Linux 学习笔记

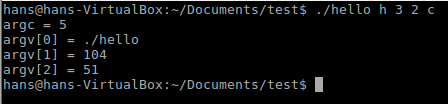
Created by Hans Hu at 2017/07/17

## 1. 系统函数使用说明

### 1、main函数原型：

int main(int *argc*, char \**argv[]*)

其中，argc为参数个数，argv为参数，首个字符串argv[0]标识函数名本身，参数个数不指定。示例如下图：



程序编写的只显示到参数argv[2]。参数个数为总的输入参数个数。从编译过程来看，除了第一个标识函数名本身的参数argv[0]为字符串之外，其余参数都是int类型参数，并将输入的参数按char型字符处理。如果某参数输入为字符串，只取第一个字符作为输入。

另外，输入的参数不是必须使用，但是如果有使用输入参数，则对应参数必须有输入，否则执行时将报错。

### 2、open函数原型：

int open(const char \**prthname*, int *flags*,…)

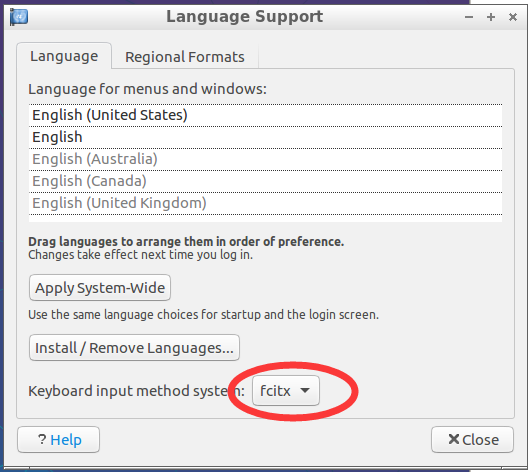
pathname为文件路径及文件名，flags为打开模式，打开模式一般为只读、只写、读写三者之一。返回值为int类型的值，文件打开成功返回一个正值，代表操作编号，类似句柄。文件打开错误则返回-1。

特别注意的是返回值fd，fd的值与文件的状态无关。在同一个进程下，fd的值按照顺序依次递增。不用进程下的fd之间无关，即使是打开同一个文件。可以理解成进程中的一个局部变量。

文件能否被打开多次，以及能否进行双工操作，由驱动函数完成，与系统函数无关。

## 2. Linux发行版lubuntu环境下中文输入法设置

1、首先设置系统preference下的language support下的keyboard input method system为fcitx如下图所示：



2、链接及内容如下

<http://www.lirui.name/post/251.html>

Lubuntu下面这个问题也算是老生常谈了。一直在更新，一直未修正。

简单说，如果运行了下面这条命令输入法就正常了的话：

killall fcitx-qimpanel

那么：

sudo apt-get remove fcitx-ui-qimpanel

问题解决！！

方法二[推荐]：

在输入法图标上右键，配置当前输入法。把Kimpanel前面的勾去掉。



3、链接及内容如下

安装第三方源并安装compton

sudo apt-add-repository ppa:richardgv/compton

sudo apt-get update && sudo apt-get install compton

配置compton

编辑文件 ~/.compton.conf，输入如下内容

backend = "glx";

paint-on-overlay = true;

glx-no-stencil = true;

glx-no-rebind-pixmap = true;

vsync = "opengl-swc";

# These are important. The first one enables the opengl backend. The last one

# is the vsync method. Depending on the driver you might need to use a

# different method.

# The other options are smaller performance tweaks that work well in most

# cases.

# You can find the rest of the options here:

# https://github.com/chjj/compton/wiki/perf-guide, and here:

# https://github.com/chjj/compton/wiki/vsync-guide

# Shadow

shadow = false;          # Enabled client-side shadows on windows.

no-dock-shadow = true;      # Avoid drawing shadows on dock/panel windows.

no-dnd-shadow = true;       # Don't draw shadows on DND windows.

clear-shadow = true;        # Zero the part of the shadow's mask behind the window (experimental).

shadow-radius = 7;      # The blur radius for shadows. (default 12)

shadow-offset-x = -7;       # The left offset for shadows. (default -15)

shadow-offset-y = -7;       # The top offset for shadows. (default -15)

shadow-exclude = [

    "! name~=''",

    "n:e:Notification",

    "n:e:Plank",

    "n:e:Docky",

    "g:e:Synapse",

    "g:e:Kupfer",

    "g:e:Conky",

    "n:w:\*Firefox\*",

    "n:w:\*Chrome\*",

    "n:w:\*Chromium\*",

    "class\_g ?= 'Notify-osd'",

    "class\_g ?= 'Cairo-dock'",

    "class\_g ?= 'Xfce4-notifyd'",

    "class\_g ?= 'Xfce4-power-manager'"

];

# The shadow exclude options are helpful if you have shadows enabled. Due to

# the way compton draws its shadows, certain applications will have visual

# glitches

# (most applications are fine, only apps that do weird things with xshapes or

# argb are affected).

# This list includes all the affected apps I found in my testing. The "!

# name~=''" part excludes shadows on any "Unknown" windows, this prevents a

# visual glitch with the XFWM alt tab switcher.

# Fading

fading = true; # Fade windows during opacity changes.

fade-delta = 4; # The time between steps in a fade in milliseconds. (default 10).

fade-in-step = 0.03; # Opacity change between steps while fading in. (default 0.028).

fade-out-step = 0.03; # Opacity change between steps while fading out. (default 0.03).

