Linux 下的 C 编程实战（一）

开发平台搭建

1.引言

Linux 操作系统在服务器领域的应用和普及已经有较长的历史，这源于它的开源特点以及其超越 Windows 的安全性和稳定性。而近年来，Linux 操作系统在嵌入式系统领域的延伸也可谓是如日中天，许多版本的嵌入式 Linux 系统被开发出来，如 ucLinux、RTLinux、ARM-Linux 等等。在嵌入式操作系统方面，Linux 的地位是不容怀疑的，它开源、它包含 TCP/IP 协议栈、它易集成 GUI。

鉴于 Linux 操作系统在服务器和嵌入式系统领域愈来愈广泛的应用，社会上越来越需要基于 Linux 操作系统进行编程的开发人员。

浏览许多论坛，经常碰到这样的提问：“现在是不是很流行 unix/linux 下的 c 编程？所以想学习一下！但是不知道该从何学起，如何下手！有什么好的建议吗？各位高手！哪些书籍比较合适初学者？在深入浅出的过程中应该看哪些不同层次的书？比如好的网站、论坛请大家赐教！不慎感激！”

鉴于读者的需求，在本文中，笔者将对 Linux 平台下 C 编程的几个方面进行实例讲解，并力求回答读者们关心的问题，以与读者朋友们进行交流，共同提高。

在本文的连载过程中，有任何问题或建议，您可以给笔者发送 email：

21cnbao@21cn.com，您也可以进入笔者的博客参与讨论：

http://blog.donews.com/21cnbao。

笔者建议在 PC 内存足够大的情况下，不要直接安装 Linux 操作系统，最好把它安装在运行 VMWare 虚拟机软件的 Windows 平台上。

在 Linux 平台下，可用任意一个文本编辑工具编辑源代码，但笔者建议使用 emacs 软件，它具备语法高亮、版本控制等附带功能。

2.GCC 编译器

GCC 是 Linux 平台下最重要的开发工具，它是 GNU 的 C 和 C++编译器，其基本用法为：

gcc [options] [filenames]

options 为编译选项，GCC 总共提供的编译选项超过100个，但只有少数几个会被频繁使用，我们仅对几个常用选项进行介绍。

假设我们编译一输出“Hello World”的程序：

Cpp 代码

```
/* Filename:helloworld.c */
main()
{
    printf("Hello World\n");
}
```

最简单的编译方法是不指定任何编译选项：

gcc helloworld.c

它会为目标程序生成默认的文件名 a.out，我们可用-o 编译选项来为将产生的可执行文件指定一个文件名来代替 a.out。例如，将上述名为 helloworld.c 的 C 程序编译为名叫 helloworld 的可执行文件，需要输入如下命令：

gcc -o helloworld helloworld.c

-c 选项告诉 GCC 仅把源代码编译为目标代码而跳过汇编和连接的步骤；

-S 编译选项告诉 GCC 在为 C 代码产生了汇编语言文件后停止编译。GCC 产生的汇编语言文件的缺省扩展名是.s，上述程序运行如下命令：

gcc -S helloworld.c

将生成 helloworld.c 的汇编代码，使用的是 AT&T 汇编。用 emacs 打开汇编代码。

-E 选项指示编译器仅对输入文件进行预处理。当这个选项被使用时，预处理器的输出被送到标准输出（默认为屏幕）而不是储存在文件里。

-O 选项告诉 GCC 对源代码进行基本优化从而使得程序执行地更快；而-O2选项告诉 GCC 产生尽可能小和尽可能快的代码。使用-O2选项编译的速度比使用-O 时慢，但产生的代码执行速度会更快。

-g 选项告诉 GCC 产生能被 GNU 调试器使用的调试信息以便调试你的程序，可喜的是，在 GCC 里，我们能联用-g 和-O（产生优化代码）。

-pg 选项告诉 GCC 在你的程序里加入额外的代码，执行时，产生 gprof 用的剖析信息以显示你的程序的耗时情况。

3.GDB 调试器

GCC 用于编译程序，而 Linux 的另一个 GNU 工具 gdb 则用于调试程序。gdb 是一个用来调试 C 和 C++程序的强力调试器，我们能通过它进行一系列调试工作，包括设置断点、观查变量、单步等。

其最常用的命令如下：

file：装入想要调试的可执行文件。

kill：终止正在调试的程序。

list：列表显示源代码。

next：执行一行源代码但不进入函数内部。

step：执行一行源代码而且进入函数内部。

run：执行当前被调试的程序

quit：终止 gdb

watch：监视一个变量的值

break：在代码里设置断点，程序执行到这里时挂起

make：不退出 gdb 而重新产生可执行文件

shell：不离开 gdb 而执行 shell

下面我们来演示怎样用 GDB 来调试一个求0+1+2+3+…+99的程序：

Cpp 代码

```
/* Filename:sum.c */
main()
{
  int i, sum;
  sum = 0;
  for (i = 0; i < 100; i++)
  {
    sum +  = i;
  }
  printf("the sum of 1+2+...+ is %d", sum);
}
```

执行如下命令编译 sum.c（加-g 选项产生 debug 信息）：

gcc -g -o sum sum.c

在命令行上键入 gdb sum 并按回车键就可以开始调试 sum 了，再运行 run 命令执行 sum。

list 命令：

list 命令用于列出源代码，对上述程序两次运行 list，将出现如下画面（源代码被标行号）。

根据列出的源程序，如果我们将断点设置在第5行，只需在 gdb 命令行提示符下键入如下命令设置断点：(gdb) break 5。

这个时候我们再 run，程序会停止在第5行。

设置断点的另一种语法是 break <function>，它在进入指定函数 (function) 时停住。

相反的，clear 用于清除所有的已定义的断点，clear <function>清除设置在函数上的断点， clear <linenum>则清除设置在指定行上的断点。

watch 命令：

watch 命令用于观查变量或表达式的值，我们观查 sum 变量只需要运行 watch sum。

watch <expr>为表达式（变量）expr 设置一个观察点，一量表达式值有变化时，程序会停止执行。

要观查当前设置的 watch，可以使用 info watchpoints 命令。

next、step 命令：

next、step 用于单步执行，在执行的过程中，被 watch 变量的变化情况将实时呈现(分别显示 Old value 和 New value)。

next、step 命令的区别在于 step 遇到函数调用，会跳转到到该函数定义的开始行去执行，而 next 则不进入到函数内部，它把函数调用语句当作一条普通语句执行。

4.Make

make 是所有想在 Linux 系统上编程的用户必须掌握的工具，对于任何稍具规模的程序，我们都会使用到 make，几乎可以说不使用 make 的程序不具备任何实用价值。

在此，我们有必要解释编译和连接的区别。编译器使用源码文件来产生某种形式的目标文件(object files)，在编译过程中，外部的符号参考并没有被解释或替换（即外部全局变量和函数并没有被找到）。因此，在编译阶段所报的错误一般都是语法错误。而连接器则用于连接目标文件和程序包，生成一个可执行程序。在连接阶段，一个目标文件中对别的文件中的符号的参考被解释，如果有符号不能找到，会报告连接错误。

编译和连接的一般步骤是：第一阶段把源文件一个一个的编译成目标文件，第二阶段把所有的目标文件加上需要的程序包连接成一个可执行文件。这样的过程很痛苦，我们需要使用大量的 gcc 命令。

