第一篇：起步篇

千里之行，始于足下。

第一章 UNIX初步

UNIX 自 1969 年诞生以来，已经发展为 System III & V、BSD 和 Linux 三大分支。

UniX 通过 shell 与用户交互，它是用户与系统间的界面。使用好 shell 对于学习使用 UNIX 来说是必须的。不需要你记住所有的命令，但基础的文件操作、目录操作及系统命令等却是必须的。

Vi 对于初学者是难点，不过只要通过一段时间的练习就能习惯；而且你会很快发现，它功能强大、更加灵活。这里不多说了：）

第二章 编程套件

学习 UNIX 对于初学者有几种选择。一种就是最直接的，在本机上安装 UNIX 系统，不如说 Linux 分支中的 Red Hat 等；一种是在 Windows 环境下使用虚拟机方式安装 UNIX 系统；另一种则是使用网络终端登录到网络环境中的某个 UNIX 系统中。

如果你相对黑洞洞的 UNIX 字符界面更喜欢舒适美观的 Windows 界面，那么推荐你选择第三种方式。开发套件包括：编辑器 UltraEdit、网络终端 SecureCRT 或其他。

编译器就不用说了，自然是 cc（gcc 或 xlc 系列）。这是 C 语言开发必不可少的。其中要注意-I（加载头文件路径）、-L（加载库文件路径）及-D（宏定义）参数的使用。

Make 工具使用。如何编写 makefile 是关键。后面项目中使用时会详细介绍。

Gdb 调试器。Gdb 乃符号级调试工具，它控制程序的内部执行，利用断点设置、单步运行等手段，将程序的执行过程逐步展示在调试者目前。这种调试方式在短代码中可以发挥得很好。事实上，随着软件项目的扩大化、复杂化和分布化，很少有程序员直接通过 Gdb 等工具调试；使用日志记录调试方法比 Gdb 等调试工具更为便捷和广泛。

C 工具：lint 检查源代码是否正确，gprof 分析程序时间消费量，cflow 生成 C 语言流程图。

第三章 库的使用

库分静态库和动态库两种。

静态库的操作工具：ar 命令。

编写及使用静态库

(1)设计库源码 pr1.c 和 pr2.c

```
[root@billstone make_lib]# cat pr1.c
void print1()
{
```

```
        printf("This is the first lib src!\n");
}
[root@billstone make_lib]# cat pr2.c
void print2()
{
        printf("This is the second src lib!\n");
}
```

(2) 编译.c 文件

```
[bill@billstone make_lib]$ cc -O -c pr1.c pr2.c
[bill@billstone make_lib]$ ls -l pr*.o
-rw-rw-r--    1 bill    bill          804   4 月  15 11:11 pr1.o
-rw-rw-r--    1 bill    bill          804   4 月  15 11:11 pr2.o
```

(3) 链接静态库

为了在编译程序中正确找到库文件,静态库必须按照 lib[name].a 的规则命名,如下例中[name]=pr.

```
[bill@billstone make_lib]$ ar -rsv libpr.a pr1.o pr2.o
a - pr1.o
a - pr2.o
[bill@billstone make_lib]$ ls -l *.a
-rw-rw-r--    1 bill    bill         1822   4 月  15 11:12 libpr.a
[bill@billstone make_lib]$ ar -t libpr.a
pr1.o
pr2.o
```

(4) 调用库函数代码 main.c

```
[bill@billstone make_lib]$ cat main.c
int main()
{
        print1();
        print2();

        return 0;
}
```

(5) 编译链接选项

-L 及-l 参数放在后面.其中,-L 加载库文件路径,-l 指明库文件名字.

```
[bill@billstone make_lib]$ gcc -o main main.c -L./ -lpr
[bill@billstone make_lib]$ ls -l main*
-rwxrwxr-x    1 bill    bill        11805   4 月  15 11:17 main
-rw-rw-r--    1 bill    bill           50   4 月  15 11:15 main.c
```

(6)执行目标程序

```
[bill@billstone make_lib]$ ./main
This is the first lib src!
This is the second src lib!
[bill@billstone make_lib]$
```

编写动态库

(1)设计库代码

```
[bill@billstone make_lib]$ cat pr1.c
int p = 2;

void print(){
        printf("This is the first dll src!\n");
}
[bill@billstone make_lib]$
```

(2)生成动态库

```
[bill@billstone make_lib]$ gcc -O -fpic -shared -o dl.so pr1.c
[bill@billstone make_lib]$ ls -l *.so
-rwxrwxr-x      1 bill      bill            6592    4 月  15 15:19 dl.so
[bill@billstone make_lib]$
```

动态库的隐式调用

在编译调用库函数代码时指明动态库的位置及名字, 看下面实例

```
[bill@billstone make_lib]$ cat main.c
int main()
{
        print();

        return 0;
}
[bill@billstone make_lib]$ gcc -o tdl main.c ./dl.so
[bill@billstone make_lib]$ ./tdl
This is the first dll src!
[bill@billstone make_lib]$
```

当动态库的位置活名字发生改变时, 程序将无法正常运行; 而动态库取代静态库的好处之一则是通过更新动态库而随时升级库的内容.

动态库的显式调用

显式调用动态库需要四个函数的支持, 函数 dlopen 打开动态库, 函数 dlsym 获取动态库中对象基址, 函数 dlerror 获取显式动态库操作中的错误信息, 函数 doclose 关闭动态库.

```
[bill@billstone make_lib]$ cat main.c
#include <dlfcn.h>


int main()
{
        void *pHandle;
        void (*pFunc)();                        // 指向函数的指针
        int *p;


        pHandle = dlopen("./d1.so", RTLD_NOW);          // 打开动态库
        if(!pHandle){
                printf("Can't find d1.so \n");
                exit(1);
        }
        pFunc = (void (*)())dlsym(pHandle, "print");          // 获取库函数 print 的地址
        if(pFunc)
                pFunc();
        else
                printf("Can't find function print\n");
        p = (int *)dlsym(pHandle, "p");                       // 获取库变量 p 的地址
        if(p)
                printf("p = %d\n", *p);
        else
                printf("Can't find int p\n");
        dlclose(pHandle);                                     // 关闭动态库


        return 0;
}
[bill@billstone make_lib]$ gcc -o tds main.c -ldl
[bill@billstone make_lib]$ ./tds
```

```
This is the first dll src!
p = 2
[bill@billstone make_lib]$
```

上面的程序 tds 显式调用了共享库 d1.so 中的函数 print 和变量 p.

第二篇: 文件子系统

普天之下,莫非王土;率土之滨,莫非王臣. UNIX 之中,莫非文件.

第四章 文件系统结构

磁盘在使用前, 需要分区和格式化. 格式化操作将在磁盘分区中创建文件系统, 它们将确定文件的存储方式和索引方法, 确定磁盘空间分配和回收算法.

UNIX 文件系统的存储由<目录-*i* 节点-数据块>三级构成,其中目录存储了文件的层次结构, 数据块存储了文件的具体信息, i 节点是连接文件层次结构与其数据内容的桥梁.

UNIX 文件系统将磁盘空间划分为一系列大小相同的块, 划分为引导块, 超级块, i 节点区和数据区四个部分.

文件系统通过 i 节点对文件进行控制和管理. 其中, 每个文件对应一个 i 节点, 每个 i 节点具有唯一的节点号, 记录了文件的属性和内容在磁盘上的存储位置. 但文件名并不记录在 i 节点里, 而是存储在目录文件中.

磁盘文件如何存储?

文件系统通过目录记载文件名及其对应的 i 节点编号, 通过 i 节点记录文件的信息和内容. 事实上, i 节点直接记录的只是文件的属性, 文件的具体内容存储在数据区的数据块中, i 节点中仅保留了一个<磁盘地址表>来记录文件内容存储的位置.

<磁盘文件表>由 13 个块号组成, 每个块号占用 4 个字节, 代表了数据区中的一个数据块编号.UNIX 文件系统采用三级索引结构存储文件,它把<磁盘地址表>分为直接索引地址, 一级索引地址, 二级索引地址和三级索引地址等四个部分. 其中前 10 项为直接索引地址, 直接指向文件数据所在磁盘快的块号. 第 11/12/13 项分别为一级/二级/三级索引地址. 一级间接索引的含义在于其存储的并非文件数据所在磁盘块的块号, 而是先指向一个<磁盘块号表>然后再指向具体磁盘块的块号. 同理, 二级/三级间接索引则是先间接指向了两次<磁盘块号表>才指向具体磁盘快的块号.

如果文件系统的数据块大小为 1kB, 每个<磁盘块号表>能够记录 256 个数据项. 那么, 直接索引能管辖 10 个数据块, 而一级索引能管辖 1*256 个数据块, 二级索引能管辖 1*256*256(65536)个数据块, 三级索引能管辖 1*256*256*256(16777216)个数据块.

例题: 大小为 56000K 的文件,占用多少索引块空间?

答: 因为(10+256) < 56000 < (10+256+65536), 故该文件具有二级间接索引. (56000-10-256)/256=217.7, 则文件需要二级间接索引块为 218 个,所以总索引块需要 1(一级间接索引块)+1(二级间接索引块)+218=220.

磁盘文件读取示例(仿 ls 命令)

通过 stat 结构中 st_mode 判断文件类型

```
int GetFileType(mode_t st_mode, char *resp){
```

```
        if(resp == NULL)
                return 0;
        if(S_ISDIR(st_mode))      resp[0] = 'd';              // 使用宏定义判断
        else if(S_ISCHR(st_mode))         resp[0] = 'c';
        else if(S_ISBLK(st_mode))         resp[0] = 'b';
        else if(S_ISREG(st_mode))         resp[0] = '-';
        else if(S_ISFIFO(st_mode))        resp[0] = 'p';
        else if(S_ISLNK(st_mode))         resp[0] = 'l';
        else resp[0] = ' ';

        return 1;
}
```

同样,通过 st_mode 判断文件访问权限

```
int GetFileMode(mode_t st_mode, char *resp){
        if(resp == NULL)
                return 0;
        memset(resp, '-', 9);
        if(st_mode & S_IRUSR)    resp[0] = 'r';          // 使用各种宏定义与 st_mode 做与处理判断
        if(st_mode & S_IWUSR)    resp[1] = 'w';
        if(st_mode & S_IXUSR)    resp[2] = 'x';
        if(st_mode & S_IRGRP)    resp[3] = 'r';
        if(st_mode & S_IWGRP)    resp[4] = 'w';
        if(st_mode & S_IXGRP)    resp[5] = 'x';
        if(st_mode & S_IROTH)    resp[6] = 'r';
        if(st_mode & S_IWOTH)    resp[7] = 'w';
        if(st_mode & S_IXOTH)    resp[8] = 'x';

        return 9;
}
```

处理文件其他属性如下

```
int GetFileOtherAttr(struct stat info, char *resp){
        struct tm *mtime;

        if(resp == NULL)
                return 0;
        mtime = localtime(&info.st_mtime);
                        // 按 ls 命令显示顺序处理其他属性
```

```
        return(sprintf(resp, " %3d %6d %6d %11d %04d%02d%02d", info.st_nlink, info.st_uid, \
                info.st_gid, info.st_size,mtime->tm_year+1900, mtime->tm_mon+1, mtime->tm_mday));
}
```

设计类似于 UNIX 命令<ls -l>的程序 lsl, 主程序如下

```
[bill@billstone Unix_study]$ cat lsl.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>

int GetFileType(mode_t st_mode, char *resp);
int GetFileMode(mode_t st_mode, char *resp);
int GetFileOtherAttr(struct stat info, char *resp);

int main(int argc, char **argv)
{
        struct stat info;
        char buf[100], *p = buf;

        if(argc != 2){
                printf("Usage: lsl filename\n");
                return;
        }
        memset(buf, 0, sizeof(buf));
        if(lstat(argv[1], &info) == 0){
                p += GetFileType(info.st_mode, p);
                p += GetFileMode(info.st_mode, p);
                p += GetFileOtherAttr(info, p);
                printf("%s %s\n", buf, argv[1]);
        }
        else
                printf("Open file failed!\n");

        return 0;
}
```

运行结果如下:

```
[bill@billstone Unix_study]$ make lsl
cc      lsl.c    -o lsl
```

```
[bill@billstone Unix_study]$ ./lsl

Usage: lsl filename

[bill@billstone Unix_study]$ ./lsl /etc/passwd

-rw-r--r--    1        0        0            1639 20090328 /etc/passwd

[bill@billstone Unix_study]$ ls -l /etc/passwd

-rw-r--r--      1 root       root             1639   3 月  28 16:38 /etc/passwd
```

第五章 标准文件编程库

在 UNIX 的应用中, 读写文件是最常见的任务. 标准文件编程库就是操作文件最简单的工具.

