 核数:");
p = strtok(cpuCores, "\n");
strcat(cpuBuffer, p);

strcat(cpuBuffer, "\n");
strcat(cpuBuffer, buffer);

gtk_label_set_text(GTK_LABEL(label), cpuBuffer);
return TRUE;
}

static gboolean mem_ref(GtkWidget *mem_label)
{
    char buffer0[120];
    char data[30];

    long total = 0, free = 0;
    int counter = 0;
    int fd;
    int i = 0;

    char *buffer;

    fd = open("/proc/meminfo", O_RDONLY);
    read(fd, buffer0, 100);
    buffer = buffer0;

    while (1)
    {
        if (':' == buffer[i])
        {
            counter++;
            i += 2;
            buffer += i;
            i = 0;
            while ('k' != buffer[i])
            {
                i++;
            }
            buffer[i] = '\0';
            if (1 == counter)
            {
                total = atol(buffer)/1024;
            }
        }
    }
}

```

```

        i += 3;
        buffer += i;
        i = 0;
    }
    else if (2 == counter)
    {
        free = atol(buffer)/1024;
        break;
    }

    }
    else
    {
        i++;
    }
}

sprintf(data,"MemTotal:%ldM\nMemFree:%ldM",total,free);
close(fd);

gtk_label_set_text(GTK_LABEL(mem_label), data);

return TRUE;
}

void CreateCpuPage(GtkWidget *notebook)
{
    GtkWidget *frame = gtk_frame_new("CPU Info");
    gtk_container_set_border_width(GTK_CONTAINER(frame), 10);
    gtk_widget_set_size_request(frame, 100, 335);
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), frame, gtk_label_new("CPU 信息
"));

    GtkWidget *cpu_label = gtk_label_new("cpu");
    g_timeout_add(1000, (GSourceFunc)cpu_ref, (gpointer)cpu_label);
    gtk_container_add(GTK_CONTAINER(frame), cpu_label);

}

void CreateMemPage(GtkWidget *notebook)
{
    GtkWidget *frame = gtk_frame_new("Memory Info");
    gtk_container_set_border_width(GTK_CONTAINER(frame), 10);
    gtk_widget_set_size_request(frame, 100, 335);
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), frame, gtk_label_new("内存信息"));

    GtkWidget *mem_label = gtk_label_new("mem");
    g_timeout_add(1000, (GSourceFunc)mem_ref, (gpointer)mem_label);
    gtk_container_add(GTK_CONTAINER(frame), mem_label);

}

static gboolean get_time(GtkWidget *widget)
{
    time_t cur_time;
    struct tm *loc_time;
    static char buffer[256];
    cur_time = time(NULL);
    loc_time = localtime(&cur_time);

```

```

        sprintf(buffer,"当前时间 : %04d-%02d-%02d %02d:%02d:%02d\n",loc_time->tm_year+1900,
loc_time->tm_mon+1,loc_time->tm_mday,loc_time->tm_hour, loc_time->tm_min, loc_time->tm_sec);
        gtk_widget_set_size_request(time_label, 200, 20);
        //gtk_label_set_justify(GTK_LABEL(time_label), GTK_JUSTIFY_RIGHT); // 设置对齐方式
(试了无效)
        gtk_misc_set_alignment(GTK_MISC(time_label), 1, 0); // 设置对齐(左右、上下)
        gtk_label_set_text(GTK_LABEL(time_label), buffer);

        return TRUE;
}

```

## 设计内容（5）

### [file\_system.c]

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NAME_LEN_MAX 32          // 目录/文件名最大长度
#define PATH_LEN_MAX 512        // 路径名最大长度
#define MAX_CHILD_DIR 8         // 最大子目录数
#define MAX_CHILD_FILE 16       // 最大子文件数
#define DIR_NUM 50              // 最大目录数
#define FILE_NUM 100            // 最大文件数
#define BLOCK_NUM FILE_NUM// 磁盘块数

// 目录节点结构
struct dir_node {
    char dir_name[NAME_LEN_MAX]; // 目录名
    int child_dir[MAX_CHILD_DIR]; // 子目录索引
    int dir_count;                // 当前子目录数
    int child_file[MAX_CHILD_FILE]; // 子文件索引
    int file_count;              // 当前子文件数
    int parent;                  // 父目录索引
}dir[DIR_NUM];

// 文件节点结构
struct file_node {
    char file_name[NAME_LEN_MAX]; // 文件名
    int file_len;                  // 文件长度
    int parent;                    // 父目录索引
    int block_index;              // 磁盘块索引
}file[FILE_NUM];

// 磁盘块
struct block {
    char buf[1024];
}block[BLOCK_NUM];