而 make 则使我们从大量源文件的编译和连接工作中解放出来，综合为一步完成。GNU Make 的主要工作是读进一个文本文件，称为 makefile。这个文件记录了哪些文件（目的文件，目的文件不一定是最后的可执行程序，它可以是任何一种文件）由哪些文件（依靠文件）产生，用什么命令来产生。Make 依

靠此 makefile 中的信息检查磁盘上的文件，如果目的文件的创建或修改时间比它的一个依靠文件旧的话，make 就执行相应的命令，以便更新目的文件。

假设我们写下如下的三个文件，add.h 用于声明 add 函数，add.c 提供两个整数相加的函数体，而 main.c 中调用 add 函数：

Cpp 代码

```cpp
/* filename:add.h */
extern int add(int i, int j);
/* filename:add.c */
int add(int i, int j)
{
  return i + j;
}
/* filename:main.c */
#include "add.h"
main()
{
  int a, b;
  a = 2;
  b = 3;
  printf("the sum of a+b is %d", add(a + b));
}
```

怎样为上述三个文件产生 makefile 呢？如下：

```
test : main.o add.o

gcc main.o add.o -o test


main.o : main.c add.h

gcc -c main.c -o main.o


add.o : add.c add.h

gcc -c add.c -o add.o
```

上述 makefile 利用 add.c 和 add.h 文件执行 gcc -c add.c -o add.o 命令产

生 add.o 目标代码，利用 main.c 和 add.h 文件执行 gcc -c main.c -o main.o 命令产生 main.o 目标代码，最后利用 main.o 和 add.o 文件（两个模块的目标代码）执行 gcc main.o add.o -o test 命令产生可执行文件 test。

我们可在 makefile 中加入变量，另外。环境变量在 make 过程中也被解释成 make 的变量。这些变量是大小写敏感的，一般使用大写字母。Make 变量可以做很多事情，例如：

i) 存储一个文件名列表；

ii) 存储可执行文件名；

iii) 存储编译器选项。

要定义一个变量，只需要在一行的开始写下这个变量的名字，后面跟一个=号，再跟变量的值。引用变量的方法是写一个$符号，后面跟（变量名）。我们把前面的 makefile 利用变量重写一遍（并假设使用-Wall -O -g 编译选项）：

```
OBJS = main.o add.o

CC = gcc

CFLAGS = -Wall -O -g

test : $(OBJS)

$(CC) $(OBJS) -o test

main.o : main.c add.h

$(CC) $(CFLAGS) -c main.c -o main.o

add.o : add.c add.h
```

$(CC) $(CFLAGS) -c add.c -o add.o

makefile 中还可定义清除（clean）目标，可用来清除编译过程中产生的中间文件，例如在上述 makefile 文件中添加下列代码：

clean:

rm -f *.o

运行 make clean 时，将执行 rm -f *.o 命令，删除所有编译过程中产生的中间文件。

不管怎么说，自己动手编写 makefile 仍然是很复杂和烦琐的，而且很容易出错。因此，GNU 也为我们提供了 Automake 和 Autoconf 来辅助快速自动产生 makefile，读者可以参阅相关资料。

5.小结

本章主要阐述了 Linux 程序的编写、编译、调试方法及 make，实际上就是引导读者学习怎样在 Linux 下编程，为后续章节做好准备。

Linux 下的 C 编程实战（二）

——文件系统编程

宋宝华21cnbao@21cn.com

1.Linux 文件系统

Linux 支持多种文件系统，如 ext、ext2、minix、iso9660、msdos、fat、vfat、nfs 等。在这些具体文件系统的上层，Linux 提供了虚拟文件系统（VFS）来统一它们的行为，虚拟文件系统为不同的文件系统与内核的通信提供了一致的接口。下图给出了 Linux 中文件系统的关系：

在 Linux 平台下对文件编程可以使用两类函数：（1）Linux 操作系统文件 API；（2）C 语言 I/O 库函数。前者依赖于 Linux 系统调用，后者实际上与操作系统是独立的，因为在任何操作系统下，使用 C 语言 I/O 库函数操作文件的方法都是相同的。本章将对这两种方法进行实例讲解。

2.Linux 文件 API

Linux 的文件操作 API 涉及到创建、打开、读写和关闭文件。

创建

int creat(const char *filename, mode_t mode);

参数 mode 指定新建文件的存取权限，它同 umask 一起决定文件的最终权限（mode&umask），其中 umask 代表了文件在创建时需要去掉的一些存取权限。umask 可通过系统调用 umask()来改变：

int umask(int newmask);

该调用将 umask 设置为 newmask，然后返回旧的 umask，它只影响读、写和执行权限。

打开

int open(const char *pathname, int flags);

int open(const char *pathname, int flags, mode_t mode);

open 函数有两个形式，其中 pathname 是我们要打开的文件名(包含路径名称，缺省是认为在当前路径下面)，flags 可以去下面的一个值或者是几个值的组合：

标志	含义
O_RDONLY	以只读的方式打开文件
O_WRONLY	以只写的方式打开文件
O_RDWR	以读写的方式打开文件
O_APPEND	以追加的方式打开文件
O_CREAT	创建一个文件
O_EXEC	如果使用了 O_CREAT 而且文件已经存在，就会发生一个错误
O_NOBLOCK	以非阻塞的方式打开一个文件
O_TRUNC	如果文件已经存在，则删除文件的内容

O_RDONLY、O_WRONLY、O_RDWR 三个标志只能使用任意的一个。

如果使用了 O_CREATE 标志，则使用的函数是 int open(const char *pathname,int flags,mode_t mode);这个时候我们还要指定 mode 标志，用来表示文件的访问权限。mode 可以是以下情况的组合：

标志	含义
S_IRUSR	用户可以读
S_IWUSR	用户可以写
S_IXUSR	用户可以执行
S_IRWXU	用户可以读、写、执行
S_IRGRP	组可以读
S_IWGRP	组可以写
S_IXGRP	组可以执行
S_IRWXG	组可以读写执行
S_IROTH	其他人可以读
S_IWOTH	其他人可以写
S_IXOTH	其他人可以执行
S_IRWXO	其他人可以读、写、执行
S_ISUID	设置用户执行 ID
S_ISGID	设置组的执行 ID

除了可以通过上述宏进行“或”逻辑产生标志以外，我们也可以自己用数字来表示，Linux 总共用5个数字来表示文件的各种权限：第一位表示设置用户 ID；第二位表示设置组 ID；第三位表示用户自己的权限位；第四位表示组的权限；最后一位表示其他人的权限。每个数字可以取1(执行权限)、2(写权限)、4(读权限)、

0(无)或者是这些值的和。例如，要创建一个用户可读、可写、可执行，但是组没有权限，其他人可以读、可以执行的文件，并设置用户 ID 位。那么，我们应该使用的模式是1(设置用户 ID)、0(不设置组 ID)、7(1+2+4，读、写、执行)、0(没有权限)、5(1+4，读、执行)即10705：

open("test", O_CREAT, 10705);

上述语句等价于：

open("test", O_CREAT, S_IRWXU | S_IROTH | S_IXOTH | S_ISUID);

如果文件打开成功，open 函数会返回一个文件描述符，以后对该文件的所有操作就可以通过对这个文件描述符进行操作来实现。

读写

在文件打开以后，我们才可对文件进行读写了，Linux 中提供文件读写的系统调用是 read、write 函数：

int read(int fd, const void *buf, size_t length);

int write(int fd, const void *buf, size_t length);

