

# 操作系统实验报告

## 实验环境：

- ubuntu 10.04
- linux kernel 2.6.32
- linux kernel 2.6.33

## 实验项目：

1. 实验 1.3 Shell 编程
2. 实验 2.3 内核模块
3. 实验 2.4 系统调用
4. 实验 3.3 Shell 编程实验（进程管理实验）
5. 实验 4.1 观察实验（存储管理实验）
6. 实验 5.1 观察实验（进程通信）
7. 实验 6.3 lo 系统编程实验
8. 实验 6.4 设备驱动程序
9. 实验 7.1 代码分析(文件系统管理实验)

小组成员：许伟林 08211306（6 班）

刘惠庭 08211338（7 班）

## 内容目录

实验 1.3 Shell 编程.....	4
1、实验目的.....	4
2、实验内容.....	4
3、实验原理.....	4
4、实验步骤.....	4
5、实验结果: .....	4
6、人员任务分配.....	5
实验 2.3 内核模块.....	6
1、实验目的.....	6
2、实验内容.....	6
3、实验原理.....	6
4、实验步骤.....	6
5、实验结果及分析.....	9
6、人员任务分配.....	10
实验 2.4 系统调用.....	11
1、实验目的.....	11
2、实验内容.....	11
3、实验原理.....	11
4、实验步骤.....	11
5、实验结果及分析.....	16
6、人员任务分配.....	17
实验 3.3 Shell 编程实验（进程管理实验） .....	18
1、实验目的.....	18
2、实验内容.....	18
3、实验原理.....	18
4、实验步骤.....	18
5、实验结果及分析.....	19
6、人员任务分配.....	19
2、实验内容 2.....	20
3、实验原理.....	20
4、实验步骤.....	20
5、实验结果及分析.....	23
6、人员分配.....	24
实验 4.1 观察实验(存储管理实验).....	25
1、实验目的.....	25
2、实验内容.....	25
3、实验步骤.....	25
4、观测程序源代码.....	25

5、实验结果及分析.....	25
6、人员任务分配.....	31
实验 5.1 观察实验（进程通信） .....	32
1、实验目的与内容.....	32
2、实验原理.....	32
3、实验结果 .....	32
实验 6.3 IO 系统编程实验.....	36
1、实验目的.....	36
2、实验内容.....	36
3、实验原理.....	36
4、实验步骤.....	36
5、实验结果及分析.....	37
6、人员任务分配.....	37
实验 6.4 设备驱动程序.....	38
1、实验目的.....	38
2、实验内容.....	38
3、实验原理.....	38
4、实验步骤.....	38
5、实验结果和分析.....	47
6、人员任务分配.....	48
实验 7.1 代码分析(文件系统管理实验).....	49
1、实验目的.....	49
2、实验内容.....	49
3、实验结果（源代码分析） .....	49

## 实验1.3 Shell编程

### 1、实验目的

通过本实验，了解Linux系统的shell机制，掌握简单的shell编程技巧。

### 2、实验内容

编制简单的Shell程序，该程序在用户登录时自动执行，显示某些提示信息，如“Welcome to Linux”，并在命令提示符中包含当前时间、当前目录和当前用户名等基本信息。

### 3、实验原理

在计算机科学中，Shell 俗称壳（用来区别于核），是指“提供使用者使用界面”的软件（命令解析器）。它类似于DOS下的command.com。它接收用户命令，然后调用相应的应用程序。同时它又是一种程序设计语言。作为命令语言，它交互式解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高阶语言中才具有的控制结构，包括循环和分支。

### 4、实验步骤

#### ● 编写 Shell 脚本

```
#!/bin/bash
#Add the command "xterm -e welcome.sh -hold" in /etc/rc.local
#open a gnome-terminal and enter "./welcome.sh" for test.
a="-----Welcome to Linux-----"
echo $a
echo "Time: " | tr -d '\n' #delete the enter
date #show the data
echo "Path: " | tr -d '\n'
pwd #show the path
echo "User: " | tr -d '\n'
whoami #show the user
echo $a
read -nl var
#wait a type from keyboard to finish the program, for pause.
```

#### ● 在/etc/rc.local添加: "xterm -e welcome.sh -hold 命令行

### 5、实验结果:

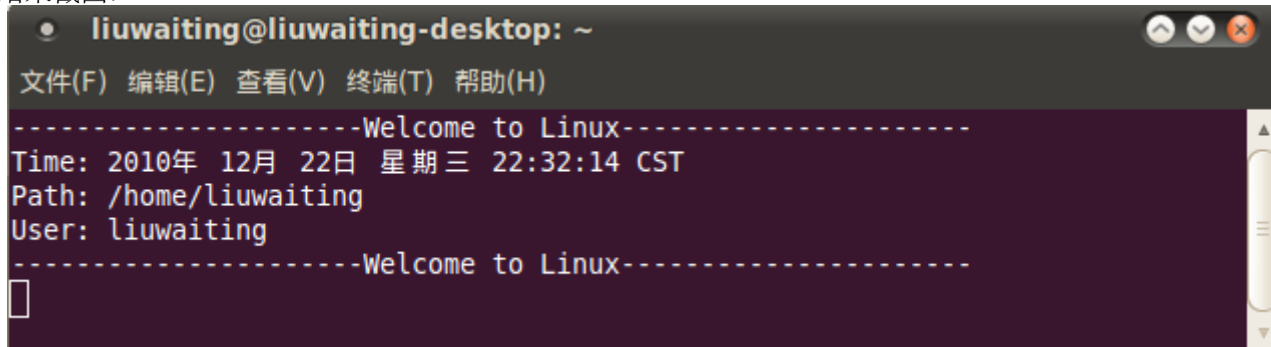
```
-----Welcome to Linux-----
Time: 2010年12月19日 星期日 20:57:13 CST
```

Path: /home/liuwaiting

User: liuwaiting

-----Welcome to Linux-----

结果截图:



## 6、人员任务分配

刘惠庭: 编程设计。

许伟林: 测试、文档。

## 实验2.3内核模块

### 1、实验目的

模块是Linux系统的一种特有机制，可用以动态扩展操作系统内核功能。编写实现某些特定功能的模块，将其作为内核的一部分在管态下运行。本实验通过内核模块编程在/proc文件系统中实现系统时钟的读操作接口。

### 2、实验内容

设计并构建一个在/proc文件系统中的内核模块clock，支持read()操作，read()返回值为一个字符串，其中包块一个空格分开的两个子串，为别代表xtime.tv\_sec和xtime.tv\_usec。

### 3、实验原理

Linux模块是一些可以作为独立程序来编译的函数和数据类型的集合。在装载这些模块式，将它的代码链接到内核中。Linux模块可以在内核启动时装载，也可以在内核运行的过程中装载。如果在模块装载之前就调用了动态模块的一个函数，那么这次调用将会失败。如果这个模块已被加载，那么内核就可以使用系统调用，并将其传递到模块中的相应函数。

### 4、实验步骤

#### ● 编写内核模块

文件中主要包含init\_clock(), exit\_clock(), read\_clock()三个函数。其中init\_clock(), exit\_clock()负责将模块从系统中加载或卸载，以及增加或删除模块在/proc中的入口。read\_clock()负责产生/proc/clock被读时的动作。

#### ● 编译内核模块Makefile文件

```
# Makefile under 2.6.25
ifneq ($(KERNELRELEASE),)
#kbuild syntax. dependency relationship of files and target modules are
listed here.
obj-m := proc_clock.o
else
PWD := $(shell pwd)
KVER ?= $(shell uname -r)
KDIR := /lib/modules/$(KVER)/build
all:
$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions *.symvers *.order
endif
```

编译完成之后生成proc\_clock.ko模块文件。

#### ● 内核模块源代码clock.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
```

```

#include <linux/string.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>

#define MODULE
#define MODULE_VERSION "1.0"
#define MODULE_NAME "clock"

struct proc_dir_entry* my_clock;

int read_clock(char* page, char** start, off_t off, int count, int* eof,
               void* data) {
    int len;
    struct timeval xtime;
    do_gettimeofday(&xtime);
    len = sprintf(page, "%d %d\n", xtime.tv_sec, xtime.tv_usec);
    printk("clock: read_func()\n");
    return len;
}

struct proc_dir_entry *clock_proc_file;

int init_clock(void)
{
    clock_proc_file = create_proc_read_entry("clock", 0, NULL, read_clock, NULL);
    return 0;
}

void exit_clock(void)
{
    remove_proc_entry("clock", clock_proc_file);
}

module_init(init_clock)
module_exit(exit_clock)
MODULE_LICENSE("GPL");

```

## ● 编译内核模块

```
# make
```

```

File Edit View Terminal Help
proc_clock modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.32-26-generic'
  CC [M] /home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.o
/home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.c:9:1: warning: "MODULE"
redefined
<command-line>: warning: this is the location of the previous definition
/home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.c:10:1: warning: "MODULE_
VERSION" redefined
In file included from /home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.c:2
:
include/linux/module.h:162:1: warning: this is the location of the previous defi
nition
/home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.c: In function 'read_cloc
k':
/home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.c:22: warning: format '%d
' expects type 'int', but argument 3 has type '__kernel_time_t'
/home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.c:22: warning: format '%d
' expects type 'int', but argument 4 has type '__kernel_suseconds_t'
Building modules, stage 2.
MODPOST 1 modules
  CC /home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.mod.o
  LD [M] /home/xuweilin/os_experiment/2.3_proc_clock/proc_clock.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.32-26-generic'
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$

```

## ● 加载内核模块

在系统 root 用户下运行用户态模块命令装载内核模块

```

# insmod proc_clock.ko
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$ sudo insmod proc_clock
.ko
[sudo] password for xuweilin:
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$ cat /proc/clock
1293006692 797591
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$

```

## ● 测试

在终端中输入以下命令:

```
# cat /proc/clock
```

或者使用以下测试代码:

```

#include <stdio.h>
#include <sys/time.h>
#include <fcntl.h>
int main(void)
{
    struct timeval getSystemTime;
    char procClockTime[256];
    int infile, len;

```



```

gettimeofday(&getSystemTime,NULL);

infile = open("/proc/clock",O_RDONLY);
len = read(infile,procClockTime,256);
close(infile);

procClockTime[len] = '\0';

printf("SystemTime is %d %d\nProcClockTime is %s\n",
getSystemTime.tv_sec ,
getSystemTime.tv_usec,
procClockTime);

sleep(1);
}

```

#### ● 卸载内核模块

在系统 root 用户下运行用户态模块命令卸载内核模块

```
#rmmod proc_clock.ko
```

## 5、实验结果及分析

在终端中输入以下命令:

```
#cat /proc/clock
```

输出了时间信息:

```
1293006692 797591
```

在终端中运行测试程序:

```
#!/test
```

输出了时间信息:

```
SystemTime is 1293006786 585309
```

```
ProcClockTime is 1293006786 585322
```

```
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$ cat /proc/clock
1293006692 797591
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$ gcc -o test test.c
test.c: In function 'main':
test.c:21: warning: format '%d' expects type 'int', but argument 2 has type '__t
ime_t'
test.c:21: warning: format '%d' expects type 'int', but argument 3 has type '__s
useconds_t'
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$ ./test
SystemTime is 1293006786 585309
ProcClockTime is 1293006786 585322
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$
```

cat 命令使用了文件系统 read() 调用，可以获得 read\_clock() 中的时间字符串信息。

## 6、人员任务分配

许伟林：编程设计。

刘惠庭：测试、文档。

## 实验2.4 系统调用

### 1、实验目的

向现有Linux内核加入一个新的系统调用从而在内核空间中实现对用户空间的读写。

例如，设计并实现一个新的内核函数`mycall()`，此函数通过一个引用参数的调用返回当前系统时间，功能上基本与`gettimeofday()`相同。

也可以实现具有其它功能的系统调用。

### 2、实验内容

在2.6.33.7内核中加入一个新的系统调用`xw1call()`，功能与`gettimeofday()`相同。

### 3、实验原理

linux的系统调用形式与POSIX兼容，也是一套C语言函数名的集合。然而，linux系统调用的内部实现方式却与DOC的INT 21H相似，它是经过INT 0X80H软中断进入后，再根据系统调用号分门别类地服务。

Linux通过`sys_call_table`来管理系统调用号。如果要增加新的系统调用，那不仅要在内核中实现这个系统调用函数，还要在`sys_call_table`中增加这个系统调用的索引。

### 4、实验步骤

#### 1. 添加新调用的源代码

在`./linux-2.6.33.7/arch/x86/kernel/sys_i386_32.c`中添加相应的调用代码

```
asmlinkage int sys_xw1call(struct timeval *tv)
{
    struct timeval ktv;
    do_gettimeofday(&ktv);
    copy_to_user(tv,&ktv,sizeof(ktv));
    printk(KERN_ALERT"PID %1d called sys_xw1call() ./n", (long)current->pid);
    return 0;
}
```

#### 2. 连接系统调用

a、修改`./linux-2.6.33.7/arch/x86/include/asm/unistd_32.h`,

在系统调用列表后面相应位置添加一行，这样在用户空间做系统调用时就不需要知道系统调用号了，如果在用户空间指明了调用号，就可以省略这一步，实际上我就没写：

```
#define __NR_xw1call 338
```

新增加的调用号位338

b、修改`./linux-2.6.33.7/arch/x86/kernel/syscall_table_32.S`

在`ENTRY(sys_call_table)`清单最后添加一行，这步至关重要，338就是这里来的：

```
.long sys_xw1call
```

#### 3. 重建新的Linux内核

先安装好编译内核必要的软件包：

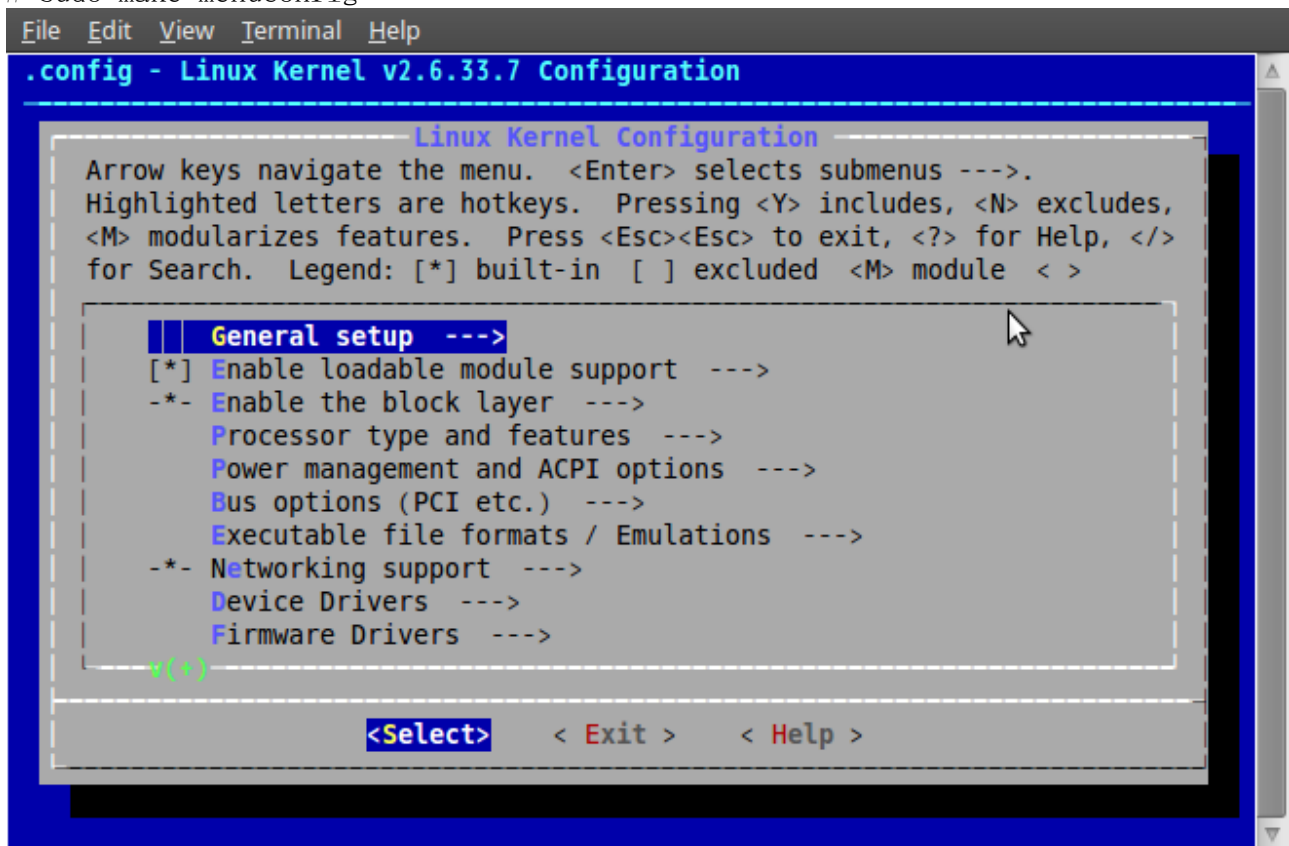
```
# sudo apt-get install build-essential kernel-package libncurses5-dev
```

复制当前内核的配置文件

```
# cp /boot/config-`uname -r` ./config
```

保存配置文件


```
# sudo make menuconfig
```




使用 debian 的的内核编译方法，要简单很多

```
# sudo make-kpkg -initrd --initrd --append-to-version=xwlcall1 kernel_image  
kernel-headers
```