标准编程函数库对文件流的输入输出操作非常灵活, 我们既可以采用所见即所得的方式, 以无格式方式读写文件, 又可以对输入输出数据进行转化, 以有格式方式读写文件.

文件的无格式读写

无格式读写分三类: 按字符读写, 按行读写和按块读写.

字符读写函数族:

```
#include <stdio.h>

int getc(FILE *stream);

int fgetc(FILE *stream);

int putc(int c, FILE *stream);

int fputc(int c, FILE *stream);
```

函数 fgetc 的功能类似于 getc, 不同的是, 它的执行速度远低于 getc.

行读写函数族:

```
#include <stdio.h>

char *gets(char *s);

char *fgets(char *s, int n, FILE *stream);

int puts(const char *s);

int fputs(const char *s, FILE *stream);
```

函数 fgets 中加入了放溢出控制, 应该优先选用. 注意函数 fputs 把字符串 s(不包括结束符'\0')写入文件流 stream 中, 但不在输出换行符'\n'; 而函数 puts 则自动输出换行符.

块读写函数族:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

函数 fread 和 fwrite 都不返回实际读写的字符个数, 而返回的是实际读写的数据项数.块读写函数常用于保存和恢复内存数据.

文件的格式化读写

文件格式化读写时能够自动转换的数据格式有: 数据类型, 精度, 宽度, 进制和标志等,而其一般格式为

```
% [标志] [宽度] [.精度] 类型
```

格式化输出函数族

```
#include <stdio.h>

int printf(const char *format, /* [arg,] */ . . .);

int fprintf(FILE *stream, const char *format, /* [arg,] */ . . .);

int sprintf(char *s, const char *format, /* [arg,] */ . . .);
```

在做字符串处理时应该善用 sprintf 函数.

格式化输入函数族

```
#include <stdio.h>

int scanf(const char format, /* [pointer,] */ . . .);

int fscanf(FILE *stream, const char format, /* [pointer,] */ . . .);

int sscanf(const char *s, const char format, /* [pointer,] */ . . .);
```

二进制读写与文本读写

记得刚开始学习 C 语言的文件操作时, 这是一个最让我疑惑的问题. 我们都知道在调用 fopen 函数时需要指定操作类型, 比如说文本写'r'和二进制写'rb'.

那么它们究竟有何区别呢? 这要牵涉到两种存储方式: 以字符为单位存储的文本文件和以二进制数据为单位存储的二进制文件. 举个例子: 我们通常阅读的 Readme.txt 文件就是文本文件, 该类型文件存储的是一个一个的字符, 这些字符往往是可以打印的; 而我们的可执行程序(比如 a.out)则是二进制文件, 该文件是不可读的,需要解析才能识别.

那么在调用 fopen 函数时该如何选择呢? 如果你是先写入再从另外的地方读出, 那么两种方式都可以; 只要按写入时的方式读取就可以了. 但是, 比起文本方式, 二进制方式在保存信息时有着优势:

a) 加快了程序的执行速度, 提高了软件的执行效率. 内存中存储的都是二进制信息, 直接以二进制方式与文件交互, 可以免除二进制格式与文本格式之间的信息转换过程.

b) 节省了存储空间. 一般来讲, 二进制信息比文件信息占用更少的空间, 比如 8 位的整型数采用文本方式存储至少需要 8 字节, 而采用二进制存储只需一个整型即 4 个字节.

编写变长参数函数

文件的格式化输入输出函数都支持变长参数. 定义时, 变长参数列表通过省略号'...'表示, 因此函数定义格式为:

type 函数名(参数 1, 参数 2, 参数 n, . . .);

UNIX 的变长参数通过 va_list 对象实现, 定义在文件'stdarg.h'中, 变长参数的应用模板如下所示:

```
#include <stdarg.h>


function(parmN, ...){
        va_list pvar;
        ..............................
        va_start(pvar, parmN);
        while()
        {
                ..................
```

```
                f = va_arg(pvar, type);
                ..................
        }
        va_end(pvar);
}
```

va_list 数据类型变量 pvar 访问变长参数列表中的参数. 宏 va_start 初始化变长参数列表, 根据 parmN 判断参数列表的起始位置. va_arg 获取变长列表中参数的值, type 指示参数的类型, 也使宏 va_arg 返回数值的类型. 宏 va_arg 执行完毕后自动更新对象 pvar, 将其指向下一个参数. va_end 关闭对变长参数的访问.

下面给出一个实例 mysum, 计算输入参数的和并返回

```
[bill@billstone Unix_study]$ cat mysum.c
#include <stdarg.h>


int mysum(int i, ...){             // 参数列表中, 第一个参数指示累加数的个数
        int r = 0, j = 0;
        va_list pvar;


        va_start(pvar, i);
        for(j=0;j<i;j++){
                r += va_arg(pvar, int);
        }
        va_end(pvar);


        return(r);
}


int main()
{
        printf("sum(1,4) = %d\n", mysum(1,4));
        printf("sum(2,4,8) = %d\n", mysum(2,4,8));


        return 0;
}
[bill@billstone Unix_study]$ make mysum
cc      mysum.c    -o mysum
[bill@billstone Unix_study]$ ./mysum
sum(1,4) = 4
sum(2,4,8) = 12
[bill@billstone Unix_study]$
```

第六章 低级文件编程库

低级文件编程库常常用于访问终端, 管道, 设备和套接字等特殊文件, 一般不用于普通磁盘文件, 这是标准文件编程库的特长.

低级文件编程库听起来似乎低级, 其实它是 UNIX 中的 I/O 系统调用. 它们使用文件描述符, 直接读写各类文件.

低级文件编程库在输入输出上只有块读写的功能.

文件锁

多用户多任务操作系统非常重要的一个内容就是文件锁. 用户在更新文件时, 期望可以使用某种机制, 防止两进程同时更新文件同一区域而造成写丢失, 或者防止文件内容在未更新完毕时被读取等并发引起的问题, 这种机制就是文件锁.

进程在操作文件期间, 可以使用文件锁, 锁定文件中的敏感信息, 防止其他进程越权操作该部分信息. 函数 fcntl 提供了对文件任意区域设置锁的能力, 既可以锁住全部文件, 又可以锁住文件的部分记录, 故文件锁又称为'记录锁'.

根据文件锁的访问方式, 可以区分为读锁和写锁两种. 文件记录在同一时刻, 可以设置多个读锁, 但仅能设置一个写锁, 并且读写锁不能同时存在.

当函数 fcntl 专用于锁操作时, 其原型为

```
int fcntl(int fildes, int cmd, struct flock *arg);
```

其中, 结构 flock 用于描述文件锁的信息, 定义于头文件'fcntl.h'中, 如下所示

```
struct flock {
        short l_type;     // 锁类型, 取值为 F_RDLCK, F_WRLCK 或 F_UNLCK 之一
        short l_whence;   // 锁区域开始地址的相对位置, 取值为 SEEK_SET, SEEK_CUR 或 SEEK_END 之一
        off_t l_start;    // 锁区域开始地址偏移量
        off_t l_len;      // 锁区域的长度, 0 表示锁至文件末
        pid_t l_pid;      // 拥有锁的进程 ID 号
};
```

函数 fcntl 在专用于锁操作时, 参数 cmd 有三种取值:

(a) F_GETLK. 获取文件描述符 fileds 对应文件指定区域的文件锁信息.

(b) F_SETLK. 在文件描述符 fileds 对应的文件中指定区域设置锁信息.

(c) F_SETLKW. 该命令是 F_SETLK 命令的阻塞版本.

文件锁最典型应用于两个方面: 一是锁定文件中的临界数据, 比如并发投票时文件记录的投票数; 二是利用具有互斥性质的写锁, 实现进程的并发控制.

在锁机制的使用中,最常见的操作有锁的请求, 释放和测试等, 下面一一说明.

(a) 测试锁. 设计函数 SeeLock, 查询文件描述符 fd 对应文件的锁信息.

```
void SeeLock(int fd, int start, int len)
{                                   // 查询描述符 fd 对应文件从 start 处开始的 len 字节中的锁信息
        struct flock arg;


        arg.l_type = F_WRLCK;
        arg.l_whence = SEEK_SET;
        arg.l_start = start;
```

```
        arg.l_len = len;


        if(fcntl(fd, F_GETLK, &arg) == -1)
                fprintf(stderr, "See Lock failed.\n");
        else if(arg.l_type == F_UNLCK)
                fprintf(stderr, "No Lock From %d to %d\n", start, start+len);
        else if(arg.l_type == F_WRLCK)
                fprintf(stderr, "Write Lock From %d to %d, id = %d\n", start, start+len, arg.l_pid);
        else if(arg.l_type == F_RDLCK)
                fprintf(stderr, "Read Lock From %d to %d, id = %d\n", start, start+len, arg.l_pid);
}
```

(b) 申请读锁. 以阻塞方式设计共享读锁申请函数 GetReadLock.

```
void GetReadLock(int fd, int start, int len)
{                       // 以阻塞方式在描述符 fd 对应文件中从 start 处的 len 字节上申请共享读锁
        struct flock arg;


        arg.l_type = F_RDLCK;
    arg.l_whence = SEEK_SET;
    arg.l_start = start;
    arg.l_len = len;


    if(fcntl(fd, F_SETLKW, &arg) == -1)
        fprintf(stderr, "[%d] See Read Lock failed.\n", getpid());
    else
        fprintf(stderr, "[%d] Set Read Lock From %d to %d\n", getpid(), start, start+len);
}
```

(c) 申请写锁. 以阻塞方式设计互斥写锁申请函数 GetWrtieLock.

```
void GetWriteLock(int fd, int start, int len)
{                       // 以阻塞方式在描述符 fd 对应文件中从 start 处的 len 字节上申请互斥写锁
        struct flock arg;


    arg.l_type = F_WRLCK;
    arg.l_whence = SEEK_SET;
    arg.l_start = start;
    arg.l_len = len;


    if(fcntl(fd, F_SETLKW, &arg) == -1)
```

```
                fprintf(stderr, "[%d] See Write Lock failed.\n", getpid());
        else
                fprintf(stderr, "[%d] Set Write Lock From %d to %d\n", getpid(), start, start+len);
}
```

(d) 释放锁. 设计文件锁释放函数 ReleaseLock.

```
void ReleaseLock(int fd, int start, int len)
{                       // 在描述符 fd 对应文件中释放从 start 处的 len 字节上的锁
        struct flock arg;

        arg.l_type = F_UNLCK;
        arg.l_whence = SEEK_SET;
        arg.l_start = start;
        arg.l_len = len;

        if(fcntl(fd, F_SETLKW, &arg) == -1)
                fprintf(stderr, "[%d] UnLock failed.\n", getpid());
        else
                fprintf(stderr, "[%d] UnLock From %d to %d\n", getpid(), start, start+len);
}
```

下面设计一个文件锁控制进程的实例 lock1. 为了观察阻塞方式下的锁申请, 在释放锁前休眠 30 秒.