其中参数 buf 为指向缓冲区的指针，length 为缓冲区的大小（以字节为单位）。函数 read()实现从文件描述符 fd 所指定的文件中读取 length 个字节到 buf 所指向的缓冲区中，返回值为实际读取的字节数。函数 write 实现将把 length 个字节从 buf 指向的缓冲区中写到文件描述符 fd 所指向的文件中，返回值为实际写入的字节数。

以 O_CREAT 为标志的 open 实际上实现了文件创建的功能，因此，下面的函数等同 creat()函数：

int open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);

定位

对于随机文件，我们可以随机的指定位置读写，使用如下函数进行定位：

int lseek(int fd, offset_t offset, int whence);

lseek()将文件读写指针相对 whence 移动 offset 个字节。操作成功时，返回文件指针相对于文件头的位置。参数 whence 可使用下述值：

SEEK_SET：相对文件开头

SEEK_CUR：相对文件读写指针的当前位置

SEEK_END：相对文件末尾

offset 可取负值，例如下述调用可将文件指针相对当前位置向前移动5个字节：

lseek(fd, -5, SEEK_CUR);

由于 lseek 函数的返回值为文件指针相对于文件头的位置，因此下列调用的返回值就是文件的长度：

lseek(fd, 0, SEEK_END);

关闭

当我们操作完成以后，我们要关闭文件了，只要调用 close 就可以了，其中 fd 是我们要关闭的文件描述符：

int close(int fd);

例程：编写一个程序，在当前目录下创建用户可读写文件“hello.txt”，在其中写入“Hello, software weekly”，关闭该文件。再次打开该文件，读取其中的内容并输出在屏幕上。

Cpp 代码

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5  #define LENGTH 100
```

```
6  main()
7  {
8    int fd, len;
9    char str[LENGTH];
10
11   fd = open("hello.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); /* 创建并
打开文件 */
12   if (fd)
13   {
14   write(fd, "Hello, Software Weekly", strlen("Hello, software weekly")); /*
写入 Hello, software weekly 字符串 */
15   close(fd);
16   }
17
18   fd = open("hello.txt", O_RDWR);
19   len = read(fd, str, LENGTH); /* 读取文件内容 */
20   str[len] = '\0';
21   printf("%s\n", str);
22   close(fd);
23 }
```

编译并运行，执行结果如下图：

<!--[if gte vml 1]><v:shape id="_x0000_i1026" type="#_x0000_t75" alt="" style="width:266.25pt;height:34.5pt" mce_style="width:266.25pt;height:34.5pt"

/><![endif]-->

3.C 语言库函数

C 库函数的文件操作实际上是独立于具体的操作系统平台的，不管是在 DOS、Windows、Linux 还是在 VxWorks 中都是这些函数：

创建和打开

FILE *fopen(const char *path, const char *mode);

fopen()实现打开指定文件 filename，其中的 mode 为打开模式，C 语言中支持的打开模式如下表：

标志	含义
r, rb	以只读方式打开
w, wb	以只写方式打开。如果文件不存在，则创建该文件，否则文件被截断
a, ab	以追加方式打开。如果文件不存在，则创建该文件
r+, r+b, rb+	以读写方式打开
w+, w+b, wh+	以读写方式打开。如果文件不存在时，创建新文件，否则文件被截断
a+, a+b, ab+	以读和追加方式打开。如果文件不存

在，创建新文件

其中 b 用于区分二进制文件和文本文件，这一点在 DOS、Windows 系统中是有区分的，但 Linux 不区分二进制文件和文本文件。

读写

C 库函数支持以字符、字符串等为单位，支持按照某中格式进行文件的读写，这一组函数为：

Cpp 代码

```cpp
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf (FILE *stream, const char *format, ...);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t n, FILE *stream);
```

fread()实现从流 stream 中读取加 n 个字段，每个字段为 size 字节，并将读取的字段放入 ptr 所指的字符数组中，返回实际已读取的字段数。在读取的字段数小于 num 时，可能是在函数调用时出现错误，也可能是读到文件的结尾。所以要通过调用 feof()和 ferror()来判断。

write()实现从缓冲区 ptr 所指的数组中把 n 个字段写到流 stream 中，每个字段长为 size 个字节，返回实际写入的字段数。

另外，C 库函数还提供了读写过程中的定位能力，这些函数包括

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, const fpos_t *pos);

int fseek(FILE *stream, long offset, int whence);

等。

关闭

利用 C 库函数关闭文件依然是很简单的操作：

int fclose (FILE *stream);

例程：将第2节中的例程用 C 库函数来实现。

Cpp 代码

```cpp
#include <stdio.h>
#define LENGTH 100
main()
{
    FILE *fd;
    char str[LENGTH];

```

```
39      fd = fopen("hello.txt", "w+"); /* 创建并打开文件 */
40      if (fd)
41      {
42          fputs("Hello, Software Weekly", fd); /* 写入 Hello, software weekly
字符串 */
43          fclose(fd);
44      }
45
46      fd = fopen("hello.txt", "r");
47      fgets(str, LENGTH, fd); /* 读取文件内容 */
48      printf("%s\n", str);
49      fclose(fd);
50  }
```

4.小结

Linux 提供的虚拟文件系统为多种文件系统提供了统一的接口，Linux 的文件编程有两种途径：基于 Linux 系统调用；基于 C 库函数。这两种编程所涉及到文件操作有新建、打开、读写和关闭，对随机文件还可以定位。本章对这两种编程方法都给出了具体的实例。

Linux 下的 C 编程实战之（三）

1.Linux 进程

Linux 进程在内存中包含三部分数据：代码 段、堆栈段和数据段。代码段存放了程序的代码。代码段可以为机器中运行同一程序的数个进程共享。堆栈段存放的是子程序（函数）的返回地址、子程序的参数及程序的局部变量。而数据段则存放程序的全局变量、常数以及动态数据分配的数据空间（比如用 malloc 函数申请的内存）。与代码段不同，如果系统中同时运行多个相同的程序，它们不能使用同一堆栈段和数据段。

Linux 进程主要有如下几种状态：用户状态（进程在用户状态下运行的状态）、 内核状态（进程在内核状态下运行的状态）、内存中就绪（进程没有执行，但处于就绪状态，只要内核调度它，就可以执行）、内存中睡眠（进程正在睡眠并且处于内存中，没有被交换到 SWAP 设备）、就绪且换出（进程处于就绪状态，但是必须把它换入内存，内核才能再次调度它进行运行）、睡眠且换出（进程正在睡眠，且被换出内存）、被抢先（进程从内核状态返回用户状态时，内核抢先于它，做了上下文切换，调度了另一个进程，原先这个进程就处于被抢先状态）、创建状态 （进程刚被创建，该进程存在，但既不是就绪状态，也不是睡眠状

态，这个状态是除了进程0以外的所有进程的最初状态）、僵死状态（进程调用 exit 结束，进 程不再存在，但在进程表项中仍有记录，该记录可由父进程收集）。

下面我们来以一个进程从创建到消亡的过程讲解 Linux 进程状态转换的“生死因果”。

（1）进程被父进程通过系统调用 fork 创建而处于创建态；

（2）fork 调用为子进程配置好内核数据结构和子进程私有数据结构后，子进程进入就绪态（或者在内存中就绪，或者因为内存不够而在 SWAP 设备中就绪）；

（3）若进程在内存中就绪，进程可以被内核调度程序调度到 CPU 运行；

（4）内核调度该进程进入内核状态，再由内核状态返回用户状态执行。该进程在用户状态运行一定时间后，又会被调度程序所调度而进入内核状态，由此转入就绪态。有时进程在用户状态运行时，也会因为需要内核服务，使用系统调用而进入内核状态，服务完毕，会由内核状态转回用户状态。要注意的是，进程在从内核状 态向用户状态返回时可能被抢占，这是由于有优先级更高的进程急需使用 CPU，不能等到下一次调度时机，从而造成抢占；

（5）进程执行 exit 调用，进入僵死状态，最终结束。

2.进程控制

进程控制中主要涉及到进程的创建、睡眠和退出等，在 Linux 中主要提供了 fork、exec、clone 的进程创建方法，sleep 的进程睡眠和 exit 的进程退出调用，另外 Linux 还提供了父进程等待子进程结束的系统调用 wait。

fork

对于没有接触过 Unix/Linux 操作系统的人来说，fork 是最难理解的概念之一，它执行一次却返回两个值，完全“不可思议”。先看下面的程序：

Cpp 代码