```
===== making target debian/stamp/build/kernel [new prereqs: vars]=====
This is kernel package version 12.032.
restore_upstream_debianization
test ! -f scripts/package/builddeb.kpkg-dist || mv -f scripts/package/builddeb.k
pkg-dist scripts/package/builddeb
test ! -f scripts/package/Makefile.kpkg-dist || mv -f scripts/package/Makefile.k
pkg-dist scripts/package/Makefile
/usr/bin/make EXTRAVERSION=.7xwlcalls ARCH=i386 \
    bzImage
make[1]: Entering directory `/home/xuweilin/交换空间/software/linux-2.6.33.7'
scripts/kconfig/conf -s arch/x86/Kconfig
make[1]: Leaving directory `/home/xuweilin/交换空间/software/linux-2.6.33.7'
make[1]: Entering directory `/home/xuweilin/交换空间/software/linux-2.6.33.7'
  CHK      include/linux/version.h
  CHK      include/generated/utsrelease.h
  CALL     scripts/checksyscalls.sh
  CHK      include/generated/compile.h
  VDSOSYM  arch/x86/vdso/vdso32-int80-syms.lds
  VDSOSYM  arch/x86/vdso/vdso32-sysenter-syms.lds
  VDSOSYM  arch/x86/vdso/vdso32-syms.lds
  LD       arch/x86/vdso/built-in.o
  LD       arch/x86/built-in.o
```



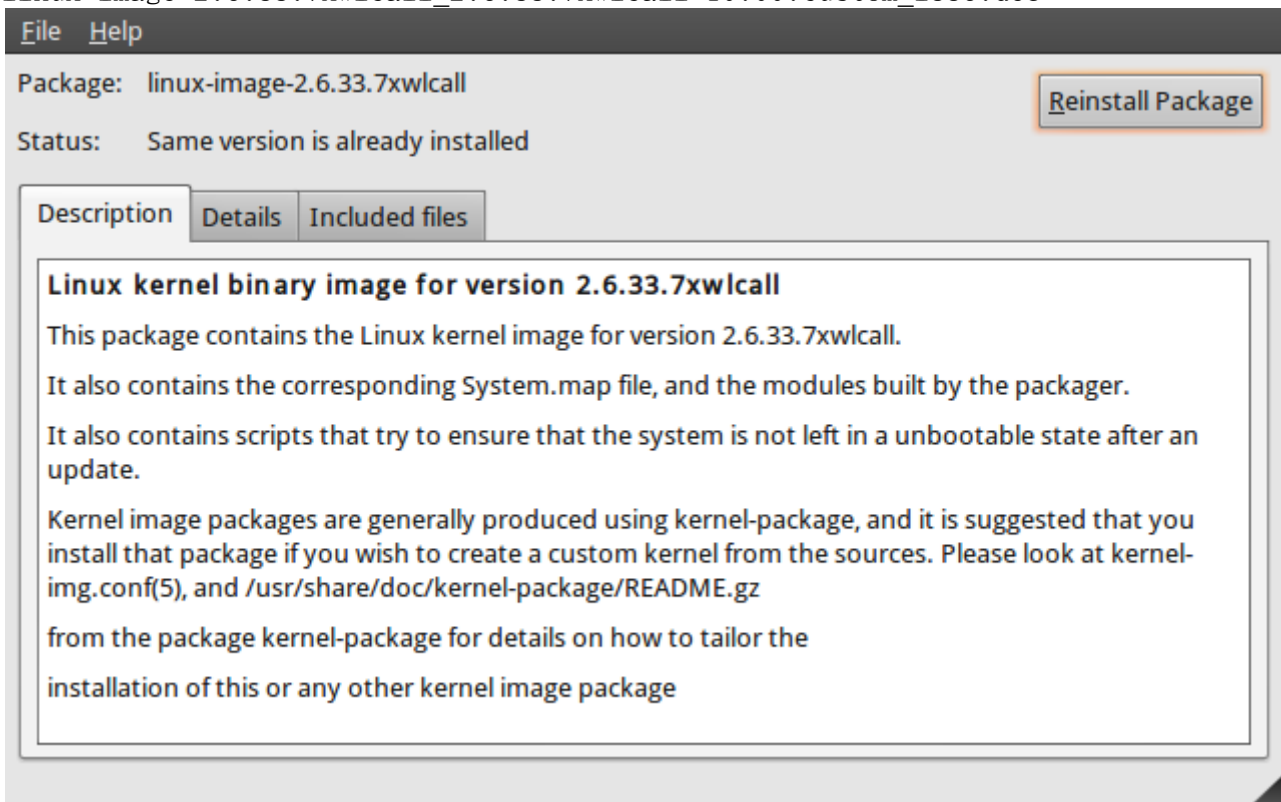
linux-headers-2.6.33.  
7xw1call\_2.6.33.  
7xw1call-10.00.Custo...



linux-image-2.6.33.  
7xw1call\_2.6.33.  
7xw1call-10.00.Custo...

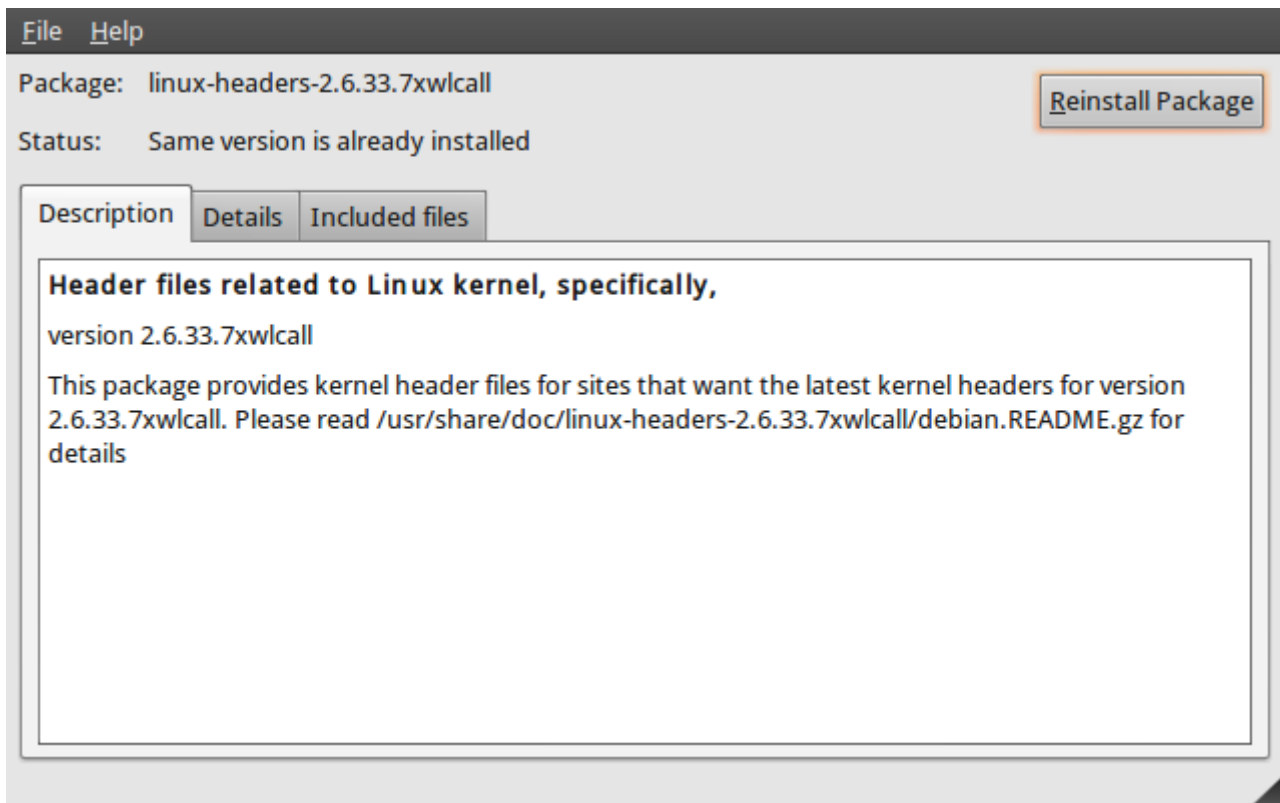
运行以下 deb 包，安装内核镜像和模块：

linux-image-2.6.33.7xw1call\_2.6.33.7xw1call-10.00.Custom\_i386.deb



运行以下 deb 包，安装内核头文件：

linux-headers-2.6.33.7xw1call\_2.6.33.7xw1call-10.00.Custom\_i386.deb



运行以下命令，使内核启动时能调用模块，比如硬件驱动：

```
# sudo update-initramfs -c -k 2.6.33.7xwlcalle
```

此次编译的内核采用 ubuntu 默认配置文件，通用性非常好，可以拷贝到大部分 x86 机器上安装。安装后系统会自动修改 grub 启动选单。

#### 4. 重建引导信息

a、安装上述的 deb 包就自动重建引导信息了，无须另行处理。

b、如果仍然不放心，可以运行

```
# update-grub
```

#### 5. 重新引导从新的内核进入

```
File Edit View Terminal Help
xuweilin@xuweilin-desktop:~$ uname -r
2.6.33.7xwlcalle
xuweilin@xuweilin-desktop:~$
```

#### 6. 修改系统调用表

因为这个系统调用没什么价值，所以不打算去影响全局配置，所有的必要改动都在测试程序代码中进行。

#### 7. 编写程序测试 app\_sys\_call.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```

#include <linux/unistd.h>

#define __NR_xwlcall 338
//_syscall1(int,xwlcall,struct timeval*,thetime);
//这个宏用不了，用下面的函数代替了。
int xwlcall(struct timeval* thetime)
{
    return syscall(__NR_xwlcall,thetime);
}

int main(int argc,char* argv[])
{
    struct timeval gettime;
    struct timeval xwlcalltime;

    gettimeofday(&gettime,NULL);
    xwlcall(&xwlcalltime);

    printf("gettimeofday:%d %d\n",gettime.tv_sec,gettime.tv_usec);
    printf("xwlcall:%d %d\n",xwlcalltime.tv_sec,xwlcalltime.tv_usec);
    return 0;
}

```

使用以下编译命令：

```
gcc -o app_sys_call app_sys_call.c
```

## 5、实验结果及分析

在终端运行以下命令：

```
# ./app_sys_call
```

输出以下信息：

```
gettimeofday:1292740032 93077
```

```
xwlcall:1292740032 93079
```



```
xuweilin@xuweilin-desktop:~/os_experiment/2.4_sys_call$ gcc -o app_sys_call app_sys_call.c
app_sys_call.c: In function 'main':
app_sys_call.c:23: warning: format '%d' expects type 'int', but argument 2 has type '__time_t'
app_sys_call.c:23: warning: format '%d' expects type 'int', but argument 3 has type '__suseconds_t'
app_sys_call.c:24: warning: format '%d' expects type 'int', but argument 2 has type '__time_t'
app_sys_call.c:24: warning: format '%d' expects type 'int', but argument 3 has type '__suseconds_t'
xuweilin@xuweilin-desktop:~/os_experiment/2.4_sys_call$ ./app_sys_call
gettimeofday:1293008350 731323
xwllcall:1293008350 731326
xuweilin@xuweilin-desktop:~/os_experiment/2.4_sys_call$
```

可见 `xwllcall()` 与 `gettimeofday()` 具有同样的功能。

## 6、人员任务分配

许伟林：编程设计。

刘惠庭：测试、文档。

## 实验3.3 Shell编程实验（进程管理实验）

### 1、实验目的

通过编写 shell 程序，了解子进程的创建和父进程与子进程间的协同，获得多进程程序的编程经验。

### 2、实验内容

设计一个简单的 shell 解释程序，能实现基本的 bsh 功能。

### 3、实验原理

将每一条命令分子段压入 argv 栈。然后再子进程中调用 execvp() 来实现该命令的功能。

### 4、实验步骤

编写代码（源代码清单）

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUFFERSIZE 256

//最简单的 shell，只是简单的执行命令调用，没有任何的其他功能
int main()
{
    char buf[BUFFERSIZE], *cmd, *argv[100];
    char inchar;
    int n, sv, buflength;
    int result;
    buflength = 0;
    for(;;) {
        printf("=> ");
//处理过长的命令;
        inchar = getchar(); //读取命令
        while (inchar != '\n' && buflength < BUFFERSIZE) {
            buf[buflength++] = inchar;
            inchar = getchar();
        }
        if (buflength > BUFFERSIZE) {
            printf("Command too long, please enter again!\n");
            buflength = 0;
            continue;
        }
        else
            buf[buflength] = '\0';
//解析命令行，分成一个个的标记
```

```

//char *strtok(char *s,char *delim)
//分解字符串为一组字符串。s 为要分解的字符串，delim 为分隔符字符串。
cmd=strtok(buf, "\t\n");
if(cmd) {
    if(strcmp(cmd,"exit")==0) exit(0);
    n=0;
    argv[n++]=cmd;
    while(argv[n++]=strtok(NULL, "\t\n"));
    if(fork()==0) {
        execvp(cmd,argv);
        fprintf(stderr,"sxh:%s:command not found.\n",buf);//如果子进程顺利
        exit(1);
    }
    wait(&sv);
    buflength = 0;
}
}
}
}

```

执行，这段话是不会执行的

程序编译

## 5、实验结果及分析

liuwaiting@liuwaiting-desktop:~/桌面/OS 实验/进程管理\$ ./xsh

=> ps -a

PID	TTY	TIME	CMD
3327	pts/1	00:00:00	su
3336	pts/1	00:00:00	bash
4309	pts/0	00:00:00	xsh
4310	pts/0	00:00:00	ps

=> pwd

/home/liuwaiting/桌面/OS 实验/进程管理

=> exit

## 6、人员任务分配

刘惠庭：代码编写编译程序

许伟林：测试及文档

## 2、实验内容 2

编写一个带有重定向和管道功能的 Shell

## 3、实验原理

通过 `fork()` 创建子进程，用 `execvp()` 更改子进程代码，用 `wait()` 等待子进程结束。这三个系统调用可以很好地创建多进程。另一方面，编写的 Shell 要实现管道功能，需要用 `pipe()` 创建管道使子进程进行通信。

## 4、实验步骤

编写代码（源代码清单）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#define BUFFERSIZE 256
//具有输入输出重定向的功能 和管道功能
int
main()
{
    char buf[256],*buf2,*cmd,*cmd2,*argv[64],*argv2[64], *infile,*outfile;
    char inchar;
    int n,sv,buflength,fd[2];

    for(;;) {
        buflen = 0;
        printf("=> ");
        inchar = getchar();
        while (inchar != '\n' && buflen < BUFFERSIZE ){
            buf[buflen++] = inchar;
            inchar = getchar();
        }
        if (buflen > BUFFERSIZE){
            fprintf(stderr,"Command too long,please enter again!\n");
            buflen = 0;
            continue;
        }
        else
            buf[buflen] = '\0';

        //检查是否具有管道操作符
        //strstr() 在字符串中查找指定字符串的第一次出现，buf2 指向管道符号前端的命令
        buf2=strstr(buf,"|");
```

```

if(buf2)
    *buf2++='\0';
else {
    //否则查看是否具有重定向的操作符
    infile=strstr(buf,"<");
    outfile=strstr(buf,">");
    if(infile) {
        *infile='\0';
        infile=strtok(infile+1," \t\n");
    }
    if(outfile) {
        *outfile='\0';
        outfile=strtok(outfile+1," \t\n");
    }
}

//解析命令行，分成一个个的标记
cmd=strtok(buf," \t\n");
//执行管道命令
if(buf2){
    if(strcmp(cmd,"exit")==0) exit(0);
    if(!cmd) {
        fprintf(stderr,"Command token error.\n");
        exit(1);
    }

    n=0;
    //管道后端的命令
    argv[n++]=cmd;
    while(argv[n++]=strtok(NULL," \t\n"));
    //管道前端的命令
    cmd2=strtok(buf2," \t\n");
    if(!cmd2) {
        fprintf(stderr,"Command token error.\n");
        exit(1);
    }
    n=0;
    argv2[n++]=cmd2;
    while(argv2[n++]=strtok(NULL," \t\n"));
    pipe(fd);
    if(fork()==0) {
        dup2(fd[0],0); //dup2 复制文件句柄，将fd[0]复制到描述符0。