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

int main()
{
        int fd;
        struct flock arg;

        if((fd = open("/tmp/tlockl", O_RDWR | O_CREAT, 0755)) < 0){
                fprintf(stderr, "open file failed.\n");
                exit(1);
        }

        SeeLock(fd, 0, 10);
        GetReadLock(fd, 0, 10);
        SeeLock(fd, 11, 20);
        GetWriteLock(fd, 11, 20);
```

```
        sleep(30);
        ReleaseLock(fd, 0, 10);
        ReleaseLock(fd, 11, 20);

        return 0;
}
```

下面是执行情况:

```
[bill@billstone Unix_study]$ make lockl
cc      lockl.c    -o lockl
[bill@billstone Unix_study]$ ./lockl &              // 先在后台执行
[2] 12725
No Lock From 0 to 10
[12725] Set Read Lock From 0 to 10
No Lock From 11 to 31
[12725] Set Write Lock From 11 to 31          // 此后休眠 30 秒
[bill@billstone Unix_study]$ ./lockl              // 再次执行
Read Lock From 0 to 10, id = 12725
[12726] Set Read Lock From 0 to 10            // 可在同一区域申请多个共享读锁
Write Lock From 11 to 31, id = 12725
[12725] UnLock From 0 to 10
[12725] UnLock From 11 to 31
[12726] Set Write Lock From 11 to 31          // 在同一区域只能申请一个互斥写锁
[12726] UnLock From 0 to 10
[12726] UnLock From 11 to 31
[2]+   Done                    ./lockl
[bill@billstone Unix_study]$
```

第七章 目录文件编程库

UNIX 专门给出了一组用于目录操作的函数, 可以方便地获取目录项的确切含义.

工作目录

进程在搜索文件相对路径时都会有一个起始点, 这个起始点称为'当前工作目录'. 在 UNIX 中对工作目录的操作可分为读取工作目录和更改工作目录两种.

(1) 读取工作目录. 函数 getcwd 和 getwd 都返回工作目录的绝对路径

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

```
char *getwd(char *pathname);
```

(2) 更改工作目录.

```
#include <unistd.h>

int chhdir(const char *path);

int fchdir(int fildes);
```

下面是一个读取和更改当前工作目录的例子

```
[bill@billstone Unix_study]$ cat dirl.c

#include <unistd.h>

#include <stdio.h>


int main()

{

        char buf[255];


        fprintf(stderr, "pwd = [%s] \n", getcwd(buf, sizeof(buf)));

        chdir("../");                   // 更改工作目录为上一级目录

        fprintf(stderr, "pwd = [%s] \n", getcwd(buf, sizeof(buf)));


        return 0;

}

[bill@billstone Unix_study]$ make dirl

cc      dirl.c    -o dirl

[bill@billstone Unix_study]$ pwd

/home/bill/Unix_study

[bill@billstone Unix_study]$ ./dirl

pwd = [/home/bill/Unix_study]

pwd = [/home/bill]                          ; 更改成功

[bill@billstone Unix_study]$ pwd

/home/bill/Unix_study                       ; 不影响当前 Shell 的工作目录

[bill@billstone Unix_study]$
```

读取目录

'目录文件编程库'不提倡直接更改目录文件内容, 它仅仅执行读取操作

```
#include <dirent.h>

DIR *opendir(const char *dirname);

struct dirent *readdir(DIR *dirp);
```

```
int closedir(DIR *dirp);
```

函数 opendir 打开目录文件 dirname, 并返回一个目录流, 存储为 DIR 结构.

函数 readdir 读取当前目录项内容存入参数 dirp 指向的结构 dirent 中, 并移动目录文件指针到下一目录项. 目录中每个目录项采用结构 dirent 描述.

```
struct dirent {
        long            d_ino;          // 文件对应 i 节点编号
        __kernel_off_t  d_off;
        unsigned short  d_reclen;
        char            d_name[256];    // 文件名称
};
```

下面是一个简单的读取目录程序 ls2, 它列举了目录下的全部文件及其对应的 i 节点编号.

```
[bill@billstone Unix_study]$ cat ls2.c
#include <stdio.h>
#include <dirent.h>

int main(int argc, char **argv)
{
        DIR *pdir;
        struct dirent *pent;

        if(argc !=2){
                fprintf(stderr, "Usage: ls2 <directory>\n");
                return 0;
        }
        if((pdir = opendir(argv[1])) == NULL){
                fprintf(stderr, "open dir failed.\n");
                exit(1);
        }
        while(1){
                if((pent = readdir(pdir)) == NULL)
                        break;
                fprintf(stderr, "%5d %s\n", pent->d_ino, pent->d_name);
        }
        closedir(pdir);

        return 0;
}
```

执行结果如下:

```
[bill@billstone Unix_study]$ make ls2
cc       ls2.c    -o ls2
[bill@billstone Unix_study]$ ./ls2 /home/bill/Doc
134706 .
    29 ..
134708 学习笔记.doc
[bill@billstone Unix_study]$ ls -ai /home/bill/Doc
 134706 .          29 ..    134708 学习笔记.doc
```

第三篇 并发程序设计

业精于勤, 而荒于嬉.

第九章 进程控制

进程是程序的一次执行, 是运行在自己的虚拟地址空间的一个具有独立功能的程序. 进程是分配和释放资源的基本单位, 当程序执行时, 系统创建进程, 分配内存和 CPU 等资源; 进程结束时, 系统回收这些资源.

线程与进程

线程又名轻负荷进程, 它是在进程基础上程序的一次执行, 一个进程可以拥有多个线程.

线程没有独立的资源, 它共享进程的 ID, 共享进程的资源.

线程是 UNIX 中最小的调度单位, 目前有系统级调度和进程级调度两种线程调度实行方式: 系统级调度的操作系统以线程为单位进行调度; 进程级调度的操作系统仍以进程为单位进行调度, 进程再为其上运行的线程提供调度控制.

环境变量

UNIX 中, 存储了一系列的变量, 在 shell 下执行'env'命令, 就可以得到环境变量列表.

环境变量分为系统环境变量和用户环境变量两种. 系统环境变量在注册时自动设置, 大部分具有特定的含义; 用户环境变量在 Shell 中使用赋值命令和 export 命令设置. 如下例先设置了变量 XYZ, 再将其转化为用户环境变量:

```
[bill@billstone Unix_study]$ XYZ=/home/bill
[bill@billstone Unix_study]$ env | grep XYZ
[bill@billstone Unix_study]$ export XYZ
[bill@billstone Unix_study]$ env | grep XYZ
XYZ=/home/bill
[bill@billstone Unix_study]$
```

UNIX 下 C 程序中有两种获取环境变量值的方法: 全局变量法和函数调用法

(a) 全局变量法

UNIX 系统中采用一个指针数组来存储全部环境值:

```
Extern char **environ;
```

该法常用于将 environ 作为参数传递的语句中, 比如后面提到的 execve 函数等.

```
[bill@billstone Unix_study]$ cat env1.c
#include <stdio.h>
extern char **environ;


int main()
{
        char **p = environ;


        while(*p){
                fprintf(stderr, "%s\n", *p);
                p++;
        }


        return 0;
}
[bill@billstone Unix_study]$ make env1
cc      env1.c    -o env1
[bill@billstone Unix_study]$ ./env1
SSH_AGENT_PID=1392
HOSTNAME=billstone
DESKTOP_STARTUP_ID=
SHELL=/bin/bash
TERM=xterm
... ... ... ...
[bill@billstone Unix_study]
```

(b) 函数调用法

UNIX 环境下操作环境变量的函数如下:

```
#include <stdlib.h>
char *getenv(char *name);
int putenv(const char *string);
```

函数 getenv 以字符串形式返回环境变量 name 的取值, 因此每次只能获取一个环境变量的值; 而且要使用该函数, 必须知道要获取环境变量的名字.

```
[bill@billstone Unix_study]$ cat env2.c
#include <stdio.h>
```

```
#include <stdlib.h>


int main(int argc, char **argv)
{
        int i;


        for(i=1;i<argc;i++)
                fprintf(stderr, "%s=%s\n", argv[i], getenv(argv[i]));


        return 0;
}
[bill@billstone Unix_study]$ make env2
cc      env2.c    -o env2
[bill@billstone Unix_study]$ ./env2 HOME LOGNAME rp
HOME=/home/bill
LOGNAME=bill
rp=(null)
[bill@billstone Unix_study]$
```

在进程中执行新程序的三种方法

进程和人类一样, 都有创建, 发展, 休眠和死亡等各种生命形态. 其中, 函数 fork 创建新进程, 函数 exec 执行新程序, 函数 sleep 休眠进程, 函数 wait 同步进程和函数 exit 结束进程.

(1) fork-exec

调用 fork 创建的子进程, 将共享父进程的代码空间, 复制父进程数据空间, 如堆栈等. 调用 exec 族函数将使用新程序的代码覆盖进程中原来的程序代码, 并使进程使用函数提供的命令行参数和环境变量去执行新的程序.

exec 函数族有六个函数如下:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., (char *)0);
int execle(const char *path, const char *arg0, ..., (char *)0, char *const envp[]);
int execlp(const char *file, const char *arg0, ..., (char *)0);
int execv(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
int execvp(const char *file, const char *argv[]);
extern char **environ;
```

如何用 fork-exec 方式执行程序'uname -a?

```
[bill@billstone Unix_study]$ cat exec1.c
#include <sys/types.h>
#include <unistd.h>
```

```
#include <stdio.h>


int main()
{
        pid_t pid;


        if((pid = fork()) == 0){
                fprintf(stderr, "---- begin ----\n");
                // sleep(3);                    // 睡眠 3 秒会导致子进程成为僵死进程
                execl("/bin/uname", "uname", "-a", 0);
                fprintf(stderr, "----   end   ----\n");
        }
        else if(pid > 0)
                fprintf(stderr, "fork child pid = [%d]\n", pid);
        else
                fprintf(stderr, "Fork failed.\n");


        return 0;
}
[bill@billstone Unix_study]$ make exec1
cc      exec1.c    -o exec1
[bill@billstone Unix_study]$ ./exec1
---- begin ----
Linux billstone 2.4.20-8 #1 Thu Mar 13 17:18:24 EST 2003 i686 athlon i386 GNU/Linux
fork child pid = [13276]
[bill@billstone Unix_study]$ ./exec1
---- begin ----
fork child pid = [13278]
[bill@billstone Unix_study]$ Linux billstone 2.4.20-8 #1 Thu Mar 13 17:18:24 EST 2003 i686 athlon i386 GNU/Linux
```

(2) vfork-exec

vfork 比起 fork 函数更快, 二者的区别如下:

a) vfork 创建的子进程并不复制父进程的数据, 在随后的 exec 调用中系统会复制新程序的数据到内存, 继而避免了一次数据复制过程

b) 父进程以 vfork 方式创建子进程后将被阻塞, 知道子进程退出或执行 exec 调用后才能继续运行.

