

华中科技大学

课程实验报告

课程名称： 操作系统原理

专业班级： CS1002

学 号： U2101014265

姓 名： 李辉

指导教师： 阳富民

报告日期： 2013.03.25

计算机科学与技术学院

目录

| | |
|----------------------|----|
| 1 课程实验概述 | 3 |
| 2 实验一 linux 编程 | 3 |
| 2.1 实验目的与内容 | 3 |
| 2.2 实验原理与方案 | 3 |
| 2.3 实验测试与结果 | 3 |
| 3 实验二 系统调用 | 4 |
| 3.1 实验目的与内容 | 4 |
| 3.2 实验原理与方案 | 5 |
| 3.3 实验测试与结果 | 5 |
| 4 实验三 字符设备驱动 | 6 |
| 4.1 实验目的与内容 | 6 |
| 4.2 实验原理与方案 | 6 |
| 4.3 实验测试与结果 | 6 |
| 5 实验四 系统监视器 | 7 |
| 5.1 实验目的与内容 | 7 |
| 5.2 实验原理与方案 | 7 |
| 5.3 实验测试与结果 | 7 |
| 6 实验总结与评价 | 10 |
| 7 参考文献 | 11 |
| 8 附录 | 12 |

1 课程实验概述

操作系统课程设计的主要目的是让学生掌握 Linux 操作系统的使用方法，了解 Linux 系统内核代码结构，掌握实例操作系统的实现方法。

2 实验一 linux 编程

2.1 实验目的与内容

实验目的：掌握 Linux 操作系统的使用方法，包括键盘命令、系统调用；掌握在 Linux 下的编程环境；

实验内容：

1. 编一个 C 程序，其内容为实现文件拷贝的功能；
2. 编一个 C 程序，其内容为分窗口同时显示三个并发进程的运行结果，要求用到 Linux 下的图形库。

2.2 实验原理与方案

1. 第一小题有很多方法可以实现这个功能，其核心就是打开文件，然后读文件，最后写入即可，可以选用 C 语言的函数，也可以调用 Linux 系统函数。

2. 三个并发程序的实现比较简单，可以使用 fork 系统调用运行三个程序，实验时编写的窗口是基于 GTK 图形库，可以显示一串字符，也可以实现数字自增的功能。

2.3 实验测试与结果

实验 1-1 代码生成的程序运行结果如图 2-1 所示，可以看出源文件与目标文件之间没有差异，内容也是一致的；实验 1-2 代码生成的程序运行结果如图 2-2 所示，程序运行时会产生三个窗口，它们在时间上是并行的。

综合上面的叙述，实验 1 的两个程序测试结果与预期一致，说明代码编写正确。

```

leah@leah-K42JZ:~/Desktop/Documents/1$ ./1-1 ./Makefile ./hello
leah@leah-K42JZ:~/Desktop/Documents/1$ diff ./Makefile ./hello
leah@leah-K42JZ:~/Desktop/Documents/1$ cat ./hello
CC=gcc
OBS=1-1 1-2
all: $(OBS)
$(OBS): % : %.c
        $(CC) $< -o $@
.PHONY: clean
clean:
        rm -f $(OBS) *.o *~

leah@leah-K42JZ:~/Desktop/Documents/1$ █

```

图 2-1 实验 1-1 测试过程及结果

```

leah@leah-K42JZ:~/Desktop/Documents/1$ ./1-2
█

```

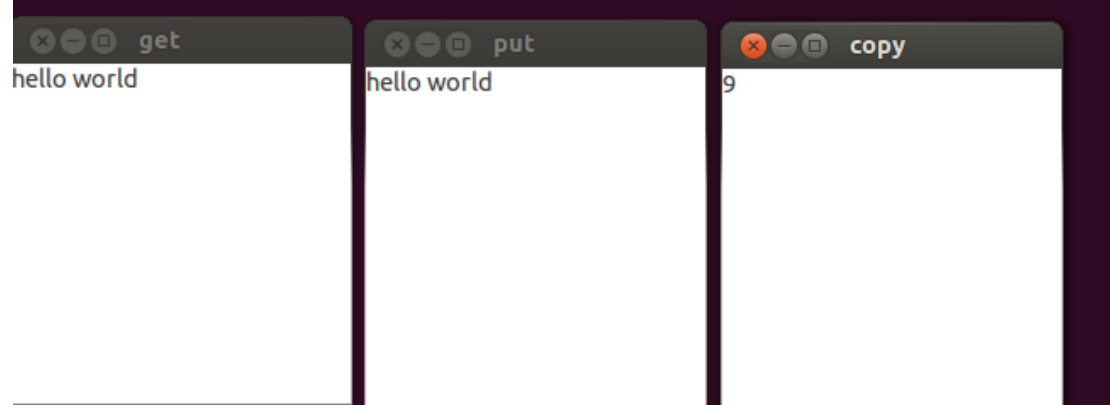


图 2-2 实验 1-2 测试过程及结果

3 实验二 系统调用

3.1 实验目的与内容

实验目的：

掌握系统调用的实现过程，通过编译内核方法，增加一个新的系统调用；

实验类容：

增加一个系统调用实现文件拷贝，编写一个程序调用新增加的系统调用。

3.2 实验原理与方案

文件复制的功能在实验一中已经实现，只不过系统调用时不能使用普通的读写函数，需要调用内核级的文件操作函数，因此只需对/kernel/sys.c 文件稍加改动，即可实现文件复制功能；同时需要在/arch/x86/kernel/syscall_table_32.S 中增加一条系统调用，然后在/arch/x86/include/asm/unistd_32.h 文件中进行申明即可编译通过。

内核编译我选取的是 ubuntu10.04.4 作为操作系统，内核选用 2.6.34.14；编译前需要安装 build-essential 和 libncursesw5-dev 这两个包，然后使用解压命令将内核解压到/usr/src/目录中，然后执行 make menuconfig 命令生成配置文件，接着就可以执行 make bzImage 生成系统映像文件了，接着就是编译模块，使用 make modules 和 make modules_install,最后执行 make install 就生成了新的系统引导项，之后只需修改 grub 文件重启进入系统了。

3.3 实验测试与结果

如图 3-1、3-2 和 3-3 所示，程序运行成功，完成文件拷贝功能；

如图 3-4 所示，当前系统版本为 2.34.14，与编译的内核版本一致。

```
leah@leah-laptop:~/Documents/2$ ./2 Makefile ./hello
success
```