```



```
}
```

编译程序

## 5、实验结果及分析

```
liuwaiting@liuwaiting-desktop:~/桌面/OS 实验/进程管理$ ./xsh2
=> ps -a | grep bash
  3336 pts/1    00:00:00 bash
=> ls
a.txt  b.txt  xsh  xsh2  xsh2.c  xsh.c
=> ls > c.txt
=> cat c.txt
a.txt
b.txt
c.txt
xsh
xsh2
xsh2.c
xsh.c
=> pwd
/home/liuwaiting/桌面/OS 实验/进程管理
=> exit
liuwaiting@liuwaiting-desktop:~/桌面/OS 实验/进程管理$
```

两次运行结果截图:

```
liuwaiting@liuwaiting-desktop: ~/桌面/OS实验/进程管理
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

liuwaiting@liuwaiting-desktop:~/桌面/OS实验/进程管理$ ./xsh
=> ps -a
  PID TTY          TIME CMD
 3327 pts/1        00:00:00 su
 3336 pts/1        00:00:00 bash
 4309 pts/0      00:00:00 xsh
 4310 pts/0        00:00:00 ps
=> pwd
/home/liuwaiting/桌面/OS实验/进程管理
=> exit
liuwaiting@liuwaiting-desktop:~/桌面/OS实验/进程管理$ ./xsh2
=> ps -a | grep bash
 3336 pts/1        00:00:00 bash
=> ls
a.txt  b.txt  xsh  xsh2  xsh2.c  xsh.c
=> ls > c.txt
=> cat c.txt | grep bash
a.txt 3336 pts/1        00:00:00 bash
b.txt  ls
c.txt txt  b.txt  xsh  xsh2  xsh2.c  xsh.c
xsh => ls > c.txt
xsh2 > cat c.txt
xsh2.c.txt
xsh.c.txt
=> pwd
/home/liuwaiting/桌面/OS实验/进程管理
=> exit
liuwaiting@liuwaiting-desktop:~/桌面/OS实验/进程管理$
```

## 6、人员分配

刘惠庭：代码编写程序编译

许伟林：测试及文档



## 实验4.1 观察实验(存储管理实验)

### 1、实验目的

利用 Linux 相关程序和命令，观察程序结构和进程执行情况。

### 2、实验内容

1. 在 Linux 下，使用 gdb 程序观察一个程序文件的内容和结构。启动该程序执行，再用 GDB 观察其内存映象的内容和结构。

2. 在 Linux 下，用 free 和 vmstat 命令观察内存使用情况。

3. 在 Linux 下，查看/proc 与内存管理相关的文件，并解释显示结果。

### 3、实验步骤

1、安装 GDB

2、编写观测程序

3、按照指令手册进行观察操作

### 4、观测程序源代码

```
#include<stdio.h>
#include<stdlib.h>
char str[50] = "Hello Linux.";
int main()
{
    int num = 10;
    while(num--){
        printf("%s\n",str);
    }
}
```

```
//gcc -g -o testing testing.c
```

### 5、实验结果及分析

#### ● Gdb 程序观察一个程序文件的内容和结构

```
liuwaiting@liuwaiting-desktop:~$ gdb testing
```

```
GNU gdb (GDB) 7.1-ubuntu
```

```
Copyright (C) 2010 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
```

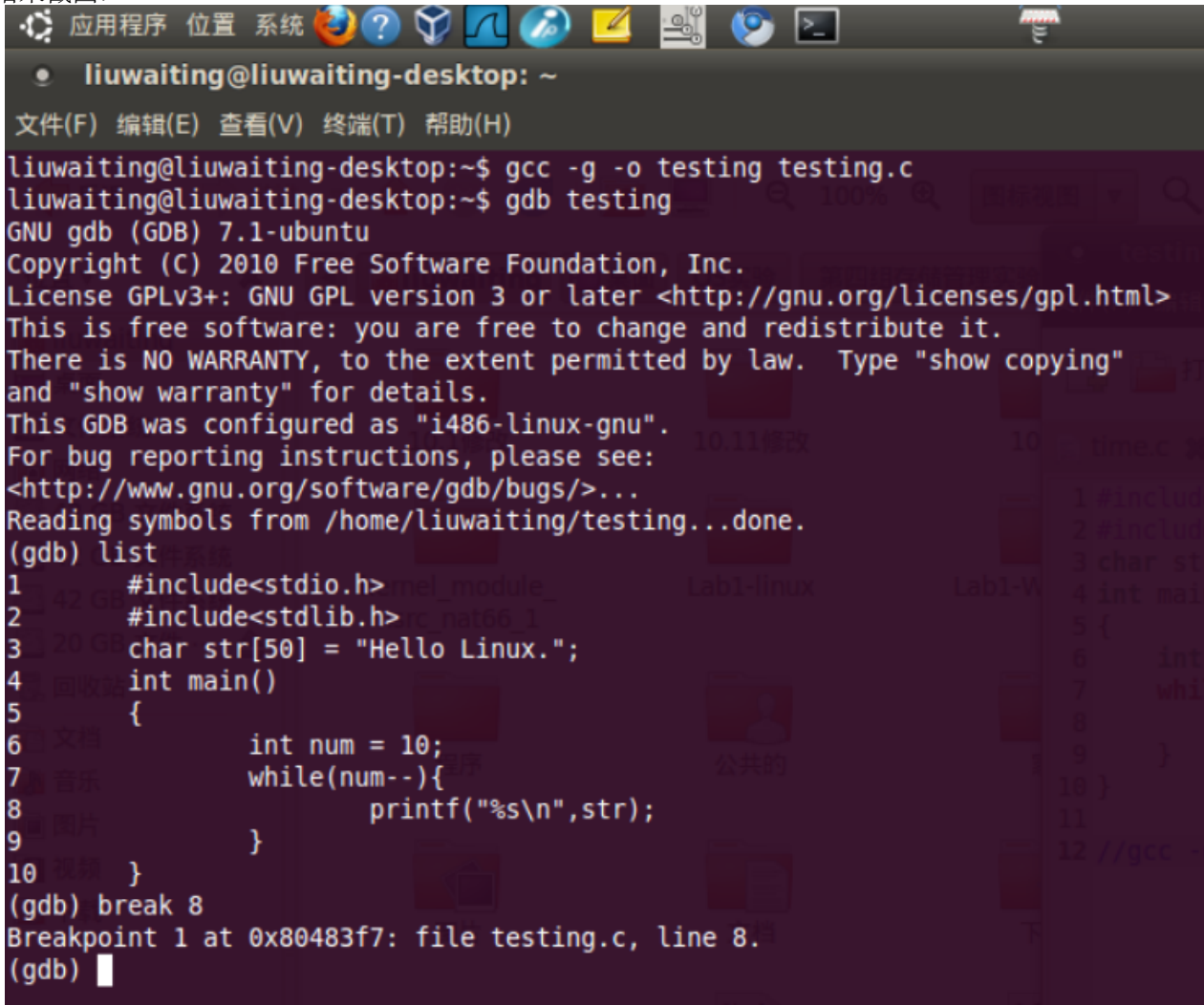
```
This GDB was configured as "i486-linux-gnu".
```

```
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>...
```

Reading symbols from /home/liuwaiting/testing...done.

结果截图:



```
liuwaiting@liuwaiting-desktop: ~  
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)  
liuwaiting@liuwaiting-desktop:~$ gcc -g -o testing testing.c  
liuwaiting@liuwaiting-desktop:~$ gdb testing  
GNU gdb (GDB) 7.1-ubuntu  
Copyright (C) 2010 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "i486-linux-gnu".  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>...  
Reading symbols from /home/liuwaiting/testing...done.  
(gdb) list  
1      #include<stdio.h>  
2      #include<stdlib.h>  
3      char str[50] = "Hello Linux.";  
4      int main()  
5      {  
6          int num = 10;  
7          while(num--){  
8              printf("%s\n",str);  
9          }  
10     }  
(gdb) break 8  
Breakpoint 1 at 0x80483f7: file testing.c, line 8.  
(gdb)
```

## ● GDB 观察程序内存映象的内容和结构

#s 设置断点

```
(gdb) break 8
```

Breakpoint 1 at 0x80483f7: file testing.c, line 8.