当子进程只用来执行新程序时, vfork-exec 模型比 fork-exec 模型具有更高的效率, 这种方法也是 Shell 创建新进程的方式.

```
[bill@billstone Unix_study]$ cat exec2.c
#include <sys/types.h>
#include <unistd.h>
```

```
#include <stdio.h>


int main()
{
        pid_t pid;


        if((pid = vfork()) == 0){
                fprintf(stderr, "---- begin ----\n");
                sleep(3);
                execl("/bin/uname", "uname", "-a", 0);
                fprintf(stderr, "----   end   ----\n");
        }
        else if(pid > 0)
                fprintf(stderr, "fork child pid = [%d]\n", pid);
        else
                fprintf(stderr, "Fork failed.\n");


        return 0;
}
[bill@billstone Unix_study]$ make exec2
make: `exec2' is up to date.
[bill@billstone Unix_study]$ ./exec2
---- begin ----
fork child pid = [13293]
[bill@billstone Unix_study]$ Linux billstone 2.4.20-8 #1 Thu Mar 13 17:18:24 EST 2003 i686 athlon i386 GNU/Linux
```

(3) system

在 UNIX 中, 我们也可以使用 system 函数完成新程序的执行.

```
#include <stdlib.h>
char *getenv(char *name);
int putenv(const char *string);
```

函数 system 会阻塞调用它的进程, 并执行字符串 string 中的 shell 命令.

```
[bill@billstone Unix_study]$ cat exec3.c
#include <unistd.h>
#include <stdio.h>


int main()
{
```

```
        char cmd[] = {"/bin/uname -a"};


        system(cmd);


        return 0;
}
[bill@billstone Unix_study]$ make exec3
cc      exec3.c    -o exec3
[bill@billstone Unix_study]$ ./exec3
Linux billstone 2.4.20-8 #1 Thu Mar 13 17:18:24 EST 2003 i686 athlon i386 GNU/Linux
[bill@billstone Unix_study]$
```

僵死进程

僵死进程是已经终止, 但没有从进程表中清除的进程, 下面是一个僵死进程的实例

```
[bill@billstone Unix_study]$ cat szomb1.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>


int main()
{
        pid_t pid;


        if((pid = fork()) == 0){
                printf("child[%d]\n", getpid());
                exit(0);
        }
        // wait();
        printf("parent[%d]\n", getpid());
        sleep(10);


        return 0;
}
[bill@billstone Unix_study]$
```

后台运行程序 szobm1, 并在子进程结束后, 父进程没有结束前, 运行命令查询进程情况:

```
[bill@billstone Unix_study]$ make szomb1
```

```
cc      szomb1.c    -o szomb1

[bill@billstone Unix_study]$ ./szomb1 &

[2] 13707

child[13708]

[bill@billstone Unix_study]$ parent[13707]

ps -ef | grep 13707

bill      13707   1441   0 04:17 pts/0      00:00:00 ./szomb1

bill      13708 13707   0 04:17 pts/0      00:00:00 [szomb1 <defunct>]       // 僵死进程

bill      13710   1441   0 04:17 pts/0      00:00:00 grep 13707

[bill@billstone Unix_study]$
```

其中, 'defunct'代表僵死进程. 对于僵死进程, 不能奢望通过 kill 命令杀死之, 因为它已经'死'了, 不再接收任何系统信号.

当子进程终止时, 它释放资源, 并且发送 SIGCHLD 信号通知父进程. 父进程接收 SIGCHLD 信号,调用 wait 返回子进程的状态, 并且释放系统进程表资源. 故如果子进程先于父进程终止, 而父进程没有调用 wait 接收子进程信息,则子进程将转化为僵死进程, 直到其父进程结束.

一旦知道了僵死进程的成因, 我们可以采用如下方法预防僵死进程:

(1) wait 法

父进程主动调用 wait 接收子进程的死亡报告, 释放子进程占用的系统进程表资源.

(2) 托管法

如果父进程先于子进程而死亡, 则它的所有子进程转由进程 init 领养, 即它所有子进程的父进程 ID 号变为 1. 当子进程结束时 init 为其释放进程表资源.

(3) 忽略 SIGC(H)LD 信号

当父进程忽略 SIGC(H)LD 信号后, 即使不执行 wait, 子进程结束时也不会产生僵死进程.

(4) 捕获 SIGC(H)LD 信号

当父进程捕获 SIGC(H)LD 信号, 并在捕获函数代码中等待(wait)子进程

守护进程

所谓守护进程是一个在后台长期运行的进程, 它们独立于控制终端, 周期性地执行某项任务, 或者阻塞直到事件发生, 默默地守护着计算机系统的正常运行. 在 UNIX 应用中, 大部分 socket 通信服务程序都是以守护进程方式执行.

完成一个守护进程的编写至少包括以下几项:

(1) 后台执行

后台运行的最大特点是不再接收终端输入, 托管法可以实现这一点

```
pid_t pid;

pid = fork();

if(pid > 0) exit(0);          // 父进程退出

/* 子进程继续运行   */

父进程结束, shell 重新接管终端控制权, 子进程移交 init 托管
```

(2) 独立于控制终端

在后台进程的基础上, 脱离原来 shell 的进程组和 session 组, 自立门户为新进程组的会话组长进程, 与原终端脱离关系

```
#include <unistd.h>

pid_t setsid();
```

函数 setsid 创建一个新的 session 和进程组.

(3) 清除文件创建掩码

进程清除文件创建掩码,代码如下:

```
umask(0);
```

(4) 处理信号

为了预防父进程不等待子进程结束而导致子进程僵死, 必须忽略或者处理 SIGCHLD 信号, 其中忽略该信号的方法为:

```
signal(SIGCHLD, SIG_IGN);
```

守护进程独立于控制终端, 它们一般以文件日志的方式进行信息输出.

下面是一个简单的守护进程实例 InitServer

```
[bill@billstone Unix_study]$ cat initServer.c
#include <assert.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>

void ClearChild(int nSignal){
        pid_t pid;
        int nState;
                //  WNOHANG 非阻塞调用 waitpid, 防止子进程成为僵死进程
        while((pid = waitpid(-1, &nState, WNOHANG)) > 0);
        signal(SIGCLD, ClearChild);     // 重新绑定 SIGCLD 信号
}

int InitServer(){
        pid_t pid;

        assert((pid = fork()) >= 0);            // 创建子进程
        if(pid != 0){                   // 父进程退出, 子进程被 init 托管
                sleep(1);
                exit(0);
        }
        assert(setsid() >= 0);              // 子进程脱离终端
        umask(0);                           // 清除文件创建掩码
        signal(SIGINT, SIG_IGN);        // 忽略 SIGINT 信号
        signal(SIGCLD, ClearChild);     // 处理 SIGCLD 信号,预防子进程僵死

        return 0;
```

```
}


int main()
{
        InitServer();
        sleep(100);


        return 0;
}
[bill@billstone Unix_study]$ make initServer
cc      initServer.c    -o initServer
[bill@billstone Unix_study]$ ./initServer
[bill@billstone Unix_study]$ ps -ef | grep initServer
bill      13721      1   0 04:40 ?           00:00:00 ./initServer        // '?'代表 initServer 独立于终端
bill      13725   1441   0 04:41 pts/0       00:00:00 grep initServer
[bill@billstone Unix_study]$
```

程序在接收到 SIGCLD 信号后立即执行函数 ClearChild, 并调用非阻塞的 waitpid 函数结束子进程结束信息, 如果结束到子进程结束信息则释放该子进程占用的进程表资源, 否则函数立刻返回. 这样既保证了不增加守护进程负担, 又成功地预防了僵死进程的产生.

第十章 时钟与信号

获取时钟

UNIX 的时间系统存在一个基点, 就是格林威治时间 1970 年 1 月 1 日凌晨 0 点 0 分 0 秒, 也是传说中 UNIX 的生日.

UNIX 中存在三种格式的时间:

(1) 系统时间. UNIX 从出生到现在的秒数, 表现为一个 time_t 类型的变量

(2) 高分辨率时间. 精确到微秒的时间, 表现为一个 timeval 结构的变量

(3) 日历时间. 以'年, 月, 日, 时, 分, 秒'结构表示的时间, 表现为 tm 结构.

系统时间. 它是 UNIX 中最基本的时间形式. 用于系统时间的函数如下:

```
#include <time.h>
time_t time(time_t *tloc);
double difftime(time_t time2, time_t time1);
```

函数 difftime 获取两次 time 调用返回的系统时间差.

秒数往往很难读懂, UNIX 中更改系统时间为日历时间的函数如下:

```
#include <time.h>
struct tm *localtime(const time_t *clock);
time_t mktime(struct tm*timeptr);
```

函数 localtime 转换系统时间, clock 为当地时间, 并以 tm 结构返回.

函数 mktime 实现函数 localtime 的反功能.

下面给出一个打印本地时间的例子

```
[bill@billstone Unix_study]$ cat time1.c
#include <time.h>
#include <stdio.h>

int main()
{
        struct tm when;
        time_t now;

        time(&now);
        when = *localtime(&now);
        printf("now=[%d] [%04d %02d %02d %02d:%02d:%02d]\n", now, \
                when.tm_year+1900, when.tm_mon+1, when.tm_mday, \
                when.tm_hour, when.tm_min, when.tm_sec);

        return 0;
}
[bill@billstone Unix_study]$ make time1
cc      time1.c    -o time1
[bill@billstone Unix_study]$ ./time1
now=[1239927129] [2009 04 17 08:12:09]
[bill@billstone Unix_study]$
```

信号的概念

信号是传送给进程的事件通知, 它可以完成进程间异步事件的通信.

导致信号产生的原因很多, 但总体说来有三种可能:

(1) 程序错误. 当硬件出现异常, 除数为 0 或者软件非法访问等情况时发生.

(2) 外部事件. 当定时器到达, 用户按健中断或者进程调用 abort 等信号发送函数时方生.

(3) 显式请求. 当进程调用 kill, raise 等信号发送函数或者用户执行 shell 命令 kill 传递信号时发生.

同样的, 当进程收到信号时有三种处理方式:

(1) 系统默认. 系统针对不同的信号有不同的默认处理方式.

(2) 忽略信号. 信号收到后, 立即丢弃. 注意信号 SIGSTOP 和 SIGKILL 不能忽略.

(3) 捕获信号. 进程接收信号, 并调用自定义的代码响应之.

信号操作

函数 signal 设置对信号的操作动作,原型如下:

```
#include <signal.h>
void (*signal (int sig, void (*f) (int)) (int);
```

这是个复杂的函数原型, 不果可以分开看:

```
typedef void (*func)(int);
```

```
func signal(int sig, func f);
```

其中, func 参数有三种选择:SIG_DFL(恢复信号默认处理机制), SIG_IGN(忽略信号处理)和函数地址(调用信号捕获函数执行处理).

首先看一个忽略终止信号 SIGINT 的例子.

```
[bill@billstone Unix_study]$ cat sig1.c
#include <signal.h>
#include <stdio.h>

int main()
{
        signal(SIGINT, SIG_IGN);
        sleep(10);                      // 睡眠 10 秒

        return 0;
}
[bill@billstone Unix_study]$ make sig1
cc      sig1.c    -o sig1
[bill@billstone Unix_study]$ ./sig1
[bill@billstone Unix_study]$
```

在程序运行的 10 秒内,即使你键入 Ctrl+C 中断命令, 进程也不退出.