```cpp
int main()
{
  int i;
  if (fork() == 0)
  {
    for (i = 1; i < 3; i++)
      printf("This is child process\n");
  }
  else
  {
    for (i = 1; i < 3; i++)
      printf("This is parent process\n");
```

```
    }
}
```

执行结果为：

```
This is child process
This is child process
This is parent process
This is parent process
```

fork 在英文中是“分叉”的意思，这个名字取得很形象。一个进程在运行中，如果使用了 fork，就产生了另一个进程，于是进程就“分叉”了。当前进程 为父进程，通过 fork()会产生一个子进程。对于父进程，fork 函数返回子程序的进程号而对于子程序，fork 函数则返回零，这就是一个函数返回两次的本质。可以说，fork 函数是 Unix 系统最杰出的成就之一，它是七十年代 Unix 早期的开发者经过理论和实践上的长期艰苦探索后取得的成果。

如果我们把上述程序中的循环放的大一点：

Cpp 代码

```cpp
int main()
{
  int i;
  if (fork() == 0)
  {
    for (i = 1; i < 10000; i++)
      printf("This is child process\n");
  }
  else
  {
    for (i = 1; i < 10000; i++)
      printf("This is parent process\n");
  }
}
```

则可以明显地看到父进程和子进程的并发执行，交替地输出“This is child process”和“This is parent process”。

此时此刻，我们还没有完全理解 fork()函数，再来看下面的一段程序，看看究竟会产生多少个进程，程序的输出是什么？

Cpp 代码

```cpp
int main()
{
  int i;
  for (i = 0; i < 2; i++)
  {
    if (fork() == 0)
    {
      printf("This is child process\n");
    }
    else
    {
      printf("This is parent process\n");
    }
  }
}
```

exec

在 Linux 中可使用 exec 函数族，包含多个函数（execl、execlp、execle、execv、execve 和 execvp），被用于启动一个指定路径和文件名的进程。

exec 函数族的特点体现在：某进程一旦调用了 exec 类函数，正在执行的程序就被干掉了，系统把代码段替换成新的程序（由 exec 类函数执行）的代 码，并且原有的数据段和堆栈段也被废弃，新的数据段与堆栈段被分配，但是进程号却被保留。也就是说，exec 执行的结果为：系统认为正在执行的还是原先的进程，但是进程对应的程序被替换了。

fork 函数可以创建一个子进程而当前进程不死，如果我们在 fork 的子进程中调用 exec 函数 族就可以实现既让父进程的代码执行又启动一个新的指定进程，这实在是很妙的。fork 和 exec 的搭配巧妙地解决了程序启动另一程序的执行但自己仍继续运 行的问题，请看下面的例子：

Cpp 代码

```cpp
char command[MAX_CMD_LEN];
void main()
{
  int rtn; /* 子进程的返回数值 */
  while (1)
  {
    /* 从终端读取要执行的命令 */
    printf(">");
    fgets(command, MAX_CMD_LEN, stdin);
```

```
53      command[strlen(command) - 1] = 0;
54      if (fork() == 0)
55      {
56        /* 子进程执行此命令 */
57        execlp(command, command);
58        /* 如果 exec 函数返回，表明没有正常执行命令，打印错误信息*/
59        perror(command);
60        exit(errorno);
61      }
62      else
63      {
64        /* 父进程，等待子进程结束，并打印子进程的返回值 */
65        wait(&rtn);
66        printf(" child process return %d\n", rtn);
67      }
68    }
69  }
```

这个函数基本上实现了一个 shell 的功能，它读取用户输入的进程名和参数，并启动对应的进程。

clone

clone 是 Linux2.0以后才具备的新功能，它较 fork 更强（可认为 fork 是 clone 要实现的一部分），可以使得创建的子进程共享父进程的资源，并且要使用此函数必须在编译内核时设置 clone_actually_works_ok 选项。

clone 函数的原型为：

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

此函数返回创建进程的 PID，函数中的 flags 标志用于设置创建子进程时的相关选项，具体含义如下表：

标志	含义
CLONE_PARENT	创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”
CLONE_FS	子进程与父进程共享相同的文件系统，包括 root、当前目录、umask
CLONE_FILES	子进程与父进程共享相同的文件描述符（file descriptor）表
CLONE_NEWNS	在新的 namespace 启动子进程，namespace 描述了进程的文件 hierarchy
CLONE_SIGHAND	子进程与父进程共享相同的信号处理（signal handler）表

CLONE_PTRACE	若父进程被 trace，子进程也被 trace
CLONE_VFORK	父进程被挂起，直至子进程释放虚拟内存资源
CLONE_VM	子进程与父进程运行于相同的内存空间
CLONE_PID	子进程在创建时 PID 与父进程一致
CLONE_THREAD	Linux 2.4中增加以支持 POSIX 线程标准，子进程与父进程共享相同的线程群

来看下面的例子：

Cpp 代码

```cpp
int variable, fd;
int do_something() {
  variable = 42;
  close(fd);
  _exit(0);
}

int main(int argc, char *argv[]) {
  void **child_stack;
  char tempch;

  variable = 9;
  fd = open("test.file", O_RDONLY);
  child_stack = (void **) malloc(16384);
  printf("The variable was %d\n", variable);

  clone(do_something, child_stack, CLONE_VM|CLONE_FILES, NULL);
  sleep(1); /* 延时以便子进程完成关闭文件操作、修改变量  */

  printf("The variable is now %d\n", variable);
  if (read(fd, &tempch, 1) < 1) {
    perror("File Read Error");
    exit(1);
  }
  printf("We could read from the file\n");
  return 0;
}
```

运行输出：

The variable is now 42
File Read Error

程序的输出结果告诉我们，子进程将文件关闭并将变量修改（调用 clone 时用到的 CLONE_VM、CLONE_FILES 标志将使得变量和文件描述符表被共享），父进程随即就感觉到了，这就是 clone 的特点。

sleep

函数调用 sleep 可以用来使进程挂起指定的秒数，该函数的原型为：

unsigned int sleep(unsigned int seconds);

该函数调用使得进程挂起一个指定的时间，如果指定挂起的时间到了，该调用返回0；如果该函数调用被信号所打断，则返回剩余挂起的时间数（指定的时间减去已经挂起的时间）。

exit

系统调用 exit 的功能是终止本进程，其函数原型为：

void _exit(int status);

_exit 会立即终止发出调用的进程，所有属于该进程的文件描述符都关闭。参数 status 作为退出的状态值返回父进程，在父进程中通过系统调用 wait 可获得此值。

wait

wait 系统调用包括：

Cpp 代码