```

```

// 目录、文件占用标志位(0 闲 1 占)
int dir_occ[DIR_NUM];
int file_occ[FILE_NUM];
int block_occ[BLOCK_NUM];

// 当前目录
int cur_dir;

int create_file(int parent, char *file_name);
int get_index(int type);
int search(int parent, char *name, int type);
int make_dir(int parent, char *dir_name);
void show_path(int index);
int del_file(int parent, char *file_name);
int del_dir(int parent, char *dir_name);
int change_dir(char *path);
void list_file(int index, int level);
void list_dir(int index, int level);
void list(int index);
int get_dir(char *path);

/**
 * @brief 读文件
 * @arg      parent: int 型, 父目录索引
 * @arg      file_name: char *, 文件名
 * @retval int
 * @return 如果成功则返回文件索引号, 否则返回-1
 */
int read_file(int parent, char *file_name)
{
    int index;

    // 按名查找文件
    if (-1 == (index = search(parent, file_name, 1))) // 文件不存在
    {
        printf("fail to read file '%s' : file doesn't exist!\n", file_name);
        return -1;
    }
    // 读磁盘内容并打印
    printf("%s\n", block[file[index].block_index].buf);
    return index;
}

/**
 * @brief 写文件
 * @arg      parent: int 型, 父目录索引
 * @arg      file_name: char *, 文件名
 * @arg      text: char *, 写入内容
 * @retval int
 * @return 如果成功则返回文件索引号, 否则返回-1
 */
int write_file(int parent, char *file_name, char *text)
{
    int index;

    // 按名查找文件

```

```

    if (-1 == (index = search(parent, file_name, 1))) // 文件不存在
    {
        printf("fail to write file '%s' : file doesn't exist!\n", file_name);
        return -1;
    }

    // 写入
    strcat(block[file[index].block_index].buf, text);
    file[index].file_len += strlen(text);
    printf("write successfully!\n");
    return index;
}

/**
 * @brief 创建文件
 * @arg      parent: int 型, 父目录索引
 * @arg      file_name: char *, 文件名
 * @retval int
 * @return 如果成功则返回文件索引号, 否则返回-1
 */
int create_file(int parent, char *file_name)
{
    int index;
    int b_index;
    if (dir[parent].file_count == MAX_CHILD_FILE) // 父目录已满
    {
        printf("fail to create file '%s' : parent dir is full!\n", file_name);
        return -1;
    }
    else if (search(parent, file_name, 1) != -1) // 同名文件存在
    {
        printf("fail to create file '%s' : file already exists!\n", file_name);
        return -1;
    }

    if (-1 == (index = get_index(1))) // 总文件数超出
    {
        printf("fail to create file '%s' : file is too much!\n", file_name);
        return -1;
    }
    if (-1 == (b_index = get_index(2))) // 磁盘块不够
    {
        printf("fail to create file '%s' : blocks are not enough !\n", file_name);
        return -1;
    }

    // 创建文件
    strcpy(file[index].file_name, file_name);
    file[index].parent = parent;
    file[index].file_len = 0;

    file[index].block_index = b_index;

    dir[parent].child_file[dir[parent].file_count++] = index;

    return index;
}

```



```

}

/**
 * @brief 获得索引号
 * @arg      type: int 型, 查找类型 (0: 目录, 1: 文件, 2: 磁盘块)
 * @retval int
 * @return 如果成功则返回索引号, 否则返回-1
 */
int get_index(int type)
{
    int i = 0;
    if (1 == type) // 文件
    {
        for (i = 0; i < FILE_NUM; i++)
        {
            if (0 == file_occ[i])
            {
                file_occ[i] = 1;
                return i;
            }
        }
        return -1;
    }
    else if (0 == type) // 目录
    {
        for (i = 1; i < DIR_NUM; i++)
        {
            if (0 == dir_occ[i])
            {
                dir_occ[i] = 1;
                return i;
            }
        }
        return -1;
    }
    else
    {
        for (i = 0; i < BLOCK_NUM; i++)
        {
            if (0 == block_occ[i])
            {
                block_occ[i] = 1;
                return i;
            }
        }
        return -1;
    }
}

/**
 * @brief 获得索引号
 * @arg      parent: int 型, 父目录索引号
 * @arg      name: char *型, 名
 * @arg      type: int 型, 查找类型 (0: 目录, 1: 文件)
 * @retval int
 * @return 如果成功则返回索引号, 否则返回-1
 */