再看一个捕获自定义信号的例子.

```
[bill@billstone Unix_study]$ cat sig2.c
#include <signal.h>
#include <stdio.h>

int usr1 = 0, usr2 = 0;
void func(int);

int main()
{
        signal(SIGUSR1, func);
        signal(SIGUSR2, func);
        for(;;)
                sleep(1);               //  死循环, 方便运行观察

        return 0;
}
```

```
void func(int sig){
        if(sig == SIGUSR1)
                usr1++;
        if(sig == SIGUSR2)
                usr2++;
        fprintf(stderr, "SIGUSR1[%d], SIGUSR2[%d]\n", usr1, usr2);
        signal(SIGUSR1, func);
        signal(SIGUSR2, func);
}
```

在后台运行, 结果如下:

```
[bill@billstone Unix_study]$ make sig2
cc      sig2.c    -o sig2
[bill@billstone Unix_study]$ ./sig2&          // 后台运行
[2] 13822
[bill@billstone Unix_study]$ kill -USR1 13822      // 发送信号 SIGUSR1
SIGUSR1[1], SIGUSR2[0]
[bill@billstone Unix_study]$ kill -USR2 13822       // 发送信号 SIGUSR2
SIGUSR1[1], SIGUSR2[1]
[bill@billstone Unix_study]$ kill -USR2 13822       // 发送信号 SIGUSR2
SIGUSR1[1], SIGUSR2[2]
[bill@billstone Unix_study]$ kill -9 13822           //   发送信号 SIGSTOP, 杀死进程
[bill@billstone Unix_study]$
[2]+   已杀死                 ./sig2
[bill@billstone Unix_study]$
```

UNIX 应用程序可以向进程显式发送任意信号, 原型如下:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

看一个发送和捕获 SIGTERM 终止信号的例子

```
[bill@billstone Unix_study]$ cat sig3.c
#include <signal.h>
#include <stdio.h>
#include <assert.h>
#include <unistd.h>
#include <sys/types.h>
```

```
void childfunc(int sig){
        fprintf(stderr, "Get Sig\n");
}

int main()
{
        pid_t pid;
        int status;

        assert((pid = fork()) >= 0);
        if(pid == 0){
                signal(SIGTERM, childfunc);
                sleep(30);
                exit(0);
        }
        fprintf(stderr, "Parent [%d] Fork child pid=[%d]\n", getpid(), pid);
        sleep(1);
        kill(pid, SIGTERM);
        wait(&status);
        fprintf(stderr, "Kill child pid=[%d], exit status[%d]\n", pid, status>>8);

        return 0;
}
[bill@billstone Unix_study]$ make sig3
cc      sig3.c    -o sig3
[bill@billstone Unix_study]$ ./sig3
Parent [13898] Fork child pid=[13899]
Get Sig
Kill child pid=[13899], exit status[0]
[bill@billstone Unix_study]$
```

定时器

UNIX 下定时器可以分为普通的定时器和精确的定时器.

普通定时器通过 alarm 函数实现, 它的精度是秒, 而且每调用一次 alarm 函数只能产生一次定时操作, 如果需要反复定时, 就要多次调用 alarm. 调用 fork 后, 子进程中的定时器将被取消, 但调用 exec 后, 定时器仍然有效.

在 UNIX 中使用普通定时器需要三个步骤:

(1) 调用 signal 函数设置捕获定时信号

(2) 调用函数 alarm 定时.

(3) 编写响应定时信号函数.

下面是一个定时器的例子, 每隔 1 秒向进程发送定时信号,用户可键入 Ctrl+C 或 Delete 结束程序.

```
[bill@billstone Unix_study]$ cat time2.c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int n = 0;

void timefunc(int sig){
        fprintf(stderr, "Alarm %d\n", n++);
        signal(SIGALRM, timefunc);
        alarm(1);
}

int main()
{
        int status;

        signal(SIGALRM, timefunc);
        alarm(1);
        while(1);

        return 0;
}
[bill@billstone Unix_study]$ make time2
cc      time2.c    -o time2
[bill@billstone Unix_study]$ ./time2
Alarm 0
Alarm 1
Alarm 2
                        // 按 Ctrl+C 结束
[bill@billstone Unix_study]$
```

函数 alarm 设置的定时器只能精确到秒, 而下面函数理论上可以精确到毫秒:

```
#include <sys/select.h>
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval value, struct itimerval *ovalue);
```

函数 setitimer 可以提供三种定时器, 它们相互独立, 任意一个定时完成都将发送定时信号到进程, 并且重新计时. 参数 which 确定了定时器的类型:

(1) ITIMER_REAL. 定时真实时间, 与 alarm 类型相同. 对应信号为 SIGALRM.

(2) ITIMER_VIRT. 定时进程在用户态下的实际执行时间. 对应信号为 SIGVTALRM.

(3) ITIMER_PROF. 定时进程在用户态和核心态下的实际执行时间. 对应信号为 SIGPROF.

在一个 UNIX 进程中, 不能同时使用 alarm 和 ITIMER_REAL 类定时器.

结构 itimerval 描述了定时器的组成:

```
struct itimerval{

        struct timeval it_interval;

        struct timeval it_value;

}
```

结构成员 it_value 指定首次定时的时间, 结构成员 it_interval 指定下次定时的时间. 定时器工作时, 先将 it_value 的时间值减到 0, 发送一个信号, 再将 it_vale 赋值为 it_interval 的值, 重新开始定时, 如此反复. 如果 it_value 值被设置为 0, 则定时器停止定时.

结构 timeval 秒数了一个精确到微秒的时间:

```
struct timeval{

        long tv_sec;

        long tv_usec;

}
```

下面设计了一个精确定时器的例子, 进程每 1.5 秒发送定时信号 SIGPROF, 用户可键入 Ctrl+C 或 Delete 结束程序.

```
[bill@billstone Unix_study]$ cat time3.c

#include <sys/select.h>

#include <sys/time.h>

#include <stdio.h>

#include <unistd.h>

#include <signal.h>


int n = 0;


void timefunc(int sig){

        fprintf(stderr, "ITIMER_PROF[%d]\n", n++);

        signal(SIGPROF, timefunc);

}


int main()

{

        struct itimerval value;


        value.it_value.tv_sec = 1;

        value.it_value.tv_usec = 500000;
```

```
        value.it_interval.tv_sec = 1;
        value.it_interval.tv_usec = 500000;
        signal(SIGPROF, timefunc);
        setitimer(ITIMER_PROF, &value, NULL);
        while(1);

        return 0;
}
[bill@billstone Unix_study]$ make time3
cc      time3.c    -o time3
[bill@billstone Unix_study]$ ./time3
ITIMER_PROF[0]
ITIMER_PROF[1]
ITIMER_PROF[2]

[bill@billstone Unix_study]$
```

全局跳转

UNIX 下的 C 语言中,有一对特殊的调用:跳转函数, 原型如下:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjump(jmp_buf env, int val);
```

函数 setjmp 存储当前的堆栈环境(包括程序的当前执行位置)到参数 env 中,当函数正常调用成功时返回 0. 函数 longjmp 恢复保存在 env 中堆栈信息, 并使程序转移到 env 中保存的位置处重新执行. 这两个函数联合使用, 可以实现程序的重复执行.

函数 longjmp 调用成功后, 程序转移到函数 setjmp 处执行, 函数 setjmp 返回 val. 如果参数 val 的取值为 0, 为了与上次正常调用 setjmp 相区别,函数 setjmp 将自动返回 1.

下面是一个使用了跳转语句的例子, 它跳转两次后退出.

```
[bill@billstone Unix_study]$ cat jmp1.c
#include <setjmp.h>

int j = 0;
jmp_buf env;

int main()
{
        auto int i, k = 0;
```

```
        i = setjmp(env);
        printf("setjmp = [%d], j = [%d], k = [%d]\n", i, j++, k++);
        if(j > 2)
                exit(0);
        sleep(1);
        longjmp(env, 1);

        return 0;
}
[bill@billstone Unix_study]$ make jmp1
cc      jmp1.c    -o jmp1
[bill@billstone Unix_study]$ ./jmp1
setjmp = [0], j = [0], k = [0]
setjmp = [1], j = [1], k = [1]
setjmp = [1], j = [2], k = [2]
[bill@billstone Unix_study]$
```

其中, j 记录了程序的执行次数. 按理说, k 的值应该保持不变, 因为当返回到 setjmp 重新执行时, 保存的堆栈中 k 应该保持 0 不变, 但实际上却变化了. 请高手指点, 是不是 setjmp 本身实现的问题(我用的环境是 Red Hat 9)?

单线程 I/O 超时处理

UNIX 下的 I/O 超时处理是一个很常见的问题, 它的通常做法是接收输入(或发送输出)后立刻返回, 如果无输入(或输出)则 n 秒后定时返回.

一般情况下, 处理 UNIX 中 I/O 超时的方式有终端方式, 信号跳转方式和多路复用方式等三种. 本节设计一个定时 I/O 的例子, 它从文件描述符 0 中读取一个字符, 当有输入时继续, 或者 3 秒钟后超时退出,并打印超时信息.

(1) 终端 I/O 超时方式

利用 ioctl 函数, 设置文件描述符对应的标准输入文件属性为”接收输入后立刻返回, 如无输入则 3 秒后定时返回.

```
[bill@billstone Unix_study]$ cat timeout1.c
#include <unistd.h>
#include <termio.h>
#include <fcntl.h>

int main()
{
        struct termio old, new;
        char c = 0;

        ioctl(0, TCGETA, &old);
```

```
        new = old;
        new.c_lflag &= ~ICANON;
        new.c_cc[VMIN] = 0;
        new.c_cc[VTIME] = 30;           // 设置文件的超时时间为 3 秒
        ioctl(0, TCSETA, &new);
        if((read(0, &c, 1)) != 1)
                printf("timeout\n");
        else
                printf("\n%d\n", c);
        ioctl(0, TCSETA, &old);

        return 0;
}
[bill@billstone Unix_study]$ make timeout1
cc      timeout1.c    -o timeout1
[bill@billstone Unix_study]$ ./timeout1
x
120
[bill@billstone Unix_study]$ ./timeout1
timeout
[bill@billstone Unix_study]$
```

(2) 信号与跳转 I/O 超时方式

在 read 函数前调用 setjmp 保存堆栈数据并使用 alarm 设定 3 秒定时.

```
[bill@billstone Unix_study]$ cat timeout2.c
#include <setjmp.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int timeout = 0;
jmp_buf env;

void timefunc(int sig){
        timeout = 1;
        longjmp(env, 1);
}

int main()
```

```
{
        char c;


        signal(SIGALRM, timefunc);
        setjmp(env);
        if(timeout == 0){
                alarm(3);
                read(0, &c, 1);
                alarm(0);
                printf("%d\n", c);
        }
        else
                printf("timeout\n");


        return 0;
}
[bill@billstone Unix_study]$ make timeout2
cc      timeout2.c    -o timeout2
[bill@billstone Unix_study]$ ./timeout2
v                       // 需要按 Enter 健激活输入
118
[bill@billstone Unix_study]$ ./timeout2
timeout
[bill@billstone Unix_study]$
```

(3) 多路复用 I/O 超时方式

一个进程可能同时打开多个文件, UNIX 中函数 select 可以同时监控多个文件描述符的输入输出, 进程将一直阻塞, 直到超时或产生 I/O 为止, 此时函数返回, 通知进程读取或发送数据.

函数 select 的原型如下:

```
#include <sys/types.h>
#include <sys/times.h>
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
FD_CLR(int fd, fd_set *fdset);      // 从 fdset 中删去文件描述符 fd
FD_ISSET(int fd, fd_set *fdset);    // 查询文件描述符是否在 fdset 中
FD_SET(int fd, fd_set *fdset);      // 在 fdset 中插入文件描述符 fd
FD_ZERO(fd_set *fdset);           // 清空 fdset
```

参数 nfds 是 select 监控的文件描述符的时间, 一般为监控的最大描述符编号加 1.

类型 fd_set 是文件描述符集合, 其元素为监控的文件描述符.

参数 timeout 是描述精确时间的 timeval 结构,它确定了函数的超时时间,有三种取值情况:

a) NULL. 函数永远等待, 直到文件描述符就绪.

b) 0. 函数不等待, 检查文件描述符状态后立即返回.

c) 其他值. 函数等待文件描述符就绪, 或者定时完成时返回.

函数 select 将返回文件描述符集合中已准备好的文件总个数. 函数 select 返回就绪文件描述符数量后, 必须执行 read 等函数, 否则函数继续返回就绪文件数.