```
97  pid_t wait(int *status);
98  pid_t waitpid(pid_t pid, int *status, int options);
```

wait 的作用为发出调用的进程只要有子进程，就睡眠到它们中的一个终止为止；waitpid 等待由参数 pid 指定的子进程退出。

3.进程间通信

Linux 的进程间通信（IPC，InterProcess Communication）通信方法有管道、消息队列、共享内存、信号量、套接口等。

管道分为有名管道和无名管道，无名管道只能用于亲属进程之间的通信，而有名管道则可用于无亲属

关系的进程之间。

Cpp 代码

```cpp
#define INPUT 0
#define OUTPUT 1
void main()
{
  int file_descriptors[2];
  /*定义子进程号 */
  pid_t pid;
  char buf[BUFFER_LEN];
  int returned_count;
  /*创建无名管道*/
  pipe(file_descriptors);
  /*创建子进程*/
  if ((pid = fork()) == - 1)
  {
    printf("Error in fork\n");
    exit(1);
  }
  /*执行子进程*/
  if (pid == 0)
  {
    printf("in the spawned (child) process...\n");
    /*子进程向父进程写数据，关闭管道的读端*/
    close(file_descriptors[INPUT]);
    write(file_descriptors[OUTPUT], "test data", strlen("test data"));
    exit(0);
  }
  else
  {
    /*执行父进程*/
    printf("in the spawning (parent) process...\n");
    /*父进程从管道读取子进程写的数据，关闭管道的写端*/
    close(file_descriptors[OUTPUT]);
    returned_count = read(file_descriptors[INPUT], buf, sizeof(buf));
    printf("%d bytes of data received from spawned process: %s\n",
    returned_count, buf);
  }
}
```

上述程序中，无名管道以

int pipe(int filedis[2]);

方式定义，参数 filedis 返回两个文件描述符 filedes[0]为读而打开，filedes[1]为写而打开，filedes[1]的输出是 filedes[0]的输入；

在 Linux 系统下，有名管道可由两种方式创建（假设创建一个名为“fifoexample”的有名管道）：

（1）mkfifo("fifoexample","rw");

（2）mknod fifoexample p

mkfifo 是一个函数，mknod 是一个系统调用，即我们可以在 shell 下输出上述命令。

有名管道创建后，我们可以像读写文件一样读写之：

Cpp 代码

```
/* 进程一：读有名管道*/
void main()
{
  FILE *in_file;
  int count = 1;
  char buf[BUFFER_LEN];
  in_file = fopen("pipeexample", "r");
  if (in_file == NULL)
  {
    printf("Error in fdopen.\n");
    exit(1);
  }
  while ((count = fread(buf, 1, BUFFER_LEN, in_file)) > 0)
    printf("received from pipe: %s\n", buf);
    fclose(in_file);
}

/* 进程二：写有名管道*/
void main()
{
  FILE *out_file;
  int count = 1;
  char buf[BUFFER_LEN];
  out_file = fopen("pipeexample", "w");
  if (out_file == NULL)
  {
```

```
162     printf("Error opening pipe.");
163     exit(1);
164   }
165   sprintf(buf, "this is test data for the named pipe example\n");
166   fwrite(buf, 1, BUFFER_LEN, out_file);
167   fclose(out_file);
168 }
```

消息队列用于运行于同一台机器上的进程间通信，与管道相似；

共享内存通常由一个进程创建，其余进程对这块内存区进行读写。得到共享内存有两种方式：映射 /dev/mem 设备和内存映像文件。前一种方式不给系统带来额外的开销，但在现实中并不常用，因为它控制存取的是实际的物理内存；常用的方式是通过 shmXXX 函数族来实现共享内存：

Cpp 代码

```
169 int shmget(key_t key, int size, int flag);  /* 获得一个共享存储标识符 */
```

该函数使得系统分配 size 大小的内存用作共享内存；

Cpp 代码

```
170 void *shmat(int shmid, void *addr, int flag); /* 将共享内存连接到自身地址空间中*/
```

shmid 为 shmget 函数返回的共享存储标识符，addr 和 flag 参数决定了以什么方式来确定连接的地址，函数的返回值即是该进程数据段所连接的实际地址。此后，进程可以对此地址进行读写操作访问共享内存。

本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。一般说来，为了获得共享资源，进程需要执行下列操作：

（1）测试控制该资源的信号量；

（2）若此信号量的值为正，则允许进行使用该资源，进程将进号量减1；

（3）若此信号量为0，则该资源目前不可用，进程进入睡眠状态，直至信号量值大于0，进程被唤醒，转入步骤（1）；

（4）当进程不再使用一个信号量控制的资源时，信号量值加1，如果此时有进程正在睡眠等待此信号量，则唤醒此进程。

下面是一个使用信号量的例子，该程序创建一个特定的 IPC 结构的关键字和一个信号量，建立此信号量的索引，修改索引指向的信号量的值，最后清除信号量：

Cpp 代码

```cpp
#include <stdio.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/ipc.h>
void main()
{
    key_t unique_key; /* 定义一个 IPC 关键字*/
    int id;
    struct sembuf lock_it;
    union semun options;
    int i;

    unique_key = ftok(".", 'a'); /* 生成关键字，字符'a'是一个随机种子*/
    /* 创建一个新的信号量集合*/
    id = semget(unique_key, 1, IPC_CREAT | IPC_EXCL | 0666);
    printf("semaphore id=%d\n", id);
    options.val = 1; /*设置变量值*/
    semctl(id, 0, SETVAL, options); /*设置索引0的信号量*/

    /*打印出信号量的值*/
    i = semctl(id, 0, GETVAL, 0);
    printf("value of semaphore at index 0 is %d\n", i);

    /*下面重新设置信号量*/
    lock_it.sem_num = 0; /*设置哪个信号量*/
    lock_it.sem_op = - 1; /*定义操作*/
    lock_it.sem_flg = IPC_NOWAIT; /*操作方式*/
    if (semop(id, &lock_it, 1) == - 1)
    {
        printf("can not lock semaphore.\n");
        exit(1);
    }
```