```

```

int search(int parent, char *name, int type)
{
    int i = 0;
    if (type) // 文件
    {
        for (i = 0; i < dir[parent].file_count; i++)
        {
            if (!strcmp(file[dir[parent].child_file[i]].file_name, name))
            {
                return dir[parent].child_file[i];
            }
        }
        return -1;
    }
    else
    {
        for (i = 0; i < dir[parent].dir_count; i++)
        {
            if (!strcmp(dir[dir[parent].child_dir[i]].dir_name, name))
            {
                return dir[parent].child_dir[i];
            }
        }
        return -1;
    }
}

/**
 * @brief 创建目录
 * @arg    parent: int 型, 父目录索引
 * @arg    dir_name: char *, 目录名
 * @retval int
 * @return 如果成功则返回目录索引号, 否则返回-1
 */
int make_dir(int parent, char *dir_name)
{
    int index;
    if (dir[parent].dir_count == MAX_CHILD_DIR) // 父目录已满
    {
        printf("fail to create dir '%s' : parent dir is full!\n", dir_name);
        return -1;
    }
    else if (search(parent, dir_name, 0) != -1) // 同名目录存在
    {
        printf("fail to create dir '%s' : dir already exists!\n", dir_name);
        return -1;
    }

    if (-1 == (index = get_index(0))) // 总目录数超出
    {
        printf("fail to create dir '%s' : dir is too much!\n", dir_name);
        return -1;
    }

    // 创建目录
    strcpy(dir[index].dir_name, dir_name);
    dir[index].dir_count = 0;
    dir[index].file_count = 0;
}

```

```

    dir[index].parent = parent;

    dir[parent].child_dir[dir[parent].dir_count++] = index;
    return index;
}

/**
 * @brief 显示当前路径
 * @arg      index: 当前目录
 * @retval void
 */
void show_path(int index)
{
    if (0 == index)
    {
        printf("/");
    }
    else
    {
        show_path(dir[index].parent);
        printf("%s/", dir[index].dir_name);
    }
}

/**
 * @brief 删除文件
 * @arg      parent: int 型, 父目录索引
 * @arg      file_name: char *, 文件名
 * @retval int
 * @return 如果成功则返回文件索引号, 否则返回-1
 */
int del_file(int parent, char *file_name)
{
    int index;
    int i;
    // 确认文件是否存在
    if (-1 == (index = search(parent, file_name, 1)))    // 文件不存在
    {
        printf("fail to delete file '%s' : file doesn't exist!\n", file_name);
        return -1;
    }
    // 修改占用标志位
    file_occ[index] = 0;
    // 修改父目录结点
    for (i = 0; i < dir[parent].file_count; i++)
    {
        if (dir[parent].child_file[i] == index)
        {
            for (; i < dir[parent].file_count-1; i++)
            {
                dir[parent].child_file[i] = dir[parent].child_file[i+1];
            }
            break;
        }
    }
    dir[parent].file_count--;
    // 释放磁盘

```

```

        block_occ[file[index].block_index] = 0;
        strcpy(block[file[index].block_index].buf, "\\0");
        file[index].file_len = 0;

        return index;
    }

/**
 * @brief 删除目录
 * @arg      parent: int 型, 父目录索引
 * @arg      dir_name: char *, 目录名
 * @retval int
 * @return 如果成功则 返回目录索引号, 否则返回-1
 */
int del_dir(int parent, char *dir_name)
{
    int index;
    int i = 0;
    // 确认目录是否存在
    if (-1 == (index = search(parent, dir_name, 0))) // 目录不存在
    {
        printf("fail to delete dir '%s' : dir doesn't exist!\n", dir_name);
        return -1;
    }
    // 修改占用标志位
    dir_occ[index] = 0;
    printf("***%d\n", index);
    // 修改父目录结点
    for (i = 0; i < dir[parent].dir_count; i++)
    {
        if (dir[parent].child_dir[i] == index)
        {
            printf("***%d\n", index);
            for (; i < dir[parent].dir_count-1; i++)
            {
                dir[parent].child_dir[i] = dir[parent].child_dir[i+1];
            }
            break;
        }
    }
    dir[parent].dir_count--;

    return index;
}

/**
 * @brief 由路径获取目录索引
 * @arg      path: char *型, 路径
 * @retval int
 * @return 成功则返回目录索引, 否则返回-1
 */
int get_dir(char *path)
{
    int i = 0;
    int temp_dir = cur_dir;
    if (path[0] == '/') // 绝对路径
    {