#运行程序

```
(gdb) r
```

Starting program: /home/liuwaiting/testing

Breakpoint 1, main () at testing.c:8

```
8      printf("%s\n",str);
```

#输出存储结构的内容

```
(gdb) print /c str
```

```
$1 = {72 'H', 101 'e', 108 'l', 108 'l', 111 'o', 32 ' ', 76 'L', 105 'i',  
110 'n', 117 'u', 120 'x', 46 '.', 0 '\000' <repeats 38 times>}  
(gdb) print /d num  
$2 = 9
```

#查看内存

```
(gdb) x/1cb str  
0x804a040 <str>: 72 'H'  
(gdb) x/1cb str+1  
0x804a041 <str+1>: 101 'e'
```

#反汇编观测内存操作

```
(gdb) disassemble
```

Dump of assembler code for function main:

```
0x080483e4 <+0>: push    %ebp  
0x080483e5 <+1>: mov     %esp,%ebp  
0x080483e7 <+3>: and     $0xffffffff0,%esp  
0x080483ea <+6>: sub     $0x20,%esp  
0x080483ed <+9>: movl    $0xa,0x1c(%esp)  
0x080483f5 <+17>: jmp     0x8048403 <main+31>  
=> 0x080483f7 <+19>: movl    $0x804a040, (%esp)  
0x080483fe <+26>: call    0x8048318 <puts@plt>  
0x08048403 <+31>: cmpl    $0x0,0x1c(%esp)  
0x08048408 <+36>: setne   %a1  
0x0804840b <+39>: subl    $0x1,0x1c(%esp)  
0x08048410 <+44>: test    %a1,%a1  
0x08048412 <+46>: jne     0x80483f7 <main+19>  
0x08048414 <+48>: leave  
0x08048415 <+49>: ret
```

End of assembler dump.

```
(gdb)
```

#观测程序内容和结构

```
(gdb) list  
1  #include<stdio.h>  
2  #include<stdlib.h>  
3  char str[50] = "Hello Linux.";  
4  int main()  
5  {  
6      int num = 10;  
7      while(num--){  
8          printf("%s\n",str);  
9      }
```

10 }

结果截图：

```
(gdb) list
1  #include<stdio.h>
2  #include<stdlib.h>
3  char str[50] = "Hello Linux.";
4  int main()
5  {
6      int num = 10;
7      while(num--){
8          printf("%s\n",str);
9      }
10 }
(gdb) break 8
Breakpoint 1 at 0x80483f7: file testing.c, line 8.
(gdb) r
Starting program: /home/liuwaiting/testing

Breakpoint 1, main () at testing.c:8
8      printf("%s\n",str);
(gdb) print /c str
$1 = {72 'H', 101 'e', 108 'l', 108 'l', 111 'o', 32 ' ', 76 'L', 105 'i',
      110 'n', 117 'u', 120 'x', 46 '.', 0 '\000' <repeats 38 times>}
(gdb) print /d num
$2 = 9
(gdb) x/1cb str
0x804a040 <str>:      72 'H'
(gdb) x/1cb str+1
0x804a041 <str+1>:    101 'e'
(gdb) disassemble
Dump of assembler code for function main:
   0x080483e4 <+0>:      push    %ebp
   0x080483e5 <+1>:      mov     %esp,%ebp
   0x080483e7 <+3>:      and     $0xffffffff0,%esp
   0x080483ea <+6>:      sub     $0x20,%esp
   0x080483ed <+9>:      movl    $0xa,0x1c(%esp)
   0x080483f5 <+17>:     jmp     0x8048403 <main+31>
=> 0x080483f7 <+19>:     movl    $0x804a040, (%esp)
   0x080483fe <+26>:     call   0x8048318 <puts@plt>
   0x08048403 <+31>:     cmpl    $0x0,0x1c(%esp)
   0x08048408 <+36>:     setne   %al
   0x0804840b <+39>:     subl    $0x1,0x1c(%esp)
   0x08048410 <+44>:     test    %al,%al
   0x08048412 <+46>:     jne     0x80483f7 <main+19>
   0x08048414 <+48>:     leave
   0x08048415 <+49>:     ret
End of assembler dump.
(gdb) █
```

## ● 在Linux下，用free和vmstat命令观察内存使用情况

liuwaiting@liuwaiting-desktop:~\$ free

	total	used	free	shared	buffers	cached
Mem:	960636	932304	28332	0	47728	316868
-/+ buffers/cache:		567708	392928			

```

Swap:          623608      159292      464316
liuwaiting@liuwaiting-desktop:~$ vmstat
procs  -----memory-----  ---swap--  -----io----  -system--  ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 0   0 159288  26564  47796 317548    0    3   23   11  483  192 11  3 85  1

```

结果截图:

```

liuwaiting@liuwaiting-desktop: ~
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
liuwaiting@liuwaiting-desktop:~$ free
              total        used        free      shared    buffers     cached
Mem:          960636        673292        287344           0        88592        181596
-/+ buffers/cache:        403104        557532
Swap:          623608        117604        506004
liuwaiting@liuwaiting-desktop:~$ vmstat
procs  -----memory-----  ---swap--  -----io----  -system--  ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 0   0 117604 287220  88600 181604    2   10   102   93  751 1450 16  5 77  2
liuwaiting@liuwaiting-desktop:~$

```

● 在Linux下, 查看/proc与内存管理相关的文件, 并解释显示结果

```
liuwaiting@liuwaiting-desktop:~$ cat /proc/meminfo
```

```

MemTotal:          960636 kB
MemFree:           25464 kB
Buffers:           47812 kB
Cached:           317632 kB
SwapCached:        18444 kB
Active:            435844 kB
Inactive:          376920 kB
Active(anon):      224652 kB
Inactive(anon):    233752 kB
Active(file):       211192 kB
Inactive(file):    143168 kB
Unevictable:         0 kB
Mlocked:            0 kB
HighTotal:          72584 kB
HighFree:           192 kB
LowTotal:          888052 kB
LowFree:           25272 kB
SwapTotal:         623608 kB
SwapFree:          464320 kB
Dirty:              12 kB
Writeback:          0 kB
AnonPages:         432680 kB

```

Mapped:	171452 kB
Shmem:	11084 kB
Slab:	19768 kB
SReclaimable:	8852 kB
SUnreclaim:	10916 kB
KernelStack:	2248 kB
PageTables:	7540 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	1103924 kB
Committed_AS:	1476432 kB
VmallocTotal:	122880 kB
VmallocUsed:	70940 kB
VmallocChunk:	43508 kB
HardwareCorrupted:	0 kB
HugePages_Total:	0
HugePages_Free:	0
HugePages_Rsvd:	0
HugePages_Surp:	0
Hugepagesize:	4096 kB
DirectMap4k:	892920 kB
DirectMap4M:	16384 kB

结果截图:

```
liuwaiting@liuwaiting-desktop: ~  
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)  
liuwaiting@liuwaiting-desktop:~$ cat /proc/meminfo  
MemTotal:          960636 kB  
MemFree:           288100 kB  
Buffers:           88992 kB  
Cached:            181808 kB  
SwapCached:        21844 kB  
Active:            252120 kB  
Inactive:          304524 kB  
Active(anon):       105852 kB  
Inactive(anon):     186644 kB  
Active(file):       146268 kB  
Inactive(file):     117880 kB  
Unevictable:        0 kB  
Mlocked:            0 kB  
HighTotal:         72584 kB  
HighFree:           200 kB  
LowTotal:          888052 kB  
LowFree:           287900 kB  
SwapTotal:         623608 kB  
SwapFree:          506132 kB  
Dirty:             200 kB  
Writeback:          0 kB  
AnonPages:         270468 kB  
Mapped:            73060 kB  
Shmem:             6652 kB  
Slab:              54148 kB  
SReclaimable:      43356 kB  
SUnreclaim:        10792 kB  
KernelStack:       2224 kB  
PageTables:         7136 kB  
NFS_Unstable:       0 kB  
Bounce:            0 kB  
WritebackTmp:       0 kB  
CommitLimit:       1103924 kB  
Committed_AS:      1089640 kB  
VmallocTotal:      122880 kB  
VmallocUsed:        70940 kB  
VmallocChunk:       44532 kB  
HardwareCorrupted:  0 kB  
HugePages_Total:    0  
HugePages_Free:     0  
HugePages_Rsvd:     0  
HugePages_Surp:     0  
Hugepagesize:       4096 kB  
DirectMap4k:        892920 kB  
DirectMap4M:        16384 kB
```

## 6、人员任务分配

刘惠庭：测试程序编写及观测操作

许伟林：文档

## 实验5.1 观察实验（进程通信）

### 1、实验目的与内容

在Linux下，用 `ipcs()` 命令观察进程通信情况，了解Linux基本通信机制。

### 2、实验原理

Linux IPC 继承了 Unix System V 及 DSD 等，共有6种机制：信号(signal)、管道(pipe 和命名管道(named pipe))、消息队列 (message queues)、共享内存 (shared memory segments)、信号量 (semaphore)、套接字 (socket)。

本实验中用到的几种进程间通信方式：

#### (1) 共享内存段 (shared memory segments) 方式

- 将2个进程的虚拟地址映射到同一内存物理地址，实现内存共享
- 对共享内存的访问同步需由用户进程自身或其它IPC机制实现（如信号量）
- 用户空间内实现，访问速度最快。
- Linux 利用 `shmid_ds` 结构描述所有的共享内存对象。

#### (2) 信号量 (semaphore) 方式

- 实现进程间的同步与互斥
- P/V 操作，Signal/wait 操作
- Linux 利用 `semid_ds` 结构表示 IPC 信号量

#### (3) 消息队列 (message queues) 方式

- 消息组成的链表，进程可从中读写消息。
- Linux 维护消息队列向量表 `msgque`，向量表中的每个元素都有一个指向 `msqid_ds` 结构的指针，每个 `msqid_ds` 结构完整描述一个消息队列

LINUX 系统提供的 IPC 函数有：

- `msgget`(关键字，方式)：创建或打开一个消息队列
- `msgsnd`(消息队列标志符，消息体指针，消息体大小，消息类型)：向队列传递消息
- `msgrcv`(消息队列标志符，消息体指针，消息体大小，消息类型)：从队列中取消息
- `msgctl`(消息队列标志符，获取/设置/删除，`mqid_ds` 缓冲区指针)：获取或设置某个队列信息，或删除某消息队列

Linux 系统中，内核，I/O 任务，服务器进程和用户进程之间采用消息队列方式，许多微内核 OS 中，内核和各组件间的基本通信也采用消息队列方式。

### 3、实验结果

```
xuweilin@xuweilin-desktop:~$ ipcs
```

```
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000   131072     xuweilin   600         393216     2          dest
0x00000000   163841     xuweilin   600         393216     2          dest
0x00000000   196610     xuweilin   600         393216     2          dest
0x00000000   229379     xuweilin   600         393216     2          dest
0x00000000   262148     xuweilin   600         393216     2          dest
0x00000000   294917     xuweilin   600         393216     2          dest
```