```
203
204     i = semctl(id, 0, GETVAL, 0);
205     printf("value of semaphore at index 0 is %d\n", i);
206
207     /*清除信号量*/
208     semctl(id, 0, IPC_RMID, 0);
209 }
```

套接字通信并不为 Linux 所专有，在所有提供了 TCP/IP 协议栈的操作系统中几乎都提供了 socket，而所有这样操作系统，对套接字的编程方法几乎是完全一样的。

4.小节

本章讲述了 Linux 进程的概念，并以多个实例讲解了进程控制及进程间通信方法，理解这一章的内容可以说是理解 Linux 这个操作系统的关键。

Linux 下的 C 编程实战之(四)

Linux 内核只提供了轻量进程的支持，未实现线程模型，但 Linux 尽最大努力优化了进程的调度开销，这在一定程度上弥补无线程的缺陷。Linux 用一个核心进程(轻量进程)对应一个线程，将线程调度等同于进程调度，交给核心完成。

1.Linux“线程”

Linux 内核只提供了轻量进程的支持，未实现线程模型，但 Linux 尽最大努力

优化了进程的调度开销，这在一定程度上弥补无线程的缺陷。Linux 用一个核心进程(轻量进程)对应一个线程，将线程调度等同于进程调度，交给核心完成。

笔者曾经在《基于嵌入式操作系统 VxWorks 的多任务并发程序设计》(《软件报》2006年第5~12期)中详细叙述了进程和线程的区别，并曾经说 明 Linux 是一种“多进程单线程”的操作系统。Linux 本身只有进程的概念，而其所谓的“线程”本质上在内核里仍然是进程。大家知道，进程是资源分配 的单位，同一进程中的多个线程共享该进程的资源(如作为共享内存的全局变量)。Linux 中所谓的“线程”只是在被创建的时候“克隆”(clone)了父 进程的资源，因此，clone 出来的进程表现为“线程”，这一点一定要弄清楚。因此，Linux“线程”这个概念只有在打冒号的情况下才是最准确的，可惜 的是几乎没有书籍留心去强调这一点。

目前 Linux 中最流行的线程机制为 LinuxThreads，所采用的就是线程－进程“一对一”模型，调度交给核心，而在用户级实现一个包括信号处 理在内的线程管理机制。LinuxThreads 由 Xavier Leroy (Xavier.Leroy@inria.fr)负责开发完成，并已绑定在 GLIBC 中发行，它实现了一种 BiCapitalized 面向 Linux 的 Posix 1003.1c “pthread”标准接口。Linuxthread 可以支持 Intel、Alpha、MIPS 等平台上的多处理器系统。

按照 POSIX 1003.1c 标准编写的程序与 Linuxthread 库相链接即可支持 Linux 平台上的多线程，在程序中需包含头文件 pthread. h，在编译链接时使

用命令：

gcc -D -REENTRANT -lpthread xxx. c

其中-REENTRANT 宏使得相关库函数(如 stdio.h、errno.h 中函数) 是可重入的、线程安全的(thread-safe)，-lpthread 则意味着链接库目录下的 libpthread.a 或 libpthread.so 文 件。使用 Linuxthread 库需要2.0以上版本的 Linux 内核及相应版本的 C 库(libc 5.2.18、libc 5.4.12、libc 6)。

2.“线程”控制

线程创建

进程被创建时，系统会为其创建一个主线程，而要在进程中创建新的线程，则可以调用 pthread_create：

Cpp 代码

```
1  pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
2  (start_routine)(void*), void *arg);
```

start_routine 为新线程的入口函数，arg 为传递给 start_routine 的参数。

每个线程都有自己的线程 ID，以便在进程内区分。线程 ID 在 pthread_create 调用时回返给创建线程的调用者；一个线程也可以在创建后使用 pthread_self()调用获取自己的线程 ID：

pthread_self (void) ;

线程退出

线程的退出方式有三：

(1)执行完成后隐式退出；

(2)由线程本身显示调用 pthread_exit 函数退出；
pthread_exit (void * retval) ;

(3)被其他线程用 pthread_cance 函数终止：
pthread_cance (pthread_t thread) ;

在某线程中调用此函数，可以终止由参数 thread 指定的线程。

如果一个线程要等待另一个线程的终止，可以使用 pthread_join 函数，该函数的作用是调用 pthread_join 的线程将被挂起直到线程 ID 为参数 thread 的线程终止：
pthread_join (pthread_t thread, void** threadreturn);
3.线程通信

线程互斥

互斥意味着“排它”，即两个线程不能同时进入被互斥保护的代码。Linux 下可

以通过 pthread_mutex_t 定义互斥体机制完成多线程的互斥操作，该机制的作用是对某个需要互斥的部分，在进入时先得到互斥体，如果没有得到互斥体，表明互斥部分被其它线程拥有，此 时欲获取互斥体的线程阻塞，直到拥有该互斥体的线程完成互斥部分的操作为止。

下面的代码实现了对共享全局变量 x 用互斥体 mutex 进行保护的目的：

Cpp 代码

```
int x; // 进程中的全局变量
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); //按缺省的属性初始化互斥体变量 mutex
pthread_mutex_lock(&mutex); // 给互斥体变量加锁
... //对变量 x 的操作
phtread_mutex_unlock(&mutex); // 给互斥体变量解除锁
```

线程同步

同步就是线程等待某个事件的发生。只有当等待的事件发生线程才继续执行，否则线程挂起并放弃处理器。当多个线程协作时，相互作用的任务必须在一定的条件下同步。

Linux 下的 C 语言编程有多种线程同步机制，最典型的是条件变量(condition variable)。pthread_cond_init 用来创建一个条件变量，其函数原型为：
pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);

pthread_cond_wait 和 pthread_cond_timedwait 用来等 待条件变量被设置，值得注意的是这两个等待调用需要一个已经上锁的互斥体 mutex，这是为了

防止在真正进入等待状态之前别的线程有可能设置该条件变量而 产生竞争。

pthread_cond_wait 的函数原型为：

pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);

pthread_cond_broadcast 用于设置条件变量，即使得事件发生，这样等待该事件的线程将不再阻塞：

pthread_cond_broadcast (pthread_cond_t *cond) ;

pthread_cond_signal 则用于解除某一个等待线程的阻塞状态：

pthread_cond_signal (pthread_cond_t *cond) ;

pthread_cond_destroy 则用于释放一个条件变量的资源。

在头文件 semaphore.h 中定义的信号量则完成了互斥体和条件变量的封装，按照多线程程序设计中访问控制机制，控制对资源的同步访问，提供程序设计人员更方便的调用接口。

sem_init(sem_t *sem, int pshared, unsigned int val);

这个函数初始化一个信号量 sem 的值为 val，参数 pshared 是共享属性控制，表明是否在进程间共享。

sem_wait(sem_t *sem);

调用该函数时，若 sem 为无状态，调用线程阻塞，等待信号量 sem 值增加 (post)成为有信号状态；若 sem 为有状态，调用线程顺序执行，但信号量的值减一。

sem_post(sem_t *sem);

调用该函数，信号量 sem 的值增加，可以从无信号状态变为有信号状态。

4.实例

下面我们还是以著名的生产者/消费者问题为例来阐述 Linux 线程的控制和通信。一组生产者线程与一组消费者线程通过缓冲区发生联系。生产者线程将生产的产品送入缓冲区，消费者线程则从中取出产品。缓冲区有 N 个，是一个环形的缓冲池。

Cpp 代码

```cpp
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 16 // 缓冲区数量
struct prodcons
{
    // 缓冲区相关数据结构
    int buffer[BUFFER_SIZE]; /* 实际数据存放的数组*/
    pthread_mutex_t lock; /* 互斥体 lock 用于对缓冲区的互斥操作 */
    int readpos, writepos; /* 读写指针*/
    pthread_cond_t notempty; /* 缓冲区非空的条件变量 */
    pthread_cond_t notfull; /* 缓冲区未满的条件变量 */
};
/* 初始化缓冲区结构 */
void init(struct prodcons *b)
{
    pthread_mutex_init(&b->lock, NULL);
    pthread_cond_init(&b->notempty, NULL);
    pthread_cond_init(&b->notfull, NULL);
    b->readpos = 0;
    b->writepos = 0;
}
/* 将产品放入缓冲区,这里是存入一个整数*/
void put(struct prodcons *b, int data)
{
```