图 3-1 运行程序

```
leah@leah-laptop:~/Documents/2$ diff ./Makefile ./hello
```

图 3-2 比较文件

```
leah@leah-laptop:~/Documents/2$ cat ./hello
CC = gcc
OBSJS = 2
all: $(OBSJS)
.PHONY:clean
clean:
    -rm -f $(OBSJS) *.o *~
```

图 3-3 查看文件内容

```
leah@leah-laptop:~$ uname -a
Linux leah-laptop 2.6.34.14 #2 SMP Tue Mar 12 22:22:36 CST 2013 i686 GNU/Linux
```

图 3-4 查看内核版本

4 实验三 字符设备驱动

4.1 实验目的与内容

实验目的：掌握增加设备驱动程序的方法；

实验类容：通过模块方法，增加一个新的设备驱动程序，其功能可以简单。

4.2 实验原理与方案

根据实验指导得知：一个典型的驱动程序,大体上可以分为这么几个部分:注册设备、定义功能函数、卸载设备。同时结合老师所给的模板填充功能函数即可完成字符设备的功能。当然由于设备的数据操作与普通文件有所不同，需要注意数据交换的方式与内存申请、释放的方法。

4.3 实验测试与结果

如图 4-1 所示，执行相应命令后可以看到数据成功写入且读出；同时在系统设备中可以看到我们的设备如图 4-2 所示；在做这些操作的时候，会将信息写入系统日志，通过查看发现与我们执行的命令一致，如图 4-3 所示。

```
root@leah-K42JZ:~/Desktop/Documents/3# insmod 3.ko
root@leah-K42JZ:~/Desktop/Documents/3# mknod /dev/3 c 60 0
root@leah-K42JZ:~/Desktop/Documents/3# chmod 666 /dev/3
root@leah-K42JZ:~/Desktop/Documents/3# echo -n abcdef >/dev/3
root@leah-K42JZ:~/Desktop/Documents/3# cat /dev/3
froot@leah-K42JZ:~/Desktop/Documents/3# cat /proc/devices
```

图 4-1 执行命令测试程序

```
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
5 ttyprintk
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
29 fb
60 3
```

图 4-2 查看系统设备

```
root@leah-K42JZ:~/Desktop/Documents/3# rmmod 3
root@leah-K42JZ:~/Desktop/Documents/3# rm /dev/3
root@leah-K42JZ:~/Desktop/Documents/3# cat /var/log/syslog | grep memory-module
Mar 25 18:22:12 leah-K42JZ kernel: [14353.795684] Inserting memory-module
Mar 25 18:23:15 leah-K42JZ kernel: [14416.613361] Removing memory-module
```

图 4-3 系统日志信息

5 实验四 系统监视器

5.1 实验目的与内容

实验目的：了解和掌握/proc 文件系统的特点和使用方法。

实验类容：

1. 了解/proc 文件的特点和使用方法；
2. 监控系统状态，显示系统中若干部件使用情况；
3. 用图形界面实现系统监控状态。

5.2 实验原理与方案

Linux 的 PROC 文件系统是进程文件系统和内核文件系统的组成的复合体，是将内核数据对象化为文件形式进行存取的一种内存文件系统，是监控内核的一种用户接口。它拥有一些特殊的文件（纯文本），从中可以获取系统状态信息。

根据上述理论，实验时只需读取文本内的信息即可获取系统状态信息；实验的 GUI 部分采用 GTK 图形库，利用 glade 软件进行图形界面的绘制，之后再编写代码实现读取和显示系统状态信息。

5.3 实验测试与结果

如图 5-1 所示，显示了程序运行时获取的系统基本信息：操作系统、内核版本、内存、cpu 等计算机基本信息；如图 5-2 所示，显示了系统进程的基本信息和操作系统的负载数据；如图 5-3 所示，显示了计算机的 cpu、内存、交换分区的使用情况；如图 5-4 所示，显示了系统文件系统的基本信息；如图 5-5 所示，显示了关于本程序的一些基本信息。

通过运行系统自带的任务管理器，对比数据发现：在进程处理方面，信息量远不及系统程序，但是显示的数据还是正确的；在显示内存、cpu 使用情况上，知识使用了简单的进度条的形式进行展示，虽然如此，数据还是一致的。