#no-fading-openclose = true; # Fade windows in/out when opening/closing

detect-client-opacity = true; # This prevents opacity being ignored for some apps. For example without this enabled my xfce4-notifyd is 100% opacity no matter what.

# Window type settings

wintypes:

{

  tooltip = { fade = true; shadow = false; };

};

重点是其中 shadow 一项的值设置为 false 。

自启动

最后，修改i3配置文件 ~/.i3/config，在最后加入下面的代码，实现自动启动compton和搜狗输入法

# auto start commands

exec --no-startup-id fcitx -r

exec --no-startup-id fcitx-qimpanel

exec --no-startup-id compton -b

安装完成了，系统也顺便带了半透明和淡入淡出效果。

## 3. 字符设备驱动

### 3.1 基本驱动

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/slab.h>

#include <linux/uaccess.h>

#include <linux/device.h>

#include "demo.h"

#ifndef MEMDEV\_NR\_DEVS

#define MEMDEV\_NR\_DEVS 2 /\*设备数\*/

#endif

#ifndef MEMDEV\_SIZE

#define MEMDEV\_SIZE 4096

#endif

//static struct class \*drv\_class;

//static struct device \*drv\_device;

/\*mem设备描述结构体\*/

struct mem\_dev

{

char \*data;

unsigned long size;

};

/\* seek文件定位函数 \*/

static loff\_t mem\_llseek(struct file \*filp, loff\_t offset, int orig)

{

loff\_t newpos;

switch(orig) {

case 0: /\* SEEK\_SET \*/

newpos = offset;

break;

case 1: /\* SEEK\_CUR \*/

newpos = filp->f\_pos + offset;

break;

case 2: /\* SEEK\_END \*/

newpos = MEMDEV\_SIZE -1 + offset;

break;

default: /\* can't happen \*/

return -EINVAL;

}

if ((newpos<0) || (newpos>MEMDEV\_SIZE))

return -EINVAL;

filp->f\_pos = newpos;

return newpos;

}

struct mem\_dev \*mem\_devp; /\*设备结构体指针\*/

static struct cdev cdev[2];

static dev\_t ndev;

/\*写函数\*/

static ssize\_t mem\_write(struct file \*filp, const char \_\_user \*buf, size\_t size, loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct mem\_dev \*dev = filp->private\_data; /\*获得设备结构体指针\*/

/\*分析和获取有效的写长度\*/

if (p >= MEMDEV\_SIZE)

return 0;

if (count > MEMDEV\_SIZE - p)

count = MEMDEV\_SIZE - p;

/\*从用户空间写入数据\*/

if (copy\_from\_user(dev->data + p, buf, count))

ret = - EFAULT;

else

{

\*ppos += count;

ret = count;

printk("written %d bytes(s) from %zu \n",count,p);

}

return ret;

}

static int mem\_open(struct inode\* nd, struct file\* filp)

{

int major;

int minor;

major = MAJOR(nd->i\_rdev);

minor = MINOR(nd->i\_rdev);

printk("chr\_open, major = %d, minor = %d\n", major, minor);

filp->private\_data = &mem\_devp[minor];

return 0;

}

/\*读函数\*/

static ssize\_t mem\_read(struct file \*filp, char \_\_user \*buf, size\_t size,loff\_t \*ppos)

{

unsigned long p = \*ppos;

unsigned int count = size;

int ret = 0;

struct mem\_dev \*dev = filp->private\_data; /\*获得设备结构体指针\*/

/\*分析获取有效的读长度\*/

if (p >= MEMDEV\_SIZE)/\*当前位置不再文件范围内，报错\*/

return count ? - ENXIO: 0;

if (count > MEMDEV\_SIZE - p)/\*数量超出文件范围，修改读取的个数\*/

count = MEMDEV\_SIZE - p;

/\*复制数据\*/

if (copy\_to\_user(buf, (void\*)(dev->data + p), count))

ret = - EFAULT;/\*发生拷贝错误\*/

else

{

\*ppos += count;/\*修改当前位置指针\*/

ret = count;/\*返回读取的数量\*/

printk(KERN\_INFO "read %d bytes(s) from %d\n", count, p);

}

return ret;

}

/\*文件释放函数\*/

static int mem\_release(struct inode \*inode, struct file \*filp)

{

return 0;

}

struct file\_operations mem\_fops = {

.owner = THIS\_MODULE,

.llseek = mem\_llseek,

.read = mem\_read,

.write = mem\_write,

.open = mem\_open,

.release = mem\_release

};

/\*init func\*/

static int demo\_init(void)