```
33      pthread_mutex_lock(&b->lock);
34      /* 等待缓冲区未满*/
35      if ((b->writepos + 1) % BUFFER_SIZE == b->readpos)
36      {
37          pthread_cond_wait(&b->notfull, &b->lock);
38      }
39      /* 写数据,并移动指针 */
40      b->buffer[b->writepos] = data;
41      b->writepos++;
42      if (b->writepos > = BUFFER_SIZE)
43          b->writepos = 0;
44      /* 设置缓冲区非空的条件变量*/
45      pthread_cond_signal(&b->notempty);
46      pthread_mutex_unlock(&b->lock);
47  }
48  /* 从缓冲区中取出整数*/
49  int get(struct prodcons *b)
50  {
51      int data;
52      pthread_mutex_lock(&b->lock);
53      /* 等待缓冲区非空*/
54      if (b->writepos == b->readpos)
55      {
56          pthread_cond_wait(&b->notempty, &b->lock);
57      }
58      /* 读数据,移动读指针*/
59      data = b->buffer[b->readpos];
60      b->readpos++;
61      if (b->readpos > = BUFFER_SIZE)
62          b->readpos = 0;
63      /* 设置缓冲区未满的条件变量*/
64      pthread_cond_signal(&b->notfull);
65      pthread_mutex_unlock(&b->lock);
66      return data;
67  }
68
69  /* 测试:生产者线程将1 到10000 的整数送入缓冲区,消费者线
70  程从缓冲区中获取整数,两者都打印信息*/
71  #define OVER ( - 1)
72  struct prodcons buffer;
73  void *producer(void *data)
74  {
75      int n;
76      for (n = 0; n < 10000; n++)
```

```
77      {
78          printf("%d --->\n", n);
79          put(&buffer, n);
80      } put(&buffer, OVER);
81      return NULL;
82  }
83
84  void *consumer(void *data)
85  {
86      int d;
87      while (1)
88      {
89          d = get(&buffer);
90          if (d == OVER)
91              break;
92          printf("--->%d \n", d);
93      }
94      return NULL;
95  }
96
97  int main(void)
98  {
99      pthread_t th_a, th_b;
100     void *retval;
101     init(&buffer);
102     /* 创建生产者和消费者线程*/
103     pthread_create(&th_a, NULL, producer, 0);
104     pthread_create(&th_b, NULL, consumer, 0);
105     /* 等待两个线程结束*/
106     pthread_join(th_a, &retval);
107     pthread_join(th_b, &retval);
108     return 0;
109 }
```

5.WIN32、VxWorks、Linux 线程类比

目前为止，笔者已经创作了《基于嵌入式操作系统 VxWorks 的多任务并发程序设计》(《软件报》2006年5~12期连载)、《深入浅出 Win32多线程程序设计》(天极网技术专题)系列，我们来找出这两个系列文章与本文的共通点。

看待技术问题要瞄准其本质，不管是 Linux、VxWorks 还是 WIN32，其涉及到多线程的部分都是那些内容，无非就是线程控制和线程通信，它们的许多函数只是名称不同，其实质含义是等价的，下面我们来列个三大操作系统共同点详细表单：

事项	WIN32	VxWorks	Linux
线程创建	CreateThread	taskSpawn	pthread_create
线程终止	执行完成后退出；线程自身调用 ExitThread 函数即终止自己；被其他线程调用函数 TerminateThread 函数	执行完成后退出；由线程本身调用 exit 退出；被其他线程调用函数 taskDelete 终止	执行完成后退出；由线程本身调用 pthread_exit 退出；被其他线程调用函数 pthread_cance 终止
获取线程 ID	GetCurrentThreadId	taskIdSelf	pthread_self

创建互斥	CreateMutex	semMCreate	pthread_mutex_init
获取互斥	WaitForSingleObject、WaitForMultipleObjects	semTake	pthread_mutex_lock
释放互斥	ReleaseMutex	semGive	phtread_mutex_unlock
创建信号量	CreateSemaphore	semBCreate、semCCreate	sem_init
等待信号	WaitForSingleObject	semTake	sem_wait

量

释

放

信 ReleaseSemaphore semGive sem_post

号

量

6.小结

本章讲述了 Linux 下多线程的控制及线程间通信编程方法，给出了一个生产者/消费者的实例，并将 Linux 的多线程与 WIN32、VxWorks 多线程进行了类比，总结了一般规律。鉴于多线程编程已成为开发并发应用程序的主流方法，学好本章的意义也便不言自明。

Linux 下的 C 编程实战之(五)

1.引言

设备驱动程序是操作系统内核和机器硬件之间的接口，它为应用程序屏蔽硬件的细节，一般来说，Linux 的设备驱动程序需要完成如下功能：

(1)初始化设备；

(2)提供各类设备服务；

(3)负责内核和设备之间的数据交换；

(4)检测和处理设备工作过程中出现的错误。

妙不可言的是，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 Windows 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。本系列文章的第2章文件系统编程中已经看到了这些函数的真面目， 它们就是 open ()、close ()、read ()、write () 等。

Linux 主要将设备分为二类：字符设备和块设备(当然网络设备及 USB 等其它设备的驱动编写方法又稍有不同)。这两类设备的不同点在于：在对字符设 备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了，而块设备则不然，它利用一块系统内存作缓冲区，当用户进程对设备请求能满足用户的要求，就返回 请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备主要针对磁盘等慢速设备。以字符设备的驱动较为简单，因此本章主要阐述字符设备的驱 动编写。

2.驱动模块函数

init 函数用来完成对所控设备的初始化工作，并调用 register_chrdev() 函数注册字符设备。假设有一字符设备“exampledev”，则其 init 函数为：

Cpp 代码

```
void exampledev_init(void)
{
  if (register_chrdev(MAJOR_NUM, " exampledev ", &exampledev_fops))
TRACE_TXT("Device exampledev driver registered error");
  else
TRACE_TXT("Device exampledev driver registered successfully");
...//设备初始化
}
```

其中，register_chrdev 函数中的参数 MAJOR_NUM 为主设备号,“exampledev”为设备名，exampledev_fops 为包含基本函数入口点的结构体，类型为 file_operations。当执行 exampledev_init 时，它将调用内核函数 register_chrdev，把驱动程序的基本入口点指针存放在内核的字符设备地址表中，在用户进程对该设备执行系统调用时提供入口地址。

file_operations 结构体定义为：

Cpp 代码

```
struct file_operations
{
  int (*lseek)();
  int (*read)();
  int (*write)();
  int (*readdir)();
  int (*select)();
  int (*ioctl)();
  int (*mmap)();
```

```
18    int (*open)();
19    void(*release)();
20    int (*fsync)();
21    int (*fasync)();
22    int (*check_media_change)();
23    void(*revalidate)();
24  };
```

大多数的驱动程序只是利用了其中的一部分，对于驱动程序中无需提供的功能，只需要把相应位置的值设为 NULL。对于字符设备来说，要提供的主要入口有：open ()、release ()、read ()、write ()、ioctl ()。

open()函数　对设备特殊文件进行 open()系统调用时，将调用驱动程序的 open () 函数：

int open(struct inode * inode ,struct file * file);

其中参数 inode 为设备特殊文件的 inode (索引结点) 结构的指针，参数 file 是指向这一设备的文件结构的指针。open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用 MINOR(inode-> i - rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0表示成功，负数表示存在错误) 等；

release()函数　当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release () 函数：

void release (struct inode * inode ,struct file * file) ;

release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。

read()函数　当对设备特殊文件进行 read() 系统调用时，将调用驱动程序 read() 函数：

void read(struct inode * inode ,struct file * file ,char * buf ,int count) ;

参数 buf 是指向用户空间缓冲区的指针，由用户进程给出，count 为用户进程要求读取的字节数，也由用户给出。

read() 函数的功能就是从硬设备或内核内存中读取或复制 count 个字节到 buf 指定的缓冲区中。在复制数据时要注意，驱动程序运行在内核中，而 buf 指定的缓冲区在用户内存区中，是不能直接在内核中访问使用的，因此，必须使用特殊的 复制函数来完成复制工作，这些函数在<asm/ segment.h>中定义：

Cpp 代码