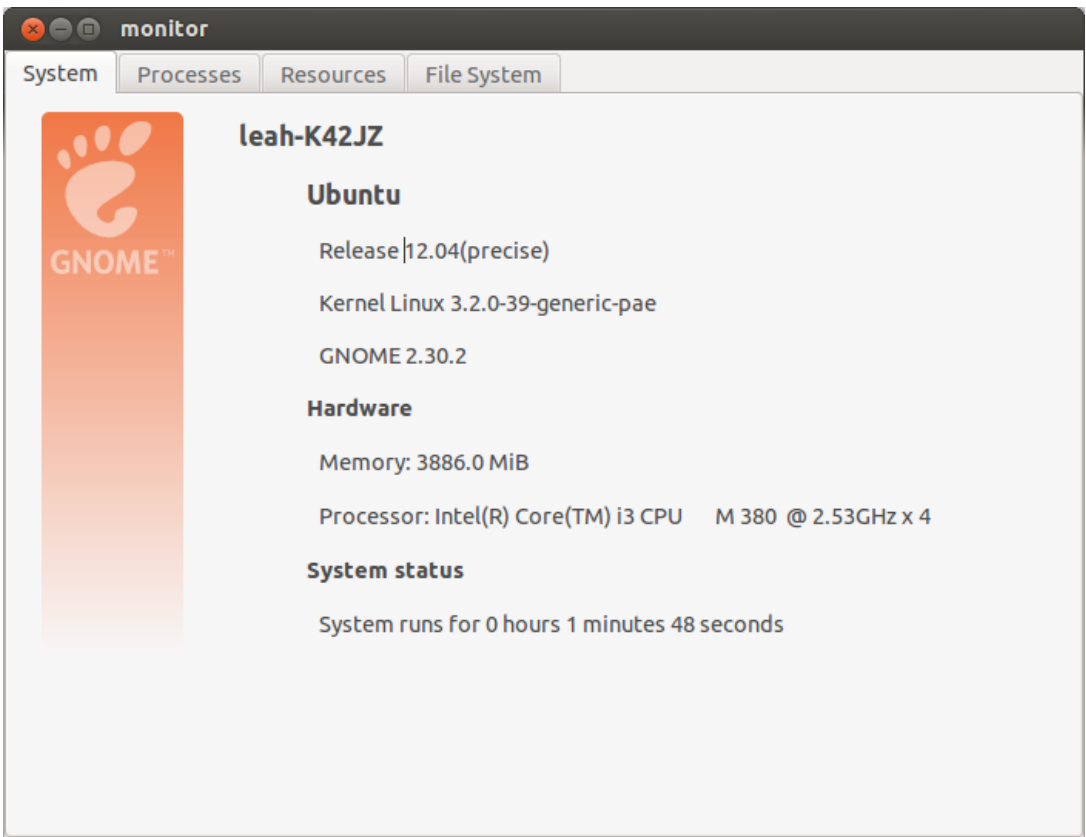


图 5-1 程序第一个标签页

| Process Name | Status | CPU% | ID | Virtual Memory | waiting channel | Nice |
|--------------|--------|------|----|----------------|-----------------------|------|
| init | S | 1.28 | 1 | 3.57 MiB | poll_schedule_timeout | 0 |
| kthreadd | S | 0.00 | 2 | 0.00 MiB | kthreadd | 0 |
| ksoftirqd/0 | S | 0.00 | 3 | 0.00 MiB | run_ksoftirqd | 0 |
| kworker/0:0 | S | 0.00 | 4 | 0.00 MiB | worker_thread | 0 |
| kworker/u:0 | S | 0.05 | 5 | 0.00 MiB | worker_thread | 0 |
| migration/0 | S | 0.00 | 6 | 0.00 MiB | cpu_stopper_thread | 0 |
| watchdog/0 | S | 0.00 | 7 | 0.00 MiB | watchdog | 0 |
| migration/1 | S | 0.07 | 8 | 0.00 MiB | cpu_stopper_thread | 0 |
| kworker/1:0 | S | 0.00 | 9 | 0.00 MiB | worker_thread | 0 |
| ksoftirqd/1 | S | 0.00 | 10 | 0.00 MiB | run_ksoftirqd | 0 |
| kworker/0:1 | S | 0.02 | 11 | 0.00 MiB | worker_thread | 0 |
| watchdog/1 | S | 0.00 | 12 | 0.00 MiB | watchdog | 0 |
| migration/2 | S | 0.00 | 13 | 0.00 MiB | cpu_stopper_thread | 0 |
| kworker/2:0 | S | 0.03 | 14 | 0.00 MiB | worker_thread | 0 |
| ksoftirqd/2 | S | 0.00 | 15 | 0.00 MiB | run_ksoftirqd | 0 |
| watchdog/2 | S | 0.00 | 16 | 0.00 MiB | watchdog | 0 |
| migration/3 | S | 0.08 | 17 | 0.00 MiB | cpu_stopper_thread | 0 |
| kworker/3:0 | S | 0.00 | 18 | 0.00 MiB | worker_thread | 0 |

图 5-2 程序第二个标签页

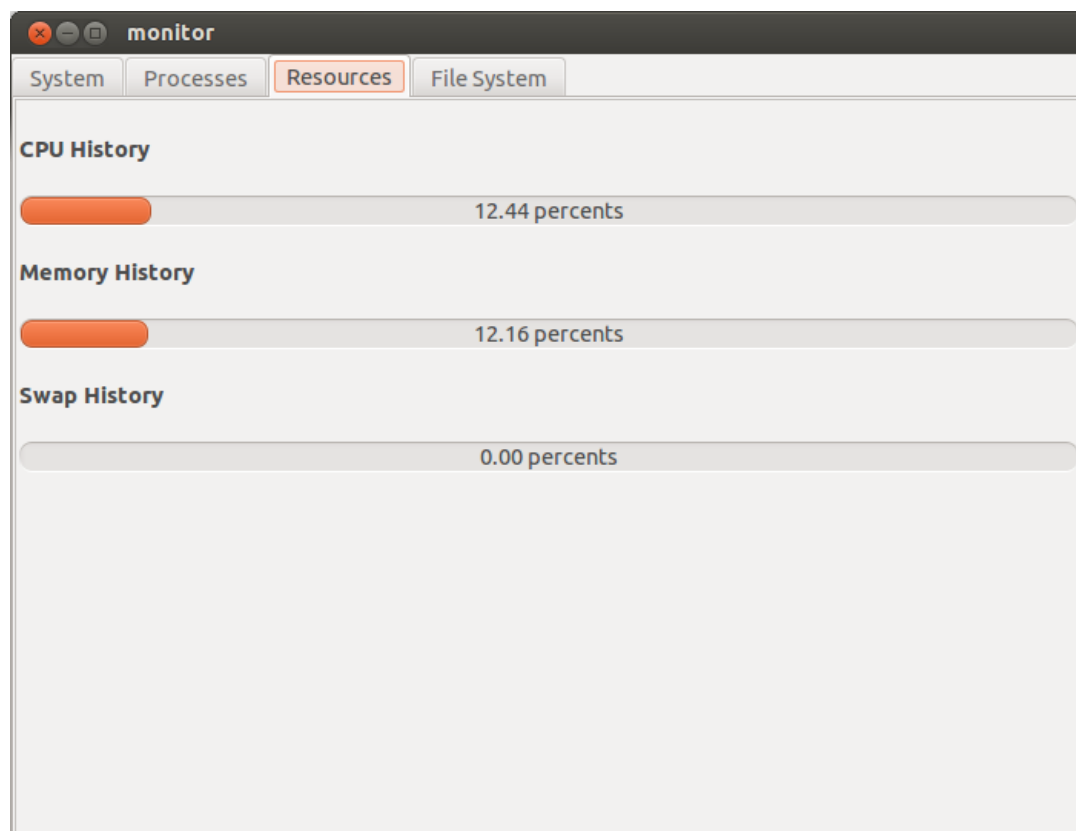


图 5-3 程序第三个标签页

| File System | | | | | | |
|-------------|--------------|---------|-----------|-----------|-----------|-----------|
| Devices | Directory | Type | Total | Free | Available | Used |
| /dev/sda7 | / | ext4 | 13.75 GiB | 9.19 GiB | 8.49 GiB | 4.56 GiB |
| /dev/sda6 | /media/music | fuseblk | 87.89 GiB | 31.49 GiB | 31.49 GiB | 56.40 GiB |
| /dev/sda8 | /home | ext4 | 9.17 GiB | 7.75 GiB | 7.29 GiB | 1.41 GiB |
| /dev/sdb1 | /media/leah | fuseblk | 14.91 GiB | 12.03 GiB | 12.03 GiB | 2.88 GiB |
| /dev/sda5 | /media/study | fuseblk | 63.61 GiB | 24.52 GiB | 24.52 GiB | 39.09 GiB |