0x00000000	327686	xuweilin	600	393216	2	dest
0x00000000	360455	xuweilin	600	393216	2	dest
0x00000000	393224	xuweilin	600	393216	2	dest
0x00000000	425993	xuweilin	600	393216	2	dest
0x00000000	458762	xuweilin	600	393216	2	dest
0x00000000	491531	xuweilin	600	393216	2	dest
0x00000000	524300	xuweilin	600	393216	2	dest
0x00000000	557069	xuweilin	600	393216	2	dest
0x00000000	589838	xuweilin	600	393216	2	dest
0x00000000	622607	xuweilin	600	393216	2	dest
0x00000000	11960336	xuweilin	600	393216	2	dest
0x00000000	11993105	xuweilin	600	393216	2	dest
0x00000000	5603346	xuweilin	600	393216	2	dest
0x00000000	12025875	xuweilin	600	393216	2	dest
0x00000000	2457620	xuweilin	600	393216	2	dest
0x00000000	12058645	xuweilin	600	393216	2	dest
0x00000000	884758	xuweilin	600	393216	2	dest
0x00000000	917527	xuweilin	600	393216	2	dest
0x00000000	1015832	xuweilin	600	393216	2	dest
0x00000000	1048601	xuweilin	600	393216	2	dest
0x00000000	1146906	xuweilin	600	393216	2	dest
0x00000000	1179675	xuweilin	600	393216	2	dest
0x00000000	6783004	xuweilin	600	393216	2	dest
0x00000000	2555933	xuweilin	600	393216	2	dest
0x00000000	2588702	xuweilin	600	393216	2	dest

----- Semaphore Arrays -----

key	semid	owner	perms	nsems
0x56000000	0	xuweilin	600	1
0xffffffff	32769	xuweilin	600	8

----- Message Queues -----

key	msqid	owner	perms	used-bytes	messages
-----	-------	-------	-------	------------	----------

xuweilin@xuweilin-desktop:~\$



xuweilin@xuweilin-desktop:~\$ ipcs

## ----- Shared Memory Segments -----

key	shmid	owner	perms	bytes	nattch	status
0x00000000	131072	xuweilin	600	393216	2	dest
0x00000000	163841	xuweilin	600	393216	2	dest
0x00000000	196610	xuweilin	600	393216	2	dest
0x00000000	229379	xuweilin	600	393216	2	dest
0x00000000	262148	xuweilin	600	393216	2	dest
0x00000000	294917	xuweilin	600	393216	2	dest
0x00000000	327686	xuweilin	600	393216	2	dest
0x00000000	360455	xuweilin	600	393216	2	dest
0x00000000	393224	xuweilin	600	393216	2	dest
0x00000000	425993	xuweilin	600	393216	2	dest
0x00000000	458762	xuweilin	600	393216	2	dest
0x00000000	491531	xuweilin	600	393216	2	dest
0x00000000	524300	xuweilin	600	393216	2	dest
0x00000000	557069	xuweilin	600	393216	2	dest
0x00000000	589838	xuweilin	600	393216	2	dest
0x00000000	622607	xuweilin	600	393216	2	dest
0x00000000	11960336	xuweilin	600	393216	2	dest
0x00000000	11993105	xuweilin	600	393216	2	dest
0x00000000	5603346	xuweilin	600	393216	2	dest
0x00000000	12025875	xuweilin	600	393216	2	dest

## 实验6.3 IO系统编程实验

### 1、实验目的

编写一个 daemon 进程，该进程定时执行 ps 命令，然后将该命令的输出写至文件 F1 尾部。通过此实验，掌握 Linux I/O 系统相关内容。

### 2、实验内容

编写一个 daemon 进程，该进程定时执行 ps 命令，然后将该命令的输出写至文件 F1 尾部。

### 3、实验原理

在这个程序中，首先 fork 一个子程序，然后，关闭父进程，这样，新生成的子进程被交给 init 进程接管，并在后台执行。

新生成的子进程里，使用 system 系统调用，将 ps 的输出重定向，输入到 f1.txt 里面。

### 4、实验步骤

编写 daemon.c

代码如下：

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

int main(int argc,char* argv[])
{
    int i,p;
    p = fork();
    if(p > 0){
        exit(0);
    }
    else if(p == 0){
        for(i = 0; i < 100; i++) {
            sleep(100);
            system("ps > f1.txt");
        }
    }
    else{
        perror("Create new process!");
    }
    return 1;
}
}
```

编译程序

```
# gcc -o daemon daemon.c
```

执行程序

```
# ./daemon
```

```
xuweilin@xuweilin-desktop:~/os_experiment/2.3_proc_clock$ cd ../6.3_daemon/  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ ls  
daemon  daemon.c  f1.txt  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ gcc -o daemon daemon.c  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ rm f1.txt  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ ls  
daemon  daemon.c  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ ./daemon  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$
```

## 5、实验结果及分析

程序 `sleep(100)` 后会在当前目录生成一个文件 `f1.txt`，内容如下：

PID	TTY	TIME	CMD
1258	pts/0	00:00:00	bash
2729	pts/0	00:00:00	daemon
2801	pts/0	00:00:00	sh
2802	pts/0	00:00:00	ps

```
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ ./daemon  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ ls  
daemon  daemon.c  f1.txt  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$ cat f1.txt  
PID TTY      TIME CMD  
3431 pts/0      00:00:00 bash  
3902 pts/0      00:00:00 daemon  
3934 pts/0      00:00:00 sh  
3935 pts/0      00:00:00 ps  
xuweilin@xuweilin-desktop:~/os_experiment/6.3_daemon$
```

再 `sleep(100)`，此文件会更新。重复执行 100 次。

## 6、人员任务分配

许伟林：编程设计。

刘惠庭：测试、文档。

## 实验6.4 设备驱动程序

### 1、实验目的

了解Linux的设备驱动程序的组织结构和设备管理机制，编写简单的字符设备和块设备驱动程序。

### 2、实验内容

编写字符型设备驱动程序，该字符设备包括6个基操作：

`scull_open()`、`scull_write()`、`scull_read`、`scull_ioctl`、`scull_release()`、`scull_llseek`，同时还需要编写一个测试程序。

### 3、实验原理

Linux下的设备驱动程序被组织为一组完成不同任务的函数的集合，通过这些函数使得linux的设备操作犹如文件一般。在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作，如`open()`、`close()`、`read()`、`write()`等。

Linux主要将设备分为二类：字符设备和块设备。字符设备是指设备发送和接收数据以字符的形式进行；而块设备则以整个数据缓冲区的形式进行。字符设备的驱动相对比较简单，本次实验即实现了一个简单的字符设备驱动。

这是一个非常简单的虚拟字符设备：这个设备中只有一个256字节的存储空间 `static u8 demoBuffer[256]`，这个设备的名字叫做"`xwl`"。

驱动程序是内核的一部分，因此我们需要把程序写成内核模块。程序中要

- 1 指明设备号
- 2 编写设备操作函数并将函数指针以文件操作结构的形式赋给 `cdev->cdev.ops`
- 3 完成其他设备初始化和清除工作

### 4、实验步骤

`xwl` 字符设备驱动程序包含三个文件：`Makefile`、`cdev.c`、`cdev.h`。测试文件 `test.c`。

**Makefile** 代码如下：

```
# Makefile under 2.6.25
ifneq ($(KERNELRELEASE),)
#kbuild syntax. dependency relationship of files and target modules are listed
here.
obj-m := cdev.o
else
PWD := $(shell pwd)
KVER ?= $(shell uname -r)
KDIR := /lib/modules/$(KVER)/build
all:
$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions *.symvers *.order
endif
```

cdev.c 代码如下:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/cdev.h>
#include <linux/version.h>
#include <linux/vmalloc.h>
#include <linux/ctype.h>
#include <linux/pagemap.h>

#include "cdev.h"

MODULE_AUTHOR("xw1");
MODULE_LICENSE("GPL");

struct DEMO_dev *DEMO_devices; //设备结构的指针
static unsigned char demo_inc=0; //计数 记录此设备被打开的次数
static u8 demoBuffer[256];

int scull_open(struct inode *inode, struct file *filp)
{
    struct DEMO_dev *dev; //不允许设备被同时打开多次, 这是致命的!
    if(demo_inc>0) return -ERESTARTSYS;
    demo_inc++;
    /*container_of 宏 通过结构中的某个变量获取结构本身的指针*/
    dev = container_of(inode->i_cdev, struct DEMO_dev, cdev);
    filp->private_data = dev;

    return 0;
}

int scull_release(struct inode *inode, struct file *filp)
{
    demo_inc--;
    return 0;
}

ssize_t scull_read(struct file *filp, char __user *buf, size_t count, loff_t
*f_pos)
```

```

{
    int result;
    loff_t pos= *f_pos; /* 文件的读写位置 */
    if(pos>=256)
    {
        result=0;
        goto out;
    }
    if(count>(256-pos))
    {
        count=256-pos;
    }
    pos += count;

    if (copy_to_user(buf,demoBuffer+*f_pos,count))
    {
        count=-EFAULT; /* 把数据写到用户空间 */
        goto out;
    }

    *f_pos = pos; /* 改变文件的读写位置 */
out:
    return count;
}

ssize_t scull_write(struct file *filp, const char __user *buf, size_t
count,loff_t *f_pos)
{
    ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
    loff_t pos= *f_pos;
    if(pos>=256)
    {
        goto out;
    }
    //如果要写入的输入大于剩下的内存空间 就只写入剩下的空间数量 以防溢出
    if(count>(256-pos))
    {
        count=256-pos;
    }

    pos += count;
    //将数据从用户空间复制到内核空间

```



```

    if (copy_from_user(demoBuffer+*f_pos, buf, count)) {
        retval = -EFAULT;
        goto out;
    }

    *f_pos = pos;
    return count;
out:
    return retval;
}

int scull_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg)
{
    if(cmd==COMMAND1)
    {
        printk("ioctl command1 successfully\n");
        return 0;
    }
    if(cmd==COMMAND2)
    {
        printk("ioctl command2 successfully\n");
        return 0;
    }

    printk("ioctl error\n");
    return -EFAULT;
}

//llseek 实现随机存取
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    loff_t pos;
    pos = filp->f_pos;
    switch (whence)
    {
    case 0:
        pos = off;
        break;
    case 1:
        pos += off;

```

```

        break;
case 2:
default:
    return -EINVAL;
}

if ((pos>256) || (pos<0))
{
    return -EINVAL;
}

return filp->f_pos=pos;
}

//file_operations 定义了设备操作有关的函数指针
struct file_operations DEMO_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};

/*****
MODULE ROUTINE
*****/

void scull_cleanup_module(void)
{
    dev_t devno = MKDEV(DEMO_MAJOR, DEMO_MINOR);

    if (DEMO_devices)
    {
        cdev_del(&DEMO_devices->cdev); //删除设备
        kfree(DEMO_devices); //释放空间
    }
    //释放分配的设备号
    unregister_chrdev_region(devno,1);
}