```
25 void put_user_byte (char data_byte ,char * u_addr) ;
26 void put_user_word (short data_word ,short * u_addr) ;
27 void put_user_long(long data_long ,long * u_addr) ;
28 void memcpy_tofs (void * u_addr ,void * k_addr ,unsigned long cnt) ;
```

参数 u_addr 为用户空间地址，k_addr 为内核空间地址，cnt 为字节数。

write( ) 函数　当设备特殊文件进行 write () 系统调用时，将调用驱动程序的 write () 函数：

void write (struct inode * inode ,struct file * file ,char * buf ,int count) ;

write ()的功能是将参数 buf 指定的缓冲区中的 count 个字节内容复制到硬件或内核内存中，和 read() 一样，复制工作也需要由特殊函数来完成：

Cpp 代码

```
unsigned char_get_user_byte (char * u_addr) ;
unsigned char_get_user_word (short * u_addr) ;
unsigned char_get_user_long(long * u_addr) ;
unsigned memcpy_fromfs(void * k_addr ,void * u_addr ,unsigned long cnt) ;
```

ioctl() 函数 该函数是特殊的控制函数，可以通过它向设备传递控制信息或从设备取得状态信息，函数原型为：

int ioctl (struct inode * inode ,struct file * file ,unsigned int cmd ,unsigned long arg);

参数 cmd 为设备驱动程序要执行的命令的代码，由用户自定义，参数 arg 为相应的命令提供参数，类型可以是整型、指针等。

同样，在驱动程序中，这些函数的定义也必须符合命名规则，按照本文约定，设备“exampledev”的驱动程序的这些函数应分别命名为 exampledev_open、exampledev_ release、exampledev_read、exampledev_write、exampledev_ioctl，因此设备 “exampledev”的基本入口点结构变量 exampledev_fops 赋值如下：

Cpp 代码

```
struct file_operations exampledev_fops {
  NULL ,
  exampledev_read ,
  exampledev_write ,
  NULL ,
```

```
38    NULL ,
39    exampledev_ioctl ,
40    NULL ,
41    exampledev_open ,
42    exampledev_release ,
43    NULL ,
44    NULL ,
45    NULL ,
46    NULL
47  } ;
```

3.内存分配

由于 Linux 驱动程序在内核中运行，因此在设备驱动程序需要申请/释放内存时，不能使用用户级的 malloc/free 函数，而需由内核级的函数 kmalloc/kfree () 来实现，kmalloc()函数的原型为：

void kmalloc (size_t size ,int priority);

参数 size 为申请分配内存的字节数；参数 priority 说明若 kmalloc()不能马上分配内存时用户进程要采用的动作：GFP_KERNEL 表示等待，即等 kmalloc()函数将一些内存安排到交换区来满足你的内存需要，GFP_ATOMIC 表示不等待，如不能立即分配到内存则返回0 值；函数的返回值指向已分配内存的起始地址，出错时，返回0。

kmalloc ()分配的内存需用 kfree()函数来释放，kfree ()被定义为：

define kfree (n) kfree_s((n) ,0)

其中 kfree_s () 函数原型为：

void kfree_s (void * ptr ,int size);

参数 ptr 为 kmalloc()返回的已分配内存的指针，size 是要释放内存的字节数，若为0 时，由内核自动确定内存的大小。

4.中断

许多设备涉及到中断操作，因此，在这样的设备的驱动程序中需要对硬件产生的中断请求提供中断服务程序。与注册基本入口点一样，驱动程序也要请求内核将特定的中断请求和中断服务程序联系在一起。在 Linux 中，用 request_irq()函数来实现请求：

```
int request_irq (unsigned int irq ,void( * handler) int ,unsigned long
type ,char * name);
```

参数 irq 为要中断请求号，参数 handler 为指向中断服务程序的指针，参数 type 用来确定是正常中断还是快速中断(正常中断指中断服务子程序返回后，内核可以执行调度程序来确定将运行哪一个进程；而快速中断是指中断服务子程序返回后， 立即执行被中断程序，正常中断 type 取值为0 ，快速中断 type 取值为 SA_INTERRUPT)，参数 name 是设备驱动程序的名称。

5.实例

笔者最近设计了一块采用三星 S3C2410 ARM 处理器的电路板(ARM 处理器广泛应用于手机、PDA 等嵌入式系统)，板上包含四个用户可编程的发光二极

管(LED)，这些 LED 连接在 ARM 处理 器的可编程 I/O 口(GPIO)上。下图给出了 ARM 中央处理器与 LED 的连接原理：

我们在 ARM 处理器上移植 Linux 操作系统，现在来编写这些 LED 的驱动：

Cpp 代码

```
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/spinlock.h>
#include <linux/irq.h>
#include <linux/delay.h>
#include <asm/hardware.h>
#define DEVICE_NAME "leds" /*定义 led 设备的名字*/
#define LED_MAJOR 231 /*定义 led 设备的主设备号*/
static unsigned long led_table[] =
{
  /*I/O 方式 led 设备对应的硬件资源*/
  GPIO_B10, GPIO_B8, GPIO_B5, GPIO_B6,
};
/*使用 ioctl 控制 led*/
static int leds_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg)
{
  switch (cmd)
  {
case 0:
case 1:
  if (arg > 4)
  {
return -EINVAL;
  }
  write_gpio_bit(led_table[arg], !cmd);
default:
  return -EINVAL;
  }
```

```
83 }
84 static struct file_operations leds_fops =
85 {
86   owner: THIS_MODULE, ioctl: leds_ioctl,
87 };
88 static devfs_handle_t devfs_handle;
89 static int __init leds_init(void)
90 {
91   int ret;
92   int i;
93   /*在内核中注册设备*/
94   ret = register_chrdev(LED_MAJOR, DEVICE_NAME, &leds_fops);
95   if (ret < 0)
96   {
97 printk(DEVICE_NAME " can't register major number\n");
98 return ret;
99   }
100   devfs_handle = devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT,
LED_MAJOR,
101 0, S_IFCHR | S_IRUSR | S_IWUSR, &leds_fops, NULL);
102   /*使用宏进行端口初始化，set_gpio_ctrl 和 write_gpio_bit 均为宏定义*/
103   for (i = 0; i < 8; i++)
104   {
105 set_gpio_ctrl(led_table[i] | GPIO_PULLUP_EN | GPIO_MODE_OUT);
106 write_gpio_bit(led_table[i], 1);
107   }
108   printk(DEVICE_NAME " initialized\n");
109   return 0;
110 }
111
112 static void __exit leds_exit(void)
113 {
114   devfs_unregister(devfs_handle);
115   unregister_chrdev(LED_MAJOR, DEVICE_NAME);
116 }
117
118 module_init(leds_init);
119 module_exit(leds_exit);
```

使用命令方式编译 led 驱动模块：

#arm-linux-gcc -D__KERNEL__ -I/arm/kernel/include

-DKBUILD_BASENAME=leds -DMODULE -c -o leds.o leds.c

以上命令将生成 leds.o 文件，把该文件复制到板子的/lib 目录下，使用以下命令就可以安装 leds 驱动模块：

#insmod /lib/ leds.o

删除该模块的命令是：

#rmmod leds

6.小结

本章讲述了 Linux 设备驱动程序的入口函数及驱动程序中的内存申请、中断等，并给出了一个通过 ARM 处理器的 GPIO 口控制 LED 的驱动实例。