图 5-4 程序第四个标签页

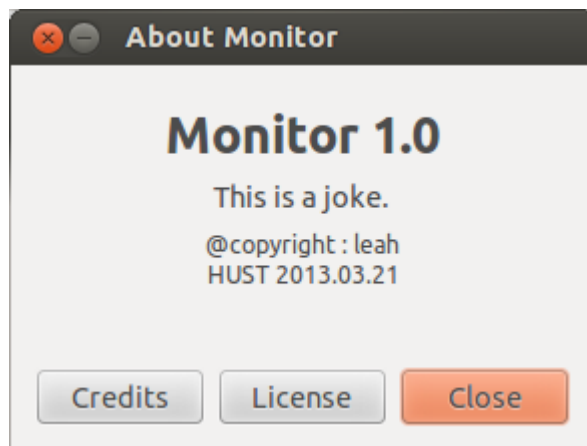


图 5-5 关于程序窗口

6 实验总结与评价

本次实验对于 linux 环境下的编程与基于 GTK 图形库的界面编程有着不小的检验，虽然最开始对于 GTK 编程不是很了解，但是在借助网络教程与优秀的界面设计器 GLADE 下，成功完成了实验内容。

总体说起来有以下几点总结：

1. 选取合适的 linux 版本和内核版本是成功的前提，许多同学由于没有采用老师的建议，用了比较高的版本，编译内核出现了很多问题，减缓了实验的进度；

2. 对 linux 环境的熟悉可以让你事半功倍，由于之前实现是在 linux 环境下完成的，对于 ubuntu 环境下的操作命令比较熟悉，因此做实验的时候没有花很多的时候上网查指令；

3. 对于图形界面编程来说短时间内难以学通，但是在 glade 的帮助下，你不需要过于关心界面，而是关注数据处理和消息事件的处理即可，这样一来不用一行一行代码的编写图形界面。

当然实验过程也出现了不少的 bug，比如实验 4 中的关于窗口最开始关闭后不能再次点击，上网查询了才知道不能销毁这个窗口，否则再次点击时窗口句柄已经不存在，无法响应；还有就是程序编译能通过但是执行的时候就是段错误，最开始没有想到调试(主要是不知道如何快速的调试)，后来才慢慢使用 gdb 进行调试，这样一来一眼看不错来的问题都能找到出错点，然后解决。实际上，实验 4 还有很多功能不完善或者没有实现，比如 kill 进程这个操作没有做，主要是考虑到有些进程不能杀死，有些则会带来问题，因此就没有编写这个按钮；在刷新进程信息上处理不科学，最开始的版本是对时间取样然后计算 cpu 利用率，但是这样一来刷新时间与取样时间的平衡难以维持，后来去除了取样，直接计算总的 cpu 利用率；关于文件系统信息的代码，由于 /proc 文件中关于这个的信息太少，仅仅读取文件不能显示文件系统的所有信息，因此就上网查询得到这样的版本，处理起来也比较复杂，用到了几个结构体。

总体来说，这次实验获取的经验还是很对的，特别是界面编程和有关内核的一部分，这些东西在操作系统课本上基本没有，这次实验取材于课本而高于课本，对于学生的学习能力也是不小的考验。总之只有会用 Linux，了解 Linux，才能高效地编写和调试程序。

7 参考文献

[1]庞丽萍.操作系统原理.武汉：华中科技大学出版社，2008.

[2]URL: <http://blog.chinaunix.net/uid-24145413-id-194358.html>

[3] URL :<http://tadeboro.blogspot.com>

8 附录

实验 1.1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    int from_fd, to_fd;
    int bytes_read, bytes_write;
    char buffer[100];
    char *ptr;
    if(argc!=3)
    {
        fprintf(stderr, "Usage: %s from file to file\n", argv[0]);
        exit(1);
    }
    if((from_fd=open(argv[1], O_RDONLY))==-1)
    {
        return -1;
    }
    if((to_fd=open(argv[2], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR))==-1)
    {
        return -2;
    }
    while(bytes_read=read(from_fd, buffer, 100))
    {
        if((bytes_read==0)&&(errno!=EINTR)) break;
        else if(bytes_read>0)
```

```

    {
        ptr=buffer;
        while(bytes_write=write(to_fd,ptr,bytes_read))
        {
            if((bytes_write==-1)&&(errno!=EINTR)) break;
            else if(bytes_write==bytes_read) break;
            else if(bytes_write>0)
            {
                ptr+=bytes_write;
                bytes_read-=bytes_write;
            }
        }
        if(bytes_write==-1) break;
    }
}
close(from_fd);
close(to_fd);
return 0;
}

```

实验 1.2

1-2.c

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    pid_t p1, p2, p3;
    if((p1 = fork()) == 0) {
        execv("./2/get", NULL);
    } else {
        if((p2 = fork()) == 0) {
            execv("./2/copy", NULL);
        } else {
            if((p3 = fork()) == 0) {
                execv("./2/put", NULL);
            }
        }
    }
}

```

```

        }
    }

    p1 = wait(&p1);
    p2 = wait(&p2);
    p3 = wait(&p3);
    return;
}

```

copy.c

```

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

GtkTextBuffer *buffer;
GtkWidget *text_view;
int i;

int fresh(GtkWidget *text_view)
{
    char str[10];
    i++;
    sprintf(str, "%d", i);
    gtk_text_buffer_set_text(buffer, str, -1); //显示数据
    return TRUE;
}

int main(int argc, char *argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;

    gtk_init(&argc, &argv);
    /* Create a Window. */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "copy");
    /* Set a decent default size for the window. */
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 200);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