```
[bill@billstone Unix_study]$ cat timeout3.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/select.h>

int main()
{
        struct timeval timeout;
        fd_set readfds;
        int i;
        char c;

        timeout.tv_sec = 3;
        timeout.tv_usec = 0;
        FD_ZERO(&readfds);
        FD_SET(0, &readfds);
        i = select (1, &readfds, NULL, NULL, &timeout);
        if(i > 0){
                read(0, &c, 1);
                printf("%d\n", c);
        }
        else if(i == 0)
                printf("timeout\n");
        else
                printf("error\n");

        return 0;
}
[bill@billstone Unix_study]$ make timeout3
cc      timeout3.c    -o timeout3
[bill@billstone Unix_study]$ ./timeout3
x
120
```

```
[bill@billstone Unix_study]$

[bill@billstone Unix_study]$ ./timeout3

timeout

[bill@billstone Unix_study]$
```

第 4 篇 进程通信篇

第十一章 管道

管道是 UNIX 中最古老的进程间通信工具, 它提供了进程之间的一种单向通信的方法.

管道分为无名管道和有名管道(FIFO)两种, 前者在父子进程中流行, 后者由于可以独立成磁盘文件而存在, 因而能够被无血缘关系的进程所共享.

无名管道

无名管道占用两个文件描述符, 不能被非血缘关系的进程所共享, 一般应用在父子进程中.

在 UNIX 中, 采用函数 pipe 创建无名管道, 原型如下:

```
#include <unistd.h>

int pipe(int fildes[2]);
```

我们一般约定 fildes[1]描述管道的输入端, 进程向此文件描述符中写入数据, fildes[0]描述管道的输出端, 进程从此文件描述符中读取数据.

无名管道常用于父子进程中, 可简单分为单向管道流模型和双向管道流模型. 其中, 单向管道流根据流向分为从父进程流向子进程的管道和从子进程流向父进程的管道.

下面设计一个实例, 数据从父进程流向子进程:父进程向管道写入一行字符, 子进程读取数据并打印到屏幕上.

```
[bill@billstone Unix_study]$ cat pipe1.c

#include <unistd.h>

#include <stdio.h>

#include <sys/types.h>

#include <assert.h>


int main()

{

        int fildes[2];

        pid_t pid;

        int i,j;

        char buf[256];
```

```
        assert(pipe(fildes) == 0);                  // 创建管道
        assert((pid = fork()) >= 0);                // 创建子进程
        if(pid == 0){                           // 子进程
                close(fildes[1]);                   // 子进程关闭管道输出
                memset(buf, 0, sizeof(buf));
                j = read(fildes[0], buf, sizeof(buf));
                fprintf(stderr, "[child] buf=[%s] len[%d]\n", buf, j);
                return;
        }
        close(fildes[0]);                           // 父进程关闭管道输入
        write(fildes[1], "hello!", strlen("hello!"));
        write(fildes[1], "world!", strlen("world!"));

        return 0;
}
[bill@billstone Unix_study]$ make pipe1
cc      pipe1.c    -o pipe1
[bill@billstone Unix_study]$ ./pipe1
[child] buf=[hello!world!] len[12]                  // 子进程一次就可以读出两次父进程写入的数据
[bill@billstone Unix_study]$
```

从上面可以看出, 在父进程中关闭 fildes[0], 向 fildes[1]写入数据; 在子进程中关闭 filedes[1], 从 fildes[0]中读取数据可实现从父进程流向子进程的管道.

在进程的通信中, 我们无法判断每次通信中报文的字节数, 即无法对数据流进行 自行拆分, 侧耳发生了上例中子进程一次性读取父进程两次通信的报文情况.

管道是进程之间的一种单向交流方法, 要实现进程间的双向交流, 就必须通过两个管道来完成. 双向管道流的创立过程如下:

(1) 创建管道, 返回两个无名管道文件描述符 fildes1 和 fildes2:
(2) 创建子进程, 子进程中继承管道 fildes1 和 fildes2.
(3) 父进程关闭只读文件描述符 fildes1[0], 只写描述符 fildes2[1]
(4) 子进程关闭只写文件描述符 fildes1[1], 只读描述符 fildes2[0]

创建的结果如下:

父进程 --写--> fildes1[1] --管道--> fildes1[0] --读--> 子进程
父进程 <--读-- fildes2[0] <--管道-- fildes2[1] <--写-- 子进程

这里实现一个父子进程间双向通信的实例: 父进程先向子进程发送两次数据, 再接收子进程传送刚来的两次数据. 为了正确拆分时间留从父进程流向子进程的管道采用'固定长度'方法传送数据; 从子进程流向父进程的管道采用'显式长度'方法传回数据.

(1) 固定长度方式

```
char bufG[255];
void WriteG(int fd, char *str, int len){
        memset(bufG, 0, sizeof(bufG));
        sprintf(bufG, "%s", str);
```

```
        write(fd, bufG, len);
}

char *ReadG(int fd, int len){
        memset(bufG, 0, sizeof(bufG));
        read(fd, bufG, len);

        return(bufG);
}
```

在此设计中, 父子程序需要约定好每次发送数据的长度; 且长度不能超过 255 个字符.

(2) 显式长度方式

```
char bufC[255];
void WriteC(int fd, char str[]){
        sprintf(bufC, "%04d%s", strlen(str), str);
        write(fd, bufC, strlen(bufC));
}

char *ReadC(int fd){
        int i, j;

        memset(bufC, 0, sizeof(bufC));
        j = read(fd, bufC, 4);
        i = atoi(bufC);
        j = read(fd, bufC, i);

        return(bufC);
}
```

父子进程约定在发送消息前先指明消息的长度.

(3) 主程序

```
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>

int main()
{
        int fildes1[2], fildes2[2];
        pid_t pid;
```

```
        char buf[255];


        assert(pipe(fildes1) == 0);
        assert(pipe(fildes2) == 0);
        assert((pid = fork()) >= 0);

        if(pid == 0){
                close(fildes1[1]);
                close(fildes2[0]);
                strcpy(buf, ReadG(fildes1[0], 10));
                fprintf(stderr, "[child] buf = [%s]\n", buf);
                WriteC(fildes2[1], buf);
                strcpy(buf, ReadG(fildes1[0], 10));
                fprintf(stderr, "[child] buf = [%s]\n", buf);
                WriteC(fildes2[1], buf);
                return(0);
        }

        close(fildes1[0]);
        close(fildes2[1]);
        WriteG(fildes1[1], "hello!", 10);
        WriteG(fildes1[1], "world!", 10);
        fprintf(stderr, "[father] buf = [%s] \n", ReadC(fildes2[0]));
        fprintf(stderr, "[father] buf = [%s] \n", ReadC(fildes2[0]));

        return 0;
}
```

执行结果如下:

```
[bill@billstone Unix_study]$ make pipe2
cc      pipe2.c    -o pipe2
[bill@billstone Unix_study]$ ./pipe2
[child] buf = [hello!]
[child] buf = [world!]
[father] buf = [hello!]
[father] buf = [world!]
[bill@billstone Unix_study]$
```

popen 模型

从前面的程序可以看出, 创建连接标准 I/O 的管道需要多个步骤, 这需要使用大量的代码, UNIX 为了简化这个操作, 它提供了一组函数实现之. 原型如下:

```
#include <stdio.h>

FILE *popen(const char *command, char *type);

int pclose(FILE *stream);
```

函数 popen 调用成功时返回一个标准的 I/O 的 FILE 文件流, 其读写属性由参数 type 决定.

这里看一个模拟 shell 命令'ps -ef | grep init'的实例.

```
[bill@billstone Unix_study]$ cat pipe3.c

#include <stdio.h>

#include <assert.h>


int main()

{

        FILE *out, *in;

        char buf[255];


        assert((out = popen("grep init", "w")) != NULL);          //  创建写管道流

        assert((in = popen("ps -ef", "r")) != NULL);              // 创建读管道流


        while(fgets(buf, sizeof(buf), in))          // 读取 ps -ef 的结果

                fputs(buf, out);                    // 转发到 grep init


        pclose(out);

        pclose(in);


        return 0;

}

[bill@billstone Unix_study]$ make pipe3

cc      pipe3.c    -o pipe3

[bill@billstone Unix_study]$ ./pipe3

root         1     0   0 Apr15 ?           00:00:04 init

bill      1392  1353   0 Apr15 ?           00:00:00 /usr/bin/ssh-agent /etc/X11/xinit/Xclients

bill     14204 14203   0 21:33 pts/0       00:00:00 grep init

[bill@billstone Unix_study]$ ps -ef | grep init

root         1     0   0 Apr15 ?           00:00:04 init

bill      1392  1353   0 Apr15 ?           00:00:00 /usr/bin/ssh-agent /etc/X11/xinit/Xclients
```

```
bill       14207   1441   0 21:35 pts/0       00:00:00 grep init
[bill@billstone Unix_study]$
```

读者可以从上面自行比较同 Shell 命令'ps -ef | grep init'的执行结果.

有名管道 FIFO

FIFO 可以在整个系统中使用.

在 Shell 中可以使用 mknod 或者 mkfifo 命令创建管道; 而在 C 程序中, 可以使用 mkfifo 函数创建有名管道.

要使用有名管道, 需要下面几个步骤:

(1) 创建管道文件

(2) 在某个进程中以只写方式打开管道文件, 并写管道

(3) 在某个进程中以只读方式打开管道文件, 并读管道

(4) 关闭管道文件.

低级文件编程库和标准文件编程库都可以操作管道. 管道在执行读写操作之前, 两端必须同时打开, 否则执行打开管道某端操作的进程将一直阻塞到某个进程以相反方向打开管道为止.

下面是一个简单的实例.

首先是写进程: 创建 FIFO 文件, 再打开写端口, 然后读取标准输入并将输入信息发送到管道中, 当键盘输入'exit'或'quit'时程序退出.

```
[bill@billstone Unix_study]$ cat fifo1.c
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;

int main()
{
        FILE *fp;
        char buf[255];

        assert((mkfifo("myfifo", S_IFIFO|0666) > 0) || (errno == EEXIST));

        while(1){
                assert((fp = fopen("myfifo", "w")) != NULL);
                printf("please input: ");
                fgets(buf, sizeof(buf), stdin);
                fputs(buf, fp);
                fclose(fp);
                if(strncmp(buf, "quit", 4) == 0 || strncmp(buf, "exit", 4) == 0)
```

```
                break;
        }

        return 0;
}
[bill@billstone Unix_study]$ make fifo1
cc      fifo1.c    -o fifo1
[bill@billstone Unix_study]$
```

然后是读进程: 打开管道的读端口, 从管道中读取信息(以行为单位), 并将此信息打印到屏幕上. 当读取到'exit'或者'quit'时程序退出.