```

```

int scull_init_module(void)
{
    int result;
    // 在内核中, dev_t 类型 (在<linux/types.h>中定义) 用来保存设备编号——包括主设备号和次设备号
    dev_t dev = 0;

    /*内核中定义了三个宏来处理主、次设备号: MAJOR和 MINOR宏可以从 16 位数中提取出主、次设备号,
    而 MKDEV宏可以把主、此号合并为一个 16 位数。
    高 8 位用于主设备号, 低 8 位用于次设备号。
    http://www.kerneltravel.net/kernel-book/%E7%AC%AC%E5%8D%81%E4%B8%80%E7%AB
    %A0%20%20%E8%AE%BE%E5%A4%87%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F/11.2.3.htm*/

    dev = MKDEV(DEMO_MAJOR, DEMO_MINOR);

    /*获取一个或多个设备编号来使用
    如果分配成功进行, register_chrdev_region 的返回值是 0. 出错的情况下, 返回一个负的错误码,
    你不能存取请求的区域. */

    result = register_chrdev_region(dev, 1, "DEMO");
    if (result < 0)
    {
        printk(KERN_WARNING "DEMO: can't get major %d\n", DEMO_MAJOR);
        return result;
    }
    //为自定义的设备结构申请空间
    DEMO_devices = kmalloc(sizeof(struct DEMO_dev), GFP_KERNEL);
    if (!DEMO_devices)
    {
        result = -ENOMEM;
        goto fail;
    }
    memset(DEMO_devices, 0, sizeof(struct DEMO_dev));
    //初始化一个字符驱动
    cdev_init(&DEMO_devices->cdev, &DEMO_fops);
    DEMO_devices->cdev.owner = THIS_MODULE;
    DEMO_devices->cdev.ops = &DEMO_fops;
    //在内核中添加字符驱动
    result = cdev_add (&DEMO_devices->cdev, dev, 1);
    if(result)
    {
        printk(KERN_NOTICE "Error %d adding DEMO\n", result);
    }
}

```

```

        goto fail;
    }

    return 0;

fail:
    scull_cleanup_module();
    return result;
}

module_init(scull_init_module);
module_exit(scull_cleanup_module);

```

**cdev.h** 代码如下:

```

#ifndef _DEMO_H_
#define _DEMO_H_

#include <linux/ioctl.h> /* needed for the _IOW etc stuff used later */

/*****
 * Macros to help debugging
 *****/

#undef PDEBUG          /* undef it, just in case */
#ifdef DEMO_DEBUG
#ifdef __KERNEL__
#   define PDEBUG(fmt, args...) printk( KERN_DEBUG "DEMO: " fmt, ## args)
#else//usr space
#   define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#endif
#else
#   define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */

//设备号和 ioctl 命令
#define DEMO_MAJOR 50
#define DEMO_MINOR 0
#define COMMAND1 1
#define COMMAND2 2

```

```

//自己定义的设备结构
struct DEMO_dev
{
    struct cdev cdev;    /* Char device structure */
};

ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                  loff_t *f_pos);
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
                   loff_t *f_pos);
loff_t scull_llseek(struct file *filp, loff_t off, int whence);
int      scull_ioctl(struct inode *inode, struct file *filp,
                    unsigned int cmd, unsigned long arg);

#endif /* _DEMO_H_ */

```

测试文件 test.c 源代码:

```

#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<linux/rtc.h>
#include<linux/ioctl.h>
#include<stdio.h>
#include<stdlib.h>

#define COMMAND1 1
#define COMMAND2 2

int main(int argc, char* argv[])
{
    int fd;
    int i;
    char data[256];

    int retval;
    fd=open("/dev/xw1", O_RDWR);
    if(fd==-1)
    {
        perror("error open\n");
    }
}

```

```

    exit(-1);
}
printf("open /dev/xwl successfully\n");
retval=ioctl(fd,COMMAND1,0);
if(retval==-1)
{
    perror("ioctl error\n");
    exit(-1);
}
printf("send command1 successfully\n");

retval=write(fd,"xwl_os_experiment\0",18);
if(retval==-1)
{
    perror("write error\n");
    exit(-1);
}
retval=lseek(fd,0,0);
if(retval==-1)
{
    perror("lseek error\n");
    exit(-1);
}
retval=read(fd,data,18);

if(retval==-1)
{
    perror("read error\n");
    exit(-1);
}
printf("read successfully:%s\n",data);
close(fd);
return 1;
}

```

**编译工程:**

```
# make
```

```
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ make clean
rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions *.symvers *.order
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ ls
cdev.c cdev.h Makefile test test.c
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ make
make -C /lib/modules/2.6.32-26-generic/build M=/home/xuweilin/os_experiment/6.4_char_dev modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.32-26-generic'
  CC [M]  /home/xuweilin/os_experiment/6.4_char_dev/cdev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/xuweilin/os_experiment/6.4_char_dev/cdev.mod.o
  LD [M]  /home/xuweilin/os_experiment/6.4_char_dev/cdev.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.32-26-generic'
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$
```

产生内核模块文件 cdev.ko。

加载模块：

# sudo insmod cdev.ko。

```
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ sudo insmod cdev.ko
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ lsmod | grep cdev
cdev                2469  0
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$
```

创建设备节点：

# mknod /dev/xwl c 50 0

```
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ mknod /dev/xwl c 50 0
mknod: `/dev/xwl': Permission denied
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ sudo mknod /dev/xwl c 50 0
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ ls /dev | grep xwl
xwl
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$
```

编译测试文件：

# gcc -o test test.c

## 5、实验结果和分析

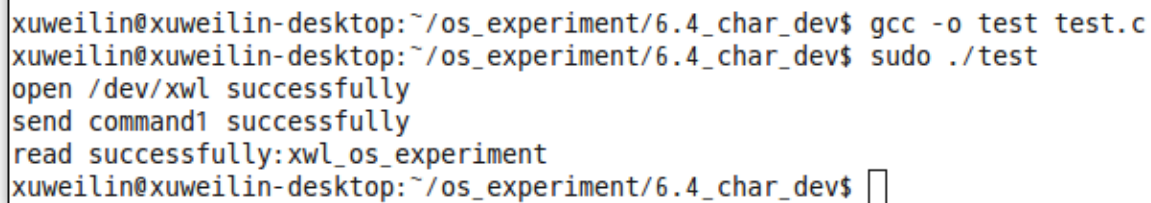
运行测试程序：

# sudo ./test

执行效果：

```
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ sudo ./test
[sudo] password for xuweilin:
open /dev/xwl successfully
```

```
send command1 successfully
read successfully:xwl_os_experiment
```

A terminal window showing the execution of a C program. The user is at the prompt 'xuweilin@xuweilin-desktop:~/os\_experiment/6.4\_char\_dev\$'. They run 'gcc -o test test.c', then 'sudo ./test'. The program outputs 'open /dev/xwl successfully', 'send command1 successfully', and 'read successfully:xwl\_os\_experiment'. The prompt returns.

```
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ gcc -o test test.c
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ sudo ./test
open /dev/xwl successfully
send command1 successfully
read successfully:xwl_os_experiment
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$
```

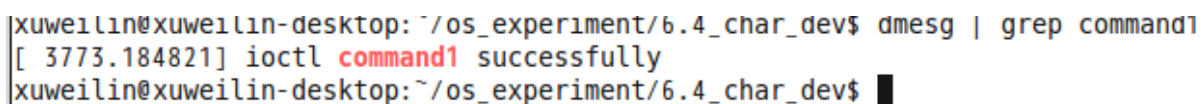
**分析：**在程序中，使用文件系统 API 对 xwl 字符设备进行操作。首先打开 /dev/xwl 文件，在文件开始处写入了一个字符串，然后用 lseek 移动文件指针到开始处，再从设备中读出字符串并输出到屏幕。

**查看内核输出信息：**

```
# dmesg
```

显示有如下信息：

```
[ 3773.184821] ioctl command1 successfully
```

A terminal window showing the output of the command 'dmesg | grep command1'. The output is '[ 3773.184821] ioctl command1 successfully'. The prompt returns.

```
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$ dmesg | grep command1
[ 3773.184821] ioctl command1 successfully
xuweilin@xuweilin-desktop:~/os_experiment/6.4_char_dev$
```

**分析：**ioctl 得到设备响应。

## 6、人员任务分配

许伟林：编码、设计。

刘惠庭：测试、文档。



## 实验7.1 代码分析(文件系统管理实验)

### 1、实验目的

了解与文件管理有关的 Linux 内核模块的代码结构。

### 2、实验内容

阅读 Linux/Minix 中有关文件模块的调用主线,并写出分析报告, 包括  
文件建立模块, 即系统调用 create()  
文件删除模块, 即系统调用 rm()  
读/写模块, 即 read/write

### 3、实验结果 (源代码分析)

#### A. 创建文件模块分析

```
5780  /*creat system call */
5781  Creat()
5782  {
5783      resister *ip;
5784      extern uchar;
5785
5786      ip = namei(&uchar,1);
5787      if(ip == NULL){
5788          if(u.u_error)
5789              return;
5790          ip = maknode(u.u_arg[1]&07777&(~ISVTX));
5791          if (ip == NULL)
5792              return;
5793          openl(ip,FWRITE,2);
5794      }else
5795          openl(ip,FWRITE,1);
5796  }
```

第 5 7 8 6: “namei” ( 7 5 1 8 ) 将一路径名变换成一个 “inode” 指针。“uchar” 是一个过程的名字, 它从用户程序数据区一个字符一个字符地取得文件路径名。

5 7 8 7: 一个空 “inode” 指针表示出了一个错, 或者并没有具有给定路径名的文件存在。

5 7 8 8: 对于出错的各种条件, 请见 U P M 的 C R E A T ( I I )。

5 7 9 0: “maknode” ( 7 4 5 5 ) 调用 “ialloc” 创建一内存 “inode”, 然后对其赋初值, 并使其进入适当的目录。注意, 显式地清除了 “粘住” 位( I S V T X )。

#### B. 删除文件 rm 模块分析

```
3510 unlink()
3511 {
3512     resister *ip,*pp;
```

```

3513 extern uchar;
3514
3515 pp = namei(&uchar,2);
3516 if (pp ==NULL)
3517     return;
3518 prele(pp);
3519 ip = iset(pp ->dev,u.u_dent.u_ino);
3520 if (ip == NULL)
3521     panic (*unlink — iset *);
3522 if ((ip ->i_mode%IFMT) == IFDIR && !suser())
3523     goto out;
3524 u.u_offset[1] = - DIRSIZ+2;
3525 u.ubase = &u.u_dent;
3526 u.ucount = DIRSIZE +2;
3527 u.u_dent.u_ino = 0;
3528 writei(pp);
3529 ip ->i_nlink--;
3530 ip->i_flag =! IUPD;
3531
3532 out:
3533 input(pp);
3534 input(ip);
3535 }

```

新文件作为永久文件自动进入文件目录。关闭文件不会自动地造成文件被删除。当内存“inode”项中的“i\_nlink”字段值为0并且相应文件未被打开时，将删除该文件。在创建文件时，该字段由“makeinode”赋初值为1。系统调用“link”(5941)可将其值加1，系统调用“unlink”(3529)则可将其值减1。创建临时“工作文件”的程序应当在其终止前执行“unlink”系统调用将这些文件删除。