```

```

g_signal_connect(G_OBJECT(window), "destroy",
                 G_CALLBACK(gtk_main_quit), NULL);
vbox = gtk_vbox_new (FALSE, 2);
gtk_container_add (GTK_CONTAINER (window), vbox);
/* Create a multiline text widget. */
text_view = gtk_text_view_new ();
gtk_box_pack_start (GTK_BOX (vbox), text_view, 1, 1, 0);
buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text_view));
gtk_text_buffer_set_text(buffer, " ", -1);
g_timeout_add(1000, (GSourceFunc)fresh, (gpointer)text_view);
gtk_widget_show_all(window);
gtk_main();
return 0;
}

```

实验 2

sys.c

```

asmlinkage int sys_mySysCall(char *src, char *dst)
{
    int bytes_read, bytes_write;
    int from_fd, to_fd;
    char buffer[100];
    char *ptr;
    mm_segment_t old_fs;
    old_fs = get_fs();
    set_fs(KERNEL_DS);
    if((from_fd = sys_open(src, O_RDONLY, 0)) == -1)
        return -1;
    if((to_fd = sys_open(dst, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR)) == -1)
        return -2;
    while((bytes_read = sys_read(from_fd, buffer, 1)))
    {
        if(bytes_read == -1) break;
        else if(bytes_read > 0)
        {

```

```

        ptr = buffer;
        while((bytes_write = sys_write(to_fd, ptr, bytes_read)))
        {
            if(bytes_write == -1) break;
            else if(bytes_write == bytes_read) break;
            else if(bytes_write > 0)
            {
                ptr += bytes_write;
                bytes_read -= bytes_write;
            }
        }
        if(bytes_write == -1) break;
    }
}
set_fs(old_fs);
return 0;
}

```

测试代码:

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    if(argc != 3)
        printf("error\n");
    else
    {
        i = syscall(338, argv[1], argv[2]); // 系统调用
        printf("success\n");
    }
    return 0;
}

```

实验 3

```

#include <linux/init.h>

```



```

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/slab.h>
#include <linux/types.h>
#include <linux/proc_fs.h>
#include <linux/fcntl.h>
#include <asm/system.h>
#include <asm/uaccess.h>
MODULE_LICENSE("GPL");

int memory_open(struct inode *inode, struct file *filp);
int memory_release(struct inode *inode, struct file *filp);
ssize_t memory_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t memory_write(struct file *filp, char *buf, size_t count, loff_t *f_pos);
void memory_exit(void);
int memory_init(void);
struct file_operations memory_fops = {
    .read = memory_read,
    .write = memory_write,
    .open = memory_open,
    .release = memory_release
};
module_init(memory_init);
module_exit(memory_exit);
int memory_major = 60;
char *memory_buffer;
int memory_init(void)
{
    int result;
    result = register_chrdev(memory_major, "3", &memory_fops);
    if(result < 0) {
        printk("<1>memory: cannot obtain major number %d\n", memory_major);
    }
    memory_buffer = kmalloc(1, GFP_KERNEL);

```

```

    if(!memory_buffer) {
        result = -ENOMEM;
        memory_exit();
        return result;
    }
    memset(memory_buffer, 0, 1);
    printk("<1>Inserting memory-module\n");
    return 0;
}

void memory_exit(void)
{
    unregister_chrdev(memory_major, "3");
    if(memory_buffer) {
        kfree(memory_buffer);
    }
    printk("<1>Removing memory-module\n");
}

int memory_open(struct inode *inode, struct file *filp)
{
    return 0;
}

int memory_release(struct inode *inode, struct file *filp)
{
    return 0;
}

ssize_t memory_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
    copy_to_user(buf, memory_buffer, 1); //转向用户态
    if(*f_pos == 0) {
        *f_pos += 1;
        return 1;
    }
}

```

```

    } else {
        return 0;
    }
}

ssize_t memory_write(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
    char *tmp;
    tmp = buf + count - 1;
    copy_from_user(memory_buffer, tmp, 1); //转向内核态
    return 1;
}

```

实验 4

```

#include <gtk/gtk.h>
#include <stdio.h>
#include <stdlib.h>
#include "fcntl.h"
#include <unistd.h>
#include <string.h>
#include <sys/vfs.h>
#include <mntent.h>
#include "dirent.h"
float IDLE, TOTAL;
GtkWidget *dialog;
int cpu_core;
enum {
    DEVICES_COLUMN,
    DIR_COLUMN,
    TYPE_COLUMN,
    TOTAL_COLUMN,
    FREE_COLUMN,
    AVAILABLE_COLUMN,
    USED_COLUMN
};
enum {

```

```

        PROCESS_COLUMN,
        STATUS_COLUMN,
        CPU_COLUMN,
        ID_COLUMN,
        MEMORY_COLUMN,
        WAIT_COLUMN,
        NICE_COLMN
};

typedef struct {
    char device[256];
    char mntpnt[256];
    char type[256];
    long blocks;
    long bfree;
    long bused;
    long available_disk;
    int bused_percent;
} DiskInfo;

void reboot(GtkWidget *window,gpointer data)
{
    system("reboot");
}

void shutdown(GtkWidget *window,gpointer data)
{
    system("shutdown now");
}

void about(GtkWidget *window, gpointer data)
{
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_hide(dialog);//应该隐藏而不是销毁
}

//此处为网上 copy 的代码
int get_disk_info(GtkListStore *store)
{
    gtk_list_store_clear(store);

```

```

DiskInfo *disk_info;

struct statfs fs_info;

struct mntent *mnt_info;

FILE *fh;

float percent;

char disk_info_total[32];
char disk_info_used[32];
char disk_info_free[32];
char disk_info_available[32];

if((fh = setmntent( "/etc/mtab", "r" )) == NULL) {
    printf("Cannot open \'/etc/mtab\'!\n");
    return -1;
}

while ((mnt_info = getmntent(fh)) != NULL) {
    if(statfs(mnt_info->mnt_dir, &fs_info) < 0) {
        continue;
    }

    if((disk_info = (DiskInfo *)malloc(sizeof(DiskInfo)))==NULL) {
        continue;
    }