{

int ret,i,devno;

ret = alloc\_chrdev\_region(&ndev, 0, MEMDEV\_NR\_DEVS, "cdev");/\* 动态分配设备号 \*/

if(ret < 0 )

{

return ret;

}

//printk("demo\_init(): major = %d, minor = %d\n", MAJOR(ndev), MINOR(ndev));

/\* 为设备描述结构分配内存\*/

mem\_devp = kmalloc(MEMDEV\_NR\_DEVS \* sizeof(struct mem\_dev), GFP\_KERNEL);//目前为止我们始终用GFP\_KERNEL

if (!mem\_devp) /\*申请失败\*/

{

ret = - ENOMEM;

goto fail\_malloc;

}

memset(mem\_devp, 0, MEMDEV\_NR\_DEVS \*sizeof(struct mem\_dev));

for (i=0; i < MEMDEV\_NR\_DEVS; i++)

{

/\*为设备分配内存\*/

mem\_devp[i].size = MEMDEV\_SIZE;

mem\_devp[i].data = kmalloc(MEMDEV\_SIZE, GFP\_KERNEL);//分配出来的地址存在此

memset(mem\_devp[i].data, 0, MEMDEV\_SIZE);

/\*初始化并注册cdev\*/

devno = MKDEV(MAJOR(ndev),i);

cdev\_init(&cdev[i], &mem\_fops);/\* 使cdev与char\_fops联系起l来 \*/

cdev[i].owner = THIS\_MODULE;

//cdev.ops = &mem\_fops;

ret = cdev\_add(&cdev[i], devno, 1); /\* 注册字符设备 \*/

if(ret < 0)

{

printk(KERN\_NOTICE "Error %d adding mem %d", ret, i);

return ret;

}

printk("demo\_init(): major = %d, minor = %d\n", MAJOR(ndev), MINOR(ndev));

}

// drv\_class = class\_create(THIS\_MODULE, "class\_create"); //创建类

// drv\_device = device\_create(drv\_class, NULL, ndev, NULL, "mem\_dev"); //在类下创建设备，供用户空间的应用程序使用

return 0;

fail\_malloc:

unregister\_chrdev\_region(ndev, 1);

return ret;

}

/\*exit func\*/

static void demo\_exit(void)

{

printk("demo\_exit process!\n");

cdev\_del(&cdev); /\*注销设备\*/

kfree(mem\_devp); /\*释放设备结构体内存\*/

unregister\_chrdev\_region(ndev, MEMDEV\_NR\_DEVS); /\*释放设备号\*/

// device\_unregister(drv\_device); //卸载类下的设备

// class\_destroy(drv\_class); //卸载类

}

module\_init(demo\_init);

module\_exit(demo\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("xxiomg@163.com");

MODULE\_DESCRIPTION("A simple char device example!");

### 3.2 main函数

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

int main()

{

int fd = NULL;

char Buf[45];

/\*初始化Buf\*/

strcpy(Buf,"Mem is char dev0!");

printf("BUF: %s\n",Buf);

/\*打开设备文件\*/

fd = open("/dev/char\_dev0",O\_RDWR);

if (fd == NULL)

{

printf("Open char\_dev0 Error!\n");

return -1;

}

printf("fd number is = %d\n",fd);

/\*写入设备\*/

write(fd, Buf, 40);

/\*重新定位文件位置（思考没有该指令，会有何后果)\*/

lseek(fd,0,SEEK\_SET);

/\*清除Buf\*/

strcpy(Buf,"Buf is NULL!");

printf("BUF: %s\n",Buf);

while(1)

{

/\*读出设备\*/

read(fd,Buf, 40);

/\*检测结果\*/

printf("BUF: %s\n",Buf);

lseek(fd,0,SEEK\_SET);

sleep(1);

}

close(fd);

return 0;

}

### 3.3 Makefile文件

ifneq ($(KERNELRELEASE),)

obj-m := demo.o

else

KVERS:= $(shell uname -r)

KDIR := /lib/modules/$(KVERS)/build

default:

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

clean:

rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers

endif

### 3.4 相关shell指令

1）生成模块：make

2）装在模块：insmod demo.ko

3）卸载模块：rmmod demo

4）生成节点：mknod /dev/char\_dev0 c 246 0

5）生成可执行文件：gcc main.c –o app

6）执行文件 ：./app

### 3.5 相关说明

1）目前节点的添加方式是手动添加，主要是对应相同主设备的设备有两个，目前还未掌握自动添加两个设备节点的方法。

2）识别打开节点的方法是在文件打开函数中读取节点的次设备号，再将操作指针切换到对应的数据区。

3）引申问题，由于不同的设备采用相同的驱动，那么需要对设备资源进行保护访问时也应具有针对性。因此，对于读写等驱动程序，也要根据不同设备设置保护，也就是都要获取到此设备号才行，对此，可以将在打开时获取到的此设备号保存下来，放到设备结构中。

4）Linux中使用较少的全局变量，而是采用全局指针，好处是对于要使用的数据结构，动态进行申请即可，使用完毕之后释放。这样驱动独占的内存资源就比较少，有利于资源的优化利用。