注意，“unlink”系统调用本身并没有删除文件。当引用计数(i\_count)被减为0时(7350、7362)，才删除该文件。

为了减少在程序或系统崩溃时遗留下来的临时文件所带来的问题，程序员应当遵守下列约定：

- (1) 在打开临时文件后立即对其执行“unlink”操作。
- (2) 应在“tmp”目录下创建临时文件。在文件名中包括进程标识数就可构成一惟一文件名

## C 读/写模块，即 read/write 分析

```

/*
 * linux/fs/minix/file.c
 *
 * Copyright (C) 1991, 1992 Linus Torvalds
 *
 * minix regular file handling primitives

```

```

*/

#include <asm/segment.h>
#include <asm/system.h>
#include <linux/sched.h>
#include <linux/minix_fs.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/stat.h>
#include <linux/locks.h>

#define NBUF 32
#define MIN(a,b) (((a)>(b))?(a):(b))
#include
#include

static int ext2_file_read (struct inode *, struct file *, char *, int);
static int ext2_file_write (struct inode *, struct file *, char *, int);
static void ext2_release_file (struct inode *, struct file *);
/*
 * We have mostly NULL's here: the current defaults are ok for
 * the ext2 filesystem.
 */
static struct file_operations ext2_file_operations = {
NULL, /* lseek - default */
ext2_file_read, /* read */
ext2_file_write, /* write */
NULL, /* readdir - bad */
NULL, /* select - default */
ext2_ioctl, /* ioctl */
generic_mmap, /* mmap */
NULL, /* no special open is needed */
ext2_release_file, /* release */
ext2_sync_file /* fsync */
};

struct inode_operations ext2_file_inode_operations = {
&ext2_file_operations, /* default file operations */
NULL, /* create */
NULL, /* lookup */
NULL, /* link */
NULL, /* unlink */
NULL, /* symlink */

```

```

NULL,    /* mkdir */
NULL,    /* rmdir */
NULL,    /* mknod */
NULL,    /* rename */
NULL,    /* readlink */
NULL,    /* follow_link */
ext2_bmap, /* bmap */
ext2_truncate, /* truncate */
ext2_permission /* permission */
};

static int ext2_file_read (struct inode * inode, struct file * filp,
                           char * buf, int count)
{
    int read, left, chars;
    int block, blocks, offset;
    int bhrequest, uptodate;
    struct buffer_head ** bhb, ** bhe;
    struct buffer_head * bhreq[NBUF];
    struct buffer_head * buflist[NBUF];
    struct super_block * sb;
    unsigned int size;
    int err;
    if (!inode) {
        printk ("ext2_file_read: inode = NULL\n");
        return -EINVAL;
    }
    sb = inode->i_sb;
    if (!S_ISREG(inode->i_mode)) {
        ext2_warning (sb, "ext2_file_read", "mode = %07o",
                     inode->i_mode);
        return -EINVAL;
    }
    offset = filp->f_pos;
    size = inode->i_size;
    if (offset > size)
        left = 0;
    else
        left = size - offset;
    if (left > count)
        left = count;
    if (left > EXT2_BLOCK_SIZE_BITS(sb);
    offset &= (sb->s_blocksize - 1);

```

```

size = (size + sb->s_blocksize - 1) >> EXT2_BLOCK_SIZE_BITS(sb);
blocks = (left + offset + sb->s_blocksize - 1) >> EXT2_BLOCK_SIZE_BITS(sb);
bhb = bhe = buflist;

```

```

if (filp->f_reada) {

```

```

// 37 /* This specifies how many sectors to read ahead on the disk. */
// 39 int read_ahead[MAX_BLKDEV] = {0, };

```

```

    blocks += read_ahead[MAJOR(inode->i_dev)] >>
        (EXT2_BLOCK_SIZE_BITS(sb) - 9);
    if (block + blocks > size)
        blocks = size - block;
}

```

```

/*

```

```

    * We do this in a two stage process.  We first try and request
    * as many blocks as we can, then we wait for the first one to
    * complete, and then we try and wrap up as many as are actually
    * done.  This routine is rather generic, in that it can be used
    * in a filesystem by substituting the appropriate function in
    * for getblk

```

```

    *

```

```

    * This routine is optimized to make maximum use of the various
    * buffers and caches.

```

```

    */

```

```

do {

```

```

    bhrequest = 0;
    uptodate = 1;
    while (blocks) {
        --blocks;
        *bhb = ext2_getblk (inode, block++, 0, &err);
        if (*bhb && !(*bhb)->b_uptodate) {
            uptodate = 0;
            bhreq[bhrequest++] = *bhb;
        }
    }

```

```

    if (++bhb == &buflist[NBUF])
        bhb = buflist;

```

```

    /*

```

```

        * If the block we have on hand is uptodate, go ahead
        * and complete processing

```

```

    */
    if (uptodate)
        break;
    if (bhb == bhe)
        break;
}
/*
 * Now request them all
 */
if (bhrequest)
    ll_rw_block (READ, bhrequest, bhreq);
do {
    /*
     * Finish off all I/O that has actually completed
     */
    if (*bhe) {
        wait_on_buffer (*bhe);
        if (!(*bhe)->b_uptodate) { /* read error? */

            brelse(*bhe);
            if (++bhe == &buflist[NBUF])
                bhe = buflist;
            left = 0;
            break;
        }
    }
}

if (left < s_blocksize - offset)
    chars = left;
else
    chars = sb->s_blocksize - offset;
filp->f_pos += chars;
left -= chars;
read += chars;
if (*bhe) {
    memcpy_tofs (buf, offset + (*bhe)->b_data,
                chars);
    brelse (*bhe);
    buf += chars;
} else {
    while (chars-- > 0)
        put_fs_byte (0, buf++);
}

```

```

    }
    offset = 0;
    if (++bhe == &buflist[NBUF])
        bhe = buflist;
    } while (left > 0 && bhe != bhb && (!*bhe || !(*bhe)->b_lock));
} while (left > 0);
/*
 * Release the read-ahead blocks
 */
while (bhe != bhb) {
    brelse (*bhe);
    if (++bhe == &buflist[NBUF])
        bhe = buflist;
}
if (!read)
    return -EIO;
filp->f_reada = 1;
if (!IS_RDONLY(inode)) {
    inode->i_atime = CURRENT_TIME;
    inode->i_dirt = 1;
}
return read;
}

static int ext2_file_write (struct inode * inode, struct file * filp,
                           char * buf, int count)
{
    off_t pos;
    int written, c;
    struct buffer_head * bh;
    char * p;
    struct super_block * sb;
    int err;
    if (!inode) {
        printk("ext2_file_write: inode = NULL\n");
        return -EINVAL;
    }
    sb = inode->i_sb;
    if (sb->s_flags & MS_RDONLY)
        /*
         * This fs has been automatically remounted ro because of errors
         */
        return -ENOSPC;

```

```

if (!S_ISREG(inode->i_mode)) {
    ext2_warning (sb, "ext2_file_write", "mode = %07o\n",
        inode->i_mode);
    return -EINVAL;
}
/*
 * ok, append may not work when many processes are writing at the same time
 * but so what. That way leads to madness anyway.
 */
if (filp->f_flags & O_APPEND)
    pos = inode->i_size;
else
    pos = filp->f_pos;
written = 0;
while (written < s_blocksize, 1, &err);
    if (!bh) {
        if (!written)
            written = err;
        break;
    }
    c = s_blocksize - (pos % s_blocksize);
    if (c > count-written)
        c = count - written;
    if (c != s_blocksize && !bh->b_uptodate) {
        ll_rw_block (READ, 1, &bh);
        wait_on_buffer (bh);
        if (!bh->b_uptodate) {
            brelse (bh);
            if (!written)
                written = -EIO;
            break;
        }
    }
    p = (pos % s_blocksize) + bh->b_data;
    pos += c;
    if (pos > inode->i_size) {
        inode->i_size = pos;
        inode->i_dirt = 1;
    }
    written += c;
    memcpy_fromfs (p, buf, c); //写入到缓冲块中
    buf += c;

```



```

    bh->b_uptodate = 1;
    bh->b_dirt = 1;
    brelse (bh);
}
inode->i_ctime = inode->i_mtime = CURRENT_TIME;
filp->f_pos = pos;
inode->i_dirt = 1;
return written;
}

```

分析:

对于整个文件的索引操作担，都使用块计算的。

整块读取时，调用底层的读取函数。只要得到了更新块就跳出去处理，否则就记录要读取的块

整块的写入时很好办的，只要整体为脏就可以了。但是对于不能整块写入的，必须写入的块原先是更新的。

源码行号: 注释:

```

86  left = size - offset;  //剩余可读数
88  left = count;  //left 就是最终要读取的数据
92  //下面的 size 和 blocks 是对于整个文件的索引操作担，都使用块计算的
93  size = (size + sb->s_blocksize - 1) >> EXT2_BLOCK_SIZE_BITS(sb); //整个文件的
块数
94  blocks = (left + offset + sb->s_blocksize - 1) >> EXT2_BLOCK_SIZE_BITS(sb);
//要读取的块数
95  bhb = bhe = buflist;
97  //进行预读
98  if (filp->f_reada) {
99
100// 37 /* This specifies how many sectors to read ahead on the disk. */
101// 39 int read_ahead[MAX_BLKDEV] = {0, };
102  blocks += read_ahead[MAJOR(inode->i_dev)] >>
103  (EXT2_BLOCK_SIZE_BITS(sb) - 9);  //可能都是使用 512 标示的，这里进行校正
104  if (block + blocks > size)
105  blocks = size - block;  //校正实际能读取的块数
106}
126  bhreq[bhrequest++] = *bhb;  //同时 bhrequest 记录了需要重新读取的块数
128  if (++bhb == &buflist[NBUF])  //回卷到头部
134  if (uptodate)  //只要得到了更新块就跳出去处理，否则就记录要读取的块
136  if (bhb == bhe)  //满了，要读取 NBUF 块
142  if (bhrequest)  //调用底层的读取函数
150  if (!(*bhe)->b_uptodate) { /* read error? */
//这里当做错误处理的，left=0,跳出大循环了
154  bhe = buflist; //不是跳出去就完了，这里的 bhe 将来还是有用的

```

```

184  while (bhe != bhb) { //预读的块释放掉
191  filp->f_reada = 1; //标志已经预读了
226  if (filp->f_flags & O_APPEND) //是否追加, 更新文件指针
237  c = sb->s_blocksize - (pos % sb->s_blocksize); //c 记录本次可以读取的字符个数
240  if (c != sb->s_blocksize && !bh->b_uptodate) { //整块的写入时很好办的, 只要
整体为脏就可以了.但是对于不能整块写入的, 必须写入的块原先是更新的.但是想想呢, 好像不更新也
行啊 :-( ??
257  memcpy_fromfs (p, buf, c); //写入到缓冲块中

```