    //屏蔽某些文件系统类型
    if(strcmp( mnt_info->mnt_type, "proc") &&
        strcmp(mnt_info->mnt_type, "devfs") &&
        strcmp(mnt_info->mnt_type, "vboxsf") &&
        strcmp(mnt_info->mnt_type, "devtmpfs") &&
        strcmp(mnt_info->mnt_type, "usbfs") &&
        strcmp(mnt_info->mnt_type, "sysfs") &&
        strcmp(mnt_info->mnt_type, "tmpfs") &&
        strcmp(mnt_info->mnt_type, "devpts")&&
        strcmp(mnt_info->mnt_type, "fusectl") &&
        strcmp(mnt_info->mnt_type, "debugfs") &&
        strcmp(mnt_info->mnt_type, "binfmt_misc")&&
        strcmp(mnt_info->mnt_type, "fuse.gvfs-fuse-daemon")&&
        strcmp(mnt_info->mnt_type, "securityfs")) {
        if(fs_info.f_blocks != 0) {

```

```

        percent = (((float)fs_info.f_blocks - (float)fs_info.f_bfree)
                    * 100.0 / (float)fs_info.f_blocks);
    } else {
        percent = 0;
    }
} else {
    continue;
}

strcpy(disk_info->type, mnt_info->mnt_type);
strcpy(disk_info->device, mnt_info->mnt_fsname);
strcpy(disk_info->mntpnt, mnt_info->mnt_dir);
long block_size = fs_info.f_bsize / 1024;
disk_info->blocks = fs_info.f_blocks * block_size / 1024;
disk_info->bfree = fs_info.f_bfree * block_size / 1024;
disk_info->available_disk = fs_info.f_bavail * block_size / 1024;
disk_info->bused = (fs_info.f_blocks - fs_info.f_bfree) * block_size / 1024;
disk_info->bused_percent = (int) percent;
sprintf(disk_info_total, "%.2f GiB",
        (float) disk_info->blocks / 1024.0);
sprintf(disk_info_used, "%.2f GiB",
        (float) disk_info->bused / 1024.0);
sprintf(disk_info_free, "%.2f GiB",
        (float) disk_info->bfree / 1024.0);
sprintf(disk_info_available, "%.2f GiB",
        (float) disk_info->available_disk / 1024.0);

GtkTreeIter iter;
gtk_list_store_append(store, &iter);
gtk_list_store_set(store, &iter,
    DEVICES_COLUMN, disk_info->device,
    DIR_COLUMN, disk_info->mntpnt,
    TYPE_COLUMN, disk_info->type,
    TOTAL_COLUMN, disk_info_total,
    FREE_COLUMN, disk_info_free,
    AVAILABLE_COLUMN, disk_info_available,
    USED_COLUMN, disk_info_used,

```

```

        -1);

    };

    return TRUE;
}

//获取系统负载信息
int fresh_load(GtkWidget *label)
{
    int fd;
    char buffer[32];
    char *load_info[3];
    char str_load[64];
    fd = open("/proc/loadavg", O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    close(fd);
    load_info[0] = strtok(buffer, " ");
    load_info[1] = strtok(NULL, " ");
    load_info[2] = strtok(NULL, " ");
    sprintf(str_load, "Load average for the last 1, 5, 15minutes :%s, %s, %s",
            load_info[0], load_info[1], load_info[2]);
    gtk_label_set_text((GtkLabel *)label, str_load);
    return TRUE;
}

//显示进程信息
int get_process_info(GtkListStore *store){
    gtk_list_store_clear(store);
    DIR *dir;
    struct dirent *entry;
    int fd1, j;
    FILE *fd2;
    char dir_buf[256];
    char buffer[128];
    char *mem_info[26];
    char *delim = " ";
    GtkTreeIter iter;
    char rate_buffer[16];

```

```

double mem;
char mem_buffer[16];
char dir_temp[256];
char buffer_temp[32];
char *cpu_time[6][2];
char *process_time[20][2];
char buffer_cpu_time[128];
char buffer_process_time[128];
double sum_cpu_time[2];
double sum_process_time[2];
double total_cpu_time, total_process_time;
char process_cpu_rate[10];
double temp_rate;
int i;
dir = opendir ("/proc");
//查询并找出需要的目录
while ((entry = readdir(dir)) != NULL) {
    if ((entry->d_name[0] >= '0') && (entry->d_name[0] <= '9')) {
        sprintf(dir_buf, "/proc/%s/stat", entry->d_name);
        fd1 = open(dir_buf, O_RDONLY);
        read(fd1, buffer, sizeof(buffer));
        close(fd1);
        sprintf(dir_temp, "/proc/%s/wchan", entry->d_name);
        fd2 = fopen(dir_temp, "r");
        while(!feof(fd2)) {
            fgets(buffer_temp, 100, fd2);
        }
        fclose(fd2);
        mem_info[0] = strtok(buffer, delim);
        int pid_info = atoi(mem_info[0]);
        for (j = 1; j < 26 ; j++) {
            mem_info[j] = strtok(NULL, delim);
        }
        char *process_name = strtok(strtok(mem_info[1], "("), ")");
        for(i = 0; i < 2; i++) {

```



```

int k;
int fd_1, fd_2;
fd_1 = open("/proc/stat", O_RDONLY);
read(fd_1, buffer_cpu_time, sizeof(buffer_cpu_time));
close(fd_1);
fd_2 = open(dir_buf, O_RDONLY);
read(fd_2, buffer_process_time, sizeof(buffer_process_time));
close(fd_2);
cpu_time[0][i] = strtok(buffer_cpu_time, delim);
sum_cpu_time[i] = 0;
for(k = 1; k < 6; k++) {
    cpu_time[k][i] = strtok(NULL, delim);
    if((k >= 1) && (k <= 5)) {
        sum_cpu_time[i] += atof(cpu_time[k][i]);
    }
}
process_time[0][i] = strtok(buffer_process_time, delim);
sum_process_time[i] = 0;
for(k = 1; k < 20; k++) {
    process_time[k][i] = strtok(NULL, delim);
    if((k >= 13) && (k <= 16)) {
        sum_process_time[i] += atof(process_time[k][i]);
    }
}
}
temp_rate = ((sum_process_time[0] * 100 / sum_cpu_time[0]) +
             (sum_process_time[1] * 100 / sum_cpu_time[0])) / 2;
//事实上应该在时间上取样分析数据
sprintf(process_cpu_rate, "%.2f", temp_rate);
mem = atoi(mem_info[22]);
mem = mem / (1024 * 1024);
sprintf(mem_buffer, "%.2f MiB", mem);
gtk_list_store_append(store, &iter);
gtk_list_store_set(store, &iter,
    PROCESS_COLUMN, process_name,

```