```

```

        temp_dir = 0;
        path++;
    }
    while (1)
    {
        if ('/' == path[i])
        {
            path[i] = '\0';
            if (-1 == (temp_dir = search(temp_dir, path, 0)))
            {
                return -1;
            }
            i++;
            path += i;
            i = 0;
        }
        else if ('\0' == path[i])
        {
            return search(temp_dir, path, 0);
        }
        else
        {
            i++;
        }
    }
    return temp_dir;
}

```

```

/**
 * @brief 改变目录
 * @arg    path: char *型, 路径
 * @retval int
 * @return 成功则返回目录索引, 否则返回-1
 */

```

```

int change_dir(char *path)
{
    int temp_dir;
    if (!strcmp(path, "/"))
    {
        temp_dir = 0;
    }
    else if (!strcmp(path, ".."))
    {
        if (cur_dir == 0)
        {
            temp_dir = -1;
        }
        else
        {
            temp_dir = dir[cur_dir].parent;
        }
    }
    else if (!strcmp(path, "."))
    {
        temp_dir = cur_dir;
    }
    else
    {
        temp_dir = get_dir(path);
    }
}

```

```

    }

    if (-1 == temp_dir)
    {
        printf("dir not exist!\n");
        return -1;
    }
    else
    {
        return (cur_dir = temp_dir);
    }
}

/**
 * @brief 显示文件
 * @arg      index: int 型, 当前目录索引
 * @arg      level: int 型, 当前层数
 * @retval void
 */
void list_file(int index, int level)
{
    int i = 0;
    int l = level;
    for (; l > 0; l--)
    {
        printf(" ");
    }
    printf("<file>  name: %s\tlen: %d\n", file[index].file_name, file[index].file_len);
}

/**
 * @brief 显示目录
 * @arg      index: int 型, 当前文件索引
 * @arg      level: int 型, 当前层数
 * @retval void
 */
void list_dir(int index, int level)
{
    int i = 0;
    int l = level;
    for (; l > 0; l--)
    {
        printf(" ");
    }
    printf("<dir>  name: %s\n", dir[index].dir_name);
    for (i = 0; i < dir[index].file_count; i++)
    {
        list_file(dir[index].child_file[i], level+4);
    }
    for (i = 0; i < dir[index].dir_count; i++)
    {
        list_dir(dir[index].child_dir[i], level+4);
    }
}

/**
 * @brief 显示当前目录

```

```

* @arg      index: int 型, 当前目录索引
* @retval void
*/
void list(int index)
{
    int i = 0;
    for (i = 0; i < dir[index].file_count; i++)
    {
        list_file(dir[index].child_file[i], 0);
    }
    for (i = 0; i < dir[index].dir_count; i++)
    {
        list_dir(dir[index].child_dir[i], 0);
    }

    // 统计文件和目录数
    printf("%d files\n", dir[index].file_count);
    printf("%d dirs\n", dir[index].dir_count);
}

int main(int argc, char **argv)
{
    char command[10];
    char name_in[PATH_LEN_MAX];
    char text[1024];
    while (1)
    {
        show_path(cur_dir);
        scanf("%s", command);
        if (!strcmp(command, "mkdir"))
        {
            scanf("%s", name_in);
            if (-1 != make_dir(cur_dir, name_in))
            {
                printf("create dir '%s' successfully!\n", name_in);
            }
        }
        else if (!strcmp(command, "rmdir"))
        {
            scanf("%s", name_in);
            if (-1 != del_dir(cur_dir, name_in))
            {
                printf("delete dir '%s' successfully!\n", name_in);
            }
        }
        else if (!strcmp(command, "vim"))
        {
            scanf("%s", name_in);
            if (-1 != create_file(cur_dir, name_in))
            {
                printf("create file '%s' successfully!\n", name_in);
            }
        }
        else if (!strcmp(command, "rm"))
        {
            scanf("%s", name_in);
            if (-1 != del_file(cur_dir, name_in))
            {
                printf("delete file '%s' successfully!\n", name_in);
            }
        }
    }
}

```

```

    }
    else if (!strcmp(command, "cat"))
    {
        scanf("%s", name_in);
        read_file(cur_dir, name_in);
    }
    else if (!strcmp(command, "echo"))
    {
        scanf("%s", name_in);
        getchar();
        gets(text);
        write_file(cur_dir, name_in, text);
    }
    else if (!strcmp(command, "cd"))
    {
        scanf("%s", name_in);
        change_dir(name_in);
    }
    else if (!strcmp(command, "ls"))
    {
        list(cur_dir);
    }
    else if (!strcmp(command, "exit"))
    {
        break;
    }
    else
    {
        printf("wrong command!\n");
    }
}

return 0;
}

```