```

        STATUS_COLUMN, mem_info[2],
        CPU_COLUMN, process_cpu_rate,
        ID_COLUMN, pid_info,
        MEMORY_COLUMN, mem_buffer,
        WAIT_COLUMN, buffer_temp,
        NICE_COLMN, mem_info[18],
        -1);
    }
}

closedir (dir);
return TRUE;
}

//获取系统基本信息
void get_system_info(GtkWidget *labelA,
    GtkWidget *labelB,
    GtkWidget *labelC,
    GtkWidget *labelE,
    GtkWidget *labelF)
{
    int fd_host, fd_release, fd_version, fd_mem, fd_cpu;
    char *info[4];
    char *info_host;
    char str_host[32];
    char buffer_host[30];
    char buffer_release[128];
    char *info_release;
    char *start_release;
    char str_release[30];
    char *start_codename;
    char buffer_version[30];
    char str_version[30];
    char *info_version;
    char buffer_mem[30];
    char *info_mem;
    char str_mem[30];

```

```

char buffer_cpu[1000];
char *info_cpu[6];
char *temp;
char str_cpu[50];
char *start_cpu;
fd_host = open("/etc/hostname", O_RDONLY);
read(fd_host, buffer_host, sizeof(buffer_host));
close(fd_host);
info_host = strtok(buffer_host, "\n");
sprintf(str_host, "      %s", info_host);
gtk_label_set_text((GtkLabel *)labelA, str_host);
fd_release = open("/etc/lsb-release", O_RDONLY);
read(fd_release, buffer_release, sizeof(buffer_release));
close(fd_release);
info[0] = strtok(buffer_release, "\n");
int i;
for(i = 0; i < 3; i++) {
    info[i] = strtok(NULL, "\n");
}
start_release = strstr(info[0], "=");
start_release++;
start_codename = strstr(info[1], "=");
start_codename++;
sprintf(str_release, "      Release %s(%s)",
        start_release, start_codename);
gtk_label_set_text((GtkLabel *)labelB, str_release);
fd_version = open("/proc/sys/kernel/osrelease", O_RDONLY);
read(fd_version, buffer_version, sizeof(buffer_version));
close(fd_version);
info_version = strtok(buffer_version, "\n");
sprintf(str_version, "      Kernel Linux %s", info_version);
gtk_label_set_text((GtkLabel *)labelC, str_version);
fd_mem = open("/proc/meminfo", O_RDONLY);
read(fd_mem, buffer_mem, sizeof(buffer_mem));
close(fd_mem);

```

```

info_mem = strtok(buffer_mem, " ");
info_mem = strtok(NULL, " ");
int j;
float k;
j = atoi(info_mem);
k = j / 1024;
sprintf(str_mem, "          Memory: %.1f MiB", k);
gtk_label_set_text((GtkLabel *)labelE, str_mem);
fd_cpu = open("/proc/cpuinfo", O_RDONLY);
read(fd_cpu, buffer_cpu, sizeof(buffer_cpu));
close(fd_cpu);
info_cpu[0] = strtok(buffer_cpu, "\n");
int m;
for(m = 0; m < 5; m++) {
    info_cpu[m + 1] = strtok(NULL, "\n");
}
start_cpu = strstr(info_cpu[4], ":");
start_cpu++;
start_cpu++;
sprintf(str_cpu, "          Processor: %s x %d", start_cpu, cpu_core);
gtk_label_set_text((GtkLabel *)labelF, str_cpu);
}

//cpu 利用率
int get_cpu_rate_info(GtkWidget *progressbar)
{
    int fd;
    char buffer[128];
    float cpu_rate_info;
    char *buffer_cpu_rate[8];
    char text_cpu[20];
    float total = 0;
    fd = open("/proc/stat", O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    close(fd);
    buffer_cpu_rate[0] = strtok(buffer, " ");

```

```

int i;
for(i = 1; i < 5; i++) {
    buffer_cpu_rate[i] = strtok(NULL, " ");
    total += atof(buffer_cpu_rate[i]);
}
cpu_rate_info = (total - TOTAL - atof(buffer_cpu_rate[4]) +
    IDLE) * 100 / (total - TOTAL);
TOTAL = total;
IDLE = atof(buffer_cpu_rate[4]);
sprintf(text_cpu, "%.2f percents", cpu_rate_info);
//设置进度条文字与显示数值
gtk_progress_bar_set_fraction(GTK_PROGRESS_BAR(progressbar), cpu_rate_info / 100);
gtk_progress_bar_set_text(GTK_PROGRESS_BAR(progressbar), text_cpu);
return TRUE;
}
//交换分区利用率
int get_swap_rate_info(GtkWidget *progressbar)
{
    int fd;
    char buffer[128];
    int i;
    float swap_rate_info;
    char text_swap[32];
    char *buffer_swap_info[8];
    fd = open("/proc/swaps", O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    close(fd);
    buffer_swap_info[0] = strtok(buffer, "\n");
    buffer_swap_info[1] = strtok(NULL, "\n");
    buffer_swap_info[2] = strtok(buffer_swap_info[1], " ");
    for(i = 3; i < 8; i++) {
        buffer_swap_info[i] = strtok(NULL, " ");
    }
    if(0 == atof(buffer_swap_info[5])) {
        swap_rate_info = 0.0;
    }
}

```

```

    } else {
        swap_rate_info = atof(buffer_swap_info[5]) * 100.0 / atof(buffer_swap_info[4]);
    }
    sprintf(text_swap, "%.2f percents", swap_rate_info);
    gtk_progress_bar_set_fraction (GTK_PROGRESS_BAR(progressbar), swap_rate_info /
100);
    gtk_progress_bar_set_text(GTK_PROGRESS_BAR(progressbar), text_swap);
    return TRUE;
}
int get_mem_rate_info(GtkWidget *progressbar)
{
    int fd;
    char buffer[128];
    float buffer_mem_info[8];
    float mem_rate_info;
    char text_mem[32];
    fd = open("/proc/meminfo", O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    close(fd);
    buffer_mem_info[0] = atof(strtok(buffer, " "));
    int i;
    for(i = 1; i < 8; i++) {
        buffer_mem_info[i] = atof(strtok(NULL, " "));
    }
    mem_rate_info = (buffer_mem_info[1] - buffer_mem_info[3] -
        buffer_mem_info[5] - buffer_mem_info[7]) * 100 / (buffer_mem_info[1]);
    sprintf(text_mem, "%.2f percents", mem_rate_info);
    gtk_progress_bar_set_fraction (GTK_PROGRESS_BAR(progressbar), mem_rate_info /
100);
    gtk_progress_bar_set_text(GTK_PROGRESS_BAR(progressbar), text_mem);
    return TRUE;
}
//刷新启动时间
int fresh_uptime(GtkWidget *label)
{

```

```

FILE *fp_uptime;
char buffer_uptime[32];//读文件
char *info_uptime[3];//分割
char str_uptime[128];//输出缓冲
fp_uptime = fopen("/proc/uptime", "r");
while(!feof(fp_uptime)) {
    fgets(buffer_uptime, 100, fp_uptime);
}
fclose(fp_uptime);
info_uptime[0] = strtok(buffer_uptime, " ");
info_uptime[1] = strtok(NULL, " ");
int uptime_temp = (int)(atof(info_uptime[1]) / cpu_core);
sprintf(str_uptime, "\t\t\tSystem runs for %d hours %d minutes %d seconds",
        uptime_temp/ 3600, uptime_temp % 3600 / 60, uptime_temp % 60);
gtk_label_set_text((GtkLabel *)label, str_uptime);
return TRUE;
}
//获取 cpu 核心数
int get_cpu_core()
{
    int fd;
    int i;
    char *buffer_cpu_info[9];
    char buffer[2048];
    char *pch;
    fd = open("/proc/stat", O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    close(fd);
    buffer_cpu_info[0] = strtok(buffer, "\n");
    for(i = 1; i < 9; i++) {
        buffer_cpu_info[i] = strtok(NULL, "\n");
    }
    for(i = 0; i < 9; i++) {
        pch = strchr(buffer_cpu_info[i], 'c');
        if((pch != NULL) && (strchr(++pch, 'p') != NULL))

```

```

        && (strchr(++pch, 'u') != NULL)) {
            cpu_core++;
        }
    }
    return (--cpu_core);
}

int main(int argc, char *argv[])
{
    GtkBuilder *builder;
    GtkWidget *notebook;
    GtkWidget *window;
    GtkWidget *quit_menu, *reboot_menu, *shutdown_menu, *about_menu;
    GtkWidget *label1, *label2, *label3, *label4, *label5, *label7, *label8;
    GtkWidget *label9, *label11, *label12, *label14, *label15;
    GtkListStore *store_disk, *store_process;
    GtkWidget *progressbar_cpu, *progressbar_mem, *progressbar_swap;
    GtkWidget *fresh_button;
    gtk_init(&argc, &argv);
    builder = gtk_builder_new();
    gtk_builder_add_from_file(builder, "monitor.glade", NULL); //读取 glade 文件中的界面信息
    window = GTK_WIDGET(gtk_builder_get_object(builder, "window"));
    quit_menu = GTK_WIDGET(gtk_builder_get_object(builder, "imagemenuitem5"));
    shutdown_menu = GTK_WIDGET(gtk_builder_get_object(builder, "imagemenuitem3"));
    reboot_menu = GTK_WIDGET(gtk_builder_get_object(builder, "imagemenuitem4"));
    about_menu = GTK_WIDGET(gtk_builder_get_object(builder, "imagemenuitem10"));
    notebook = GTK_WIDGET(gtk_builder_get_object(builder, "notebook"));
    label1 = GTK_WIDGET(gtk_builder_get_object(builder, "label1"));
    label2 = GTK_WIDGET(gtk_builder_get_object(builder, "label2"));
    label3 = GTK_WIDGET(gtk_builder_get_object(builder, "label3"));
    label4 = GTK_WIDGET(gtk_builder_get_object(builder, "label4"));
    label5 = GTK_WIDGET(gtk_builder_get_object(builder, "label5"));
    label7 = GTK_WIDGET(gtk_builder_get_object(builder, "label7"));
    label8 = GTK_WIDGET(gtk_builder_get_object(builder, "label8"));
    label11 = GTK_WIDGET(gtk_builder_get_object(builder, "label11"));
    label12 = GTK_WIDGET(gtk_builder_get_object(builder, "label12"));

```



```

label14 = GTK_WIDGET(gtk_builder_get_object(builder, "label14"));
label15 = GTK_WIDGET(gtk_builder_get_object(builder, "label15"));
dialog = GTK_WIDGET(gtk_builder_get_object(builder, "aboutdialog"));
progressbar_cpu = GTK_WIDGET(gtk_builder_get_object(builder, "progressbar_cpu"));
progressbar_mem = GTK_WIDGET(gtk_builder_get_object(builder, "progressbar_mem"));
progressbar_swap = GTK_WIDGET(gtk_builder_get_object(builder, "progressbar_swap"));
get_cpu_core();
get_system_info(label5, label7, label8, label11, label12);
store_disk = GTK_LIST_STORE(gtk_builder_get_object(builder, "store_disk"));
store_process = GTK_LIST_STORE(gtk_builder_get_object(builder, "store_process"));
g_timeout_add(1000, (GSourceFunc)fresh_uptime, (gpointer)label14);
g_timeout_add(1000, (GSourceFunc)fresh_load, (gpointer)label15);
g_timeout_add(1000, (GSourceFunc)get_cpu_rate_info, (gpointer)progressbar_cpu);
g_timeout_add(1000, (GSourceFunc)get_mem_rate_info, (gpointer)progressbar_mem);
g_timeout_add(1000, (GSourceFunc)get_swap_rate_info, (gpointer)progressbar_swap);
get_disk_info(store_disk);
get_process_info(store_process);
g_timeout_add(10000, (GSourceFunc)get_disk_info, (gpointer)store_disk);
g_timeout_add(10000, (GSourceFunc)get_process_info, (gpointer)store_process);
g_signal_connect(GTK_WINDOW(window), "destroy", G_CALLBACK(gtk_main_quit),
NULL);

g_signal_connect(quit_menu, "activate", G_CALLBACK(gtk_main_quit), NULL);
g_signal_connect(reboot_menu, "activate", G_CALLBACK(reboot), NULL);
g_signal_connect(shutdown_menu, "activate", G_CALLBACK(shutdown), NULL);
g_signal_connect(about_menu, "activate", G_CALLBACK(about), NULL);
gtk_widget_show(window);
g_object_unref(builder);
gtk_main();
return 0;
}

```