```
[bill@billstone Unix_study]$ cat fifo2.c
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
        FILE *fp;
        char buf[255];

        while(1){
                assert((fp = fopen("myfifo", "r")) != NULL);
                fgets(buf, strlen(buf), fp);
                printf("gets: [%s]", buf);
                fclose(fp);
                if(strncmp(buf, "quit", 4) == 0 || strncmp(buf, "exit", 4) == 0)
                        break;
        }

        return 0;
}
[bill@billstone Unix_study]$ make fifo2
cc      fifo2.c    -o fifo2
[bill@billstone Unix_study]$
```

在一个终端上执行 fifo1, 而在另一个终端上执行 fifo2.

我们先输入'hello', 'world', 然后再输入'exit'退出:

```
[bill@billstone Unix_study]$ ./fifo1
```

```
please input: hello

please input: world

please input: exit

[bill@billstone Unix_study]$
```

我们可以看到读出结果如下:

```
[bill@billstone Unix_study]$ ./fifo2

gets: [hello

]gets: [world]gets: [exit][bill@billstone Unix_study]$
```

看到上面的输出结果, 您可能认为是我写错了. 其实不是的, 读出结果正是如此, 其实按照我们的本意, 正确的输出结果应该是这样的:

```
[bill@billstone Unix_study]$ ./fifo2

gets: [hello

]gets: [world

]gets: [exit

][bill@billstone Unix_study]$
```

那么到底是什么原因导致了那样的输出呢? 我现在也不知道. 如果有知晓个中原因的朋友说一下.

第十二章 消息队列

消息队列是 UNIX 内核中的一个先进先出的链表结构. 相对于管道, 消息队列有明显的优势, 原因在于:

(1) 消息队列是一种先进先出的队列型数据结构, 可以保证先送的货物先到达, 后送的货物后到达, 避免了插队现象.

(2) 信息队列将输出的信息进行了打包处理, 这样就可以保证以每个消息为单位进行接收了.

(3) 消息队列还可以对信息进行分类处理, 标记各种类别的信息, 这样就可以根据信息类别分别出列.

IPC 就是进程间通信, 侠义上讲, IPC 指消息队列, 信号量和共享内存三种对象. 通过 shell 命令 ipcs 可以查询当前系统的 IPC 对象信息:

```
[bill@billstone Unix_study]$ ipcs


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

key          shmid      owner      perms      bytes      nattch     status

0x00000000 196609     bill       777        393216     2          dest

0x00000000 491522     root       644        106496     2          dest

0x00000000 524291     root       644        110592     2          dest

0x00000000 557060     root       644        110592     2          dest

0x00000000 589829     root       644        110592     2          dest


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

key          semid      owner      perms      nsems


```

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

[bill@billstone Unix_study]$
```

消息队列简介

UNIX 内核使用结构 msqid_ds 来管理消息队列:

```
struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg *msg_first;          /* first message on queue, unused  */
        struct msg *msg_last;           /* last message in queue, unused */
        __kernel_time_t msg_stime;      /* last msgsnd time */
        __kernel_time_t msg_rtime;      /* last msgrcv time */
        __kernel_time_t msg_ctime;      /* last change time */
        unsigned long  msg_lcbytes;     /* Reuse junk fields for 32 bit */
        unsigned long  msg_lqbytes;     /* ditto */
        unsigned short msg_cbytes;      /* current number of bytes on queue */
        unsigned short msg_qnum;        /* number of messages in queue */
        unsigned short msg_qbytes;      /* max number of bytes on queue */
        __kernel_ipc_pid_t msg_lspid;   /* pid of last msgsnd */
        __kernel_ipc_pid_t msg_lrpid;   /* last receive pid */
};
```

其中 msg 结构的定义如下, 我们在实际项目中几乎很少使用如下的结构, 一般都需要我们根据实际的需求自行定义.

```
Struct msg{
    struct msg* msg_next;
    long msg_type;
    long msg_ts;
    short msg_spot;
};
```

理论上可以通过结构 msqid_ds 的成员 msg_first, msg_last 和结构 msg 的成员 msg_next 遍历全部消息队列并完成管理和维护消息队列的功能, 但实际上这三个成员是 UNIX 内核的直辖数据, 用户无法引用.

消息队列中消息本身是由消息类型和消息数据组成, 我们常常使用如下结构作为消息模板:

```
struct msgbuf {
        long mtype;          /* type of message */
        char mtext[1];       /* message text */
};
```

根据消息类型的不同, 我们可以在同一个信息队列中定义不同功能的消息.

使用消息队列

(1) 消息队列的创建

在 UNIX 中, 采用函数 msgget 创建消息队列

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

extern int msgget (key_t __key, int __msgflg) __THROW;
```

函数 msgget 创建一个新的消息队列, 或者访问一个已经存在的消息队列. 调用成功时返回消息队列的标志符, 否则返回-1.

(2) 消息队列的发送和接收

在 UNIX 中函数 msgsnd 向消息队列发送消息, 原型如下:

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

extern int msgsnd (int __msqid, __const void *__msgp, size_t __msgsz, int __msgflg) __THROW;
```

发送消息一般分五个步骤:

a) 根据自己的需要定义消息结构

```
struct msgbuf {

        long mtype;             /* type of message */

        char ctext[100];        /* message data */

};
```

b) 打开或创建消息队列

```
msgid = msgget(Key, 0666 | IPC_CREAT);

if(msgid < 0) 打开(或创建)队列失败
```

c) 组装消息

d) 发送消息

```
int ret;

ret = msgsnd(msgid, (void *)&buf, strlen(buf.ctext), IPC_NOWAIT);
```

e) 发送判断

```
If (ret == -1){

    if (errno == EINTR) 信号中断, 重新发送;

    else 系统错误

}
```

下面是一个发送消息的实例: 它循环读取键盘输入, 将输入的字符串信息写入到消息队列(关键字为 0x1234)中.

```
[bill@billstone Unix_study]$ cat msg1.c

#include <sys/msg.h>

#include <sys/types.h>

#include <sys/ipc.h>
```

```
#include <stdio.h>
#include <sys/errno.h>

extern int errno;

struct mymsgbuf{
        long mtype;
        char ctext[100];
};

int main(void)
{
        struct mymsgbuf buf;
        int msgid;

        if((msgid = msgget(0x1234, 0666 | IPC_CREAT)) < 0){
                fprintf(stderr, "open msg %X failed\n", 0x1234);
                exit(1);
        }
        while(1){
                printf("Please input: ");
                memset(&buf, 0, sizeof(buf));
                fgets(buf.ctext, sizeof(buf.ctext), stdin);
                buf.mtype = getpid();
                while((msgsnd(msgid, (void *)&buf, strlen(buf.ctext), IPC_NOWAIT)) < 0){
                        if(errno == EINTR)
                                continue;
                        exit(2);
                }
                if(!strncmp(buf.ctext, "exit", 4))
                        break;
        }

        return 0;
}
[bill@billstone Unix_study]$
```

运行结果如下:

```
[bill@billstone Unix_study]$ make msg1
cc       msg1.c     -o msg1
[bill@billstone Unix_study]$ ipcs -q


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


[bill@billstone Unix_study]$ ./msg1
Please input: Hello world!
Please input: Nice to meet you!
Please input: bye!
Please input: exit
[bill@billstone Unix_study]$ ipcs -q


------ Message Queues --------
key            msqid          owner          perms          used-bytes     messages
0x00001234 98304          bill           666            41             4


[bill@billstone Unix_study]$
```

在 UNIX 中函数 msgrcv 从消息队列中接收消息, 原型如下:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
extern int msgrcv (int __msqid, void *__msgp, size_t __msgsz, long int __msgtyp, int __msgflg) __THROW;
```

与函数 msgsnd 不同,这里参数 msgsz 指的是缓冲区的最大容量,包括消息类型占用的部分.

这里配合上面的例子设计接收消息的实例: 以阻塞方式不断地从消息队列(关键字为 0x1234)中读取消息, 并打印接收到的消息类型, 长度和数据等信息, 当接收到数据内容为'exit'时程序结束.

```
[bill@billstone Unix_study]$ cat msg2.c
#include <sys/msg.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/errno.h>


extern int errno;


struct mymsgbuf{
        long mtype;
```

```
        char ctext[100];
};

int main(void)
{
        struct mymsgbuf buf;
        int msgid, ret;

        if((msgid = msgget(0x1234, 0666 | IPC_CREAT)) < 0){
                fprintf(stderr, "open msg %X failed\n", 0x1234);
                exit(1);
        }
        while(1){
                memset(&buf, 0, sizeof(buf));
                while((ret = msgrcv(msgid, (void *)&buf, sizeof(buf), 0, 0)) < 0){
                        if(errno == EINTR)
                                continue;
                        fprintf(stderr, "Error no: %d", errno);
                        exit(2);
                }
                fprintf(stderr, "Msg: Type=%d, Len=%d, Text:%s", buf.mtype, ret, buf.ctext);
                if(!strncmp(buf.ctext, "exit", 4))
                        break;
        }

        return 0;
}
[bill@billstone Unix_study]$
```

运行结果如下:

```
[bill@billstone Unix_study]$ make msg2
cc      msg2.c    -o msg2
[bill@billstone Unix_study]$ ipcs -q

------ Message Queues --------
key          msqid        owner        perms        used-bytes    messages
0x00001234 98304        bill         666          41              4
```

```
[bill@billstone Unix_study]$ ./msg2
Msg: Type=15666, Len=13, Text:Hello world!
Msg: Type=15666, Len=18, Text:Nice to meet you!
Msg: Type=15666, Len=5, Text:bye!
Msg: Type=15666, Len=5, Text:exit
[bill@billstone Unix_study]$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00001234 98304      bill       666        0            0

[bill@billstone Unix_study]$
```

从上面可以看到, 采用消息队列通信比采用管道通信具有更多的灵活性.

系统调用 msgctl 对消息队列进行各种控制, 包括查询消息队列数据结构, 改变消息队列访问权限, 改变消息队列属主信息和删除消息队列等, 原型如下:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

根据 cmd 参数对 msqid 消息队列操作:

a) IPC_RMID: 删除消息队列

b) IPC_STAT: 读取消息队列

c) IPC_SET: 重置消息队列结构 msqid_ds 中的成员 uid, gid, mode 及 msg_qbytes.

第十三章 信号量

进程间的通信不仅仅包括数据交流, 也包括过程控制.

信号量是一个可以用来控制进程存储共享资源的计数器, 它可以是跟踪共享资源的生产和消费的计数器, 也可以是协调资源的生产者和消费者之间的同步器, 还可以是控制生产进程和消费进程的互斥开关.

信号量简介

操作系统通过信号量和 PV 操作, 可以完成同步和互斥操作.

信号量集合由一个或多个信号量集合组成, IPC 对象中的'信号量'通常指的是信号量集合, UNIX 的内核采用结构 semid_ds 来管理信号量, 结构如下:

```
struct semid_ds {
        struct ipc_perm sem_perm;               /* permissions .. see ipc.h */
        __kernel_time_t sem_otime;              /* last semop time */
        __kernel_time_t sem_ctime;              /* last change time */
        struct sem      *sem_base;              /* ptr to first semaphore in array */
        struct sem_queue *sem_pending;          /* pending operations to be processed */
```

```
        struct sem_queue **sem_pending_last;      /* last pending operation */
        struct sem_undo *undo;                    /* undo requests on this array */
        unsigned short   sem_nsems;               /* no. of semaphores in array */
};
```

指针 sem_base 指向一个信号量数组,信号量由结构 sem 记载,如下所示:

```
Struct sem{
    unsigned short semval;          // 信号量取值
    pid_t sempid;                   // 最近访问进程 ID
    unsigned short semncnt;         // P 阻塞进程数
    unsigned short semzcnt;         // Z 阻塞进程数
}
```

在 Shell 中可以通过'ipcs -a -s'命令查询系统中信号量的基本信息.

(1) 信号量的创建

在 UNIX 中, 函数 semget 用来创建信号量, 原型如下:

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

函数 semget 创建一个新的信号量, 或者访问一个已经存在的信号量.

(2) 信号量的控制

系统调用 semctl 用来控制信号量, 原型如下:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

函数 semctl 对标识号为 semid 的信号量集合中序号为 semnum 的信号量进行赋值, 初始化, 信息获取和删除等多相操作, 参数 cmd 指定了操作的类型, 参数 arg 指定了函数输入输出的缓冲区, 定义如下:

```
union semun {
        int val;                    /* value for SETVAL */
        struct semid_ds *buf;       /* buffer for IPC_STAT & IPC_SET */
        unsigned short *array;      /* array for GETALL & SETALL */
        struct seminfo *__buf;      /* buffer for IPC_INFO */
        void *__pad;
};
```

函数 semctl 的第四个参数 arg 在本质上是一个 4 字节的缓冲区. 调用失败时返回-1 并置 errno.

本处设计一个类似于命令'ipcs'和命令'ipcrm'的程序 ipcsem, 它从命令行参数中获取要执行的操作, 包括创建信号量, 读取信号量信息, 读取信号量取值和删除信号量等, 程序如下:

```
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <stdio.h>
```

```
#define VerifyErr(a, b)   \
        if (a) fprintf(stderr, "%s failed.\n", (b));   \
        else fprintf(stderr, "%s success.\n", (b));

int main(int argc, char *argv[1])
{
        int semid, index, i;
        unsigned short array[100];
        struct semid_ds ds;

        if(argc != 4)
                return 0;
        semid = atoi(argv[1]);
        index = atoi(argv[2]);
        if(argv[3][0] == 'c'){
                VerifyErr(semget(semid, index, 0666|IPC_CREAT|IPC_EXCL) < 0, "Create sem");
        }
        else if(argv[3][0] == 'd'){
                VerifyErr(semctl(semid, 0, IPC_RMID, NULL) < 0, "Delete sem");
        }
        else if(argv[3][0] == 'v'){
                fprintf(stderr, "T      ID      INDEX      SEMVAL      SEMIPID      SEMNCNT
SEMZCNT\n");
                fprintf(stderr, "s %6d %6d %10d %10d %10d %10d\n", semid, index,
                        semctl(semid, index, GETVAL), semctl(semid, index, GETPID),
                        semctl(semid, index, GETNCNT), semctl(semid, index, GETZCNT));
        }
        else if(argv[3][0] == 'a'){
                ds.sem_nsems = 0;
                VerifyErr(semctl(semid, 0, IPC_STAT, &ds) != 0, "Get Sem Stat");
                VerifyErr(semctl(semid, 0, GETALL, array) != 0, "Get Sem All");
                for(i=0;i<ds.sem_nsems;i++)
                        fprintf(stderr, "sem no [%d]: [%d]\n", i, array[i]);
        }
        else
                VerifyErr(semctl(semid, index, SETVAL, atoi(argv[3])) != 0, "Set Sem Val");
```

```
        return 0;
}
```

执行结果如下:

```
[bill@billstone Unix_study]$ make ipcsem
cc      ipcsem.c    -o ipcsem
[bill@billstone Unix_study]$ ipcs -s

------ Semaphore Arrays --------
key         semid      owner      perms      nsems
0x000003e8 0          bill       666        10

[bill@billstone Unix_study]$ ./ipcsem 2000 2 c
Create sem success.
[bill@billstone Unix_study]$ ipcs -s

------ Semaphore Arrays --------
key         semid      owner      perms      nsems
0x000003e8 0          bill       666        10
0x000007d0 65537      bill       666        2

[bill@billstone Unix_study]$ ./ipcsem 65537 0 100
Set Sem Val success.
[bill@billstone Unix_study]$ ./ipcsem 65537 0 v
T     ID      INDEX     SEMVAL    SEMIPID     SEMNCNT     SEMZCNT
s   65537       0         100       23829          0           0
[bill@billstone Unix_study]$ ./ipcsem 65537 1 200
Set Sem Val success.
[bill@billstone Unix_study]$ ./ipcsem 65537 0 a
Get Sem Stat success.
Get Sem All success.
sem no [0]: [100]
sem no [1]: [200]
[bill@billstone Unix_study]$ ./ipcsem 65537 0 d
Delete sem success.
[bill@billstone Unix_study]$ ipcs -s
```

```
------ Semaphore Arrays --------
key         semid      owner      perms      nsems
0x000003e8 0          bill       666        10


[bill@billstone Unix_study]$
```

操作信号量

信号量具有 P, V 和 Z 三种操作, 在 UNIX 中, 这些操作可以通过函数 semop 调用完成, 函数 semop 可以一次性操作同一信号量集合中的多个信号量. 原型如下:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

函数 semop 对标识号为 semid 的信号量集合中的一个或多个信号量执行信号数值的增加, 减少或比较操作. 参数 sops 指向一个 sembuf 结构的缓冲区, nsops 指定了缓冲区中存储的 sembuf 结构的个数.

```
struct sembuf {
        unsigned short  sem_num;        /* semaphore index in array */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

其中, 第一个信号的序号是 0. sem_op 指定了操作的类型:

a) 正数. V 操作.
b) 负数. P 操作.
c) 0 . Z 操作. 判断信号量数值是否等于 0.

而 sem_flg 取值有 IPC_NOWAIT 和 SEM_UNDO 等.

下面是一个用于临界资源的读写控制和并发进程的同步和互斥控制的实例: 假设进程 A 是生产者, 进程 B 是消费者, 系统最多只能同时容纳 5 个产品, 初始成品数为 0. 当产品数不足 5 时允许进程 A 生产, 当产品数超过 0 时允许进程 B 消费.

这里需要两个信号量模拟生产-消费过程. 信号量 A 代表了当前生产的数目, 它控制了生产者进程 A, 信号量 n 代表当前尚有 n 个成品可以生产.

信号 B 代表了当前的产品数, 他控制消费者进程 B, 当信号量为 n 时剩余 n 个产品.

生产者进程 sema.c 如下:

```
[bill@billstone Unix_study]$ cat sema.c
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/stat.h>


#define VerifyErr(a,b) \
        if (a) { fprintf(stderr, "%s failed.\n", (b)); exit(1); } \
```

```
        else fprintf(stderr, "%s success.\n", (b));


int main(void)
{
        int semid;
        struct sembuf sb;


        VerifyErr((semid = semget(2000, 2, 0666)) < 0, "Open Sem 2000");
        sb.sem_num = 0;
        sb.sem_op = -1;
        sb.sem_flg &= ~IPC_NOWAIT;
        VerifyErr(semop(semid, &sb, 1) != 0, "P sem 2000:0");
        fprintf(stderr, "[%d] producing ... ... \n", getpid());
        sleep(1);
        fprintf(stderr, "[%d] produced\n", getpid());
        sb.sem_num = 1;
        sb.sem_op = 1;
        sb.sem_flg &= ~IPC_NOWAIT;
        VerifyErr(semop(semid, &sb, 1) != 0, "V sem 2000:1");


        return 0;
}
[bill@billstone Unix_study]$
```

消费者进程 semb.c 如下:

```
[bill@billstone Unix_study]$ cat semb.c
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/stat.h>


#define VerifyErr(a,b) \
        if (a) { fprintf(stderr, "%s failed.\n", (b)); exit(1); } \
        else fprintf(stderr, "%s success.\n", (b));


int main(void)
{
```

```
        int semid;
        struct sembuf sb;

        VerifyErr((semid = semget(2000, 2, 0666)) < 0, "Open Sem 2000");
        sb.sem_num = 1;
        sb.sem_op = -1;
        sb.sem_flg &= ~IPC_NOWAIT;
        VerifyErr(semop(semid, &sb, 1) != 0, "P sem 2000:1");
        fprintf(stderr, "[%d] consuming ... ... \n", getpid());
        sleep(1);
        fprintf(stderr, "[%d] consumed\n", getpid());
        sb.sem_num = 0;
        sb.sem_op = 1;
        sb.sem_flg &= ~IPC_NOWAIT;
        VerifyErr(semop(semid, &sb, 1) != 0, "V sem 2000:1");

        return 0;
}
[bill@billstone Unix_study]$
```

编译程序并使用之前的程序 ipcsem 创建信号量集合:

```
[bill@billstone Unix_study]$ ./ipcsem 2000 2 c
Create sem success.
[bill@billstone Unix_study]$ ipcs -s

------ Semaphore Arrays --------
key         semid      owner      perms      nsems
0x000003e8 0          bill       666        10
0x000007d0 98305      bill       666        2

[bill@billstone Unix_study]$ ./ipcsem 98305 0 5
Set Sem Val success.
[bill@billstone Unix_study]$ ./ipcsem 98305 1 0
Set Sem Val success.
[bill@billstone Unix_study]$ ./ipcsem 98305 0 a
Get Sem Stat success.
Get Sem All success.
sem no [0]: [5]
```

```
sem no [1]: [0]
[bill@billstone Unix_study]$
```

在一个终端上运行 sema:

```
[bill@billstone Unix_study]$ ./ipcsem 98305 0 a
Get Sem Stat success.
Get Sem All success.
sem no [0]: [3]
sem no [1]: [2]
[bill@billstone Unix_study]$ ./sema
Open Sem 2000 success.
P sem 2000:0 success.
[23940] producing ... ...
[23940] produced
V sem 2000:1 success.
[bill@billstone Unix_study]$ ./ipcsem 98305 0 a
Get Sem Stat success.
Get Sem All success.
sem no [0]: [2]
sem no [1]: [3]
[bill@billstone Unix_study]$
```

在另一个终端上执行 semb:

```
[bill@billstone Unix_study]$ ./ipcsem 98305 0 a
Get Sem Stat success.
Get Sem All success.
sem no [0]: [2]
sem no [1]: [3]
[bill@billstone Unix_study]$ ./semb
Open Sem 2000 success.
P sem 2000:1 success.
[23942] consuming ... ...
[23942] consumed
V sem 2000:1 success.
[bill@billstone Unix_study]$ ./ipcsem 98305 0 a
Get Sem Stat success.
Get Sem All success.
sem no [0]: [3]
sem no [1]: [2]
[bill@billstone Unix_study]$
```

第十四章 共享内存

管道, 消息队列和信号量都需要借助第三方对象进行通信; 而共享内存正好弥补了这些缺陷, 它是最快的 IPC 对象. 在本质上, 共享内存是一端物理内存.

共享内存简介

共享内存中最重要的属性是内存大小和内存地址, 进程在访问共享内存前必须先将共享内存映射到进程空间的一个虚拟地址中, 然后任何对该虚拟地址的数据操作都将直接作用到物理内存上.

共享内存由进程创建, 但是进程结束时共享内存仍然保留, 除非该共享内存被显式地删除或者重启操作系统.

UNIX 的内核采用结构 shmid_ds 来管理消息队列, 它的数据成员与命令'ipcs -a -m'的结果一一对应

```
struct shmid_ds {
        struct ipc_perm         shm_perm;       /* operation perms */
        int                     shm_segsz;      /* size of segment (bytes) */
        __kernel_time_t         shm_atime;      /* last attach time */
        __kernel_time_t         shm_dtime;      /* last detach time */
        __kernel_time_t         shm_ctime;      /* last change time */
        __kernel_ipc_pid_t      shm_cpid;       /* pid of creator */
        __kernel_ipc_pid_t      shm_lpid;       /* pid of last operator */
        unsigned short          shm_nattch;     /* no. of current attaches */
        unsigned short          shm_unused;     /* compatibility */
        void                    *shm_unused2;   /* ditto - used by DIPC */
        void                    *shm_unused3;   /* unused */
};
```

命令'ipcs -m'查询系统中共享内存的基本信息:

```
[bill@billstone bill]$ ipcs -m


------ Shared Memory Segments --------
key         shmid      owner      perms      bytes      nattch     status


[bill@billstone bill]$
```

使用共享内存

(1) 共享内存的创建

在 UNIX 中, 函数 shmget 用来创建或获取共享内存, 原型如下:

```
#include <sys/ipc.h>
```

```
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
```

同样的, 参数 shm_flg 有 IPC_CREAT 和 IPC_EXCL 两种取值.

(2) 共享内存的映射

与消息队列和信号量不同, 共享内存在获取标志号后, 仍需要调用 shmat 函数将共享内存映射到进程的地址空间后才能访问, 原型如下

```
#include <sys/types.h>

#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

(3) 共享内存的释放

当进程不再需要共享内存时, 可以使用函数 shmdt 释放共享内存内存映射, 原型如下

```
#include <sys/types.h>

#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

(4) 共享内存的控制

与消息队列和信号量一样, 共享内存也有自给的控制函数 shmctl, 原型如下:

```
#include <sys/ipc.h>

#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

这里设计一个类似于命令'ipcs'和命令'ipcrm'的程序 ipcshm: 它从命令行参数中获取要执行的操作, 包括创建共享内存, 读取共享内存信息和删除共享内存等, 主体代码如下代码如下

```
#include <sys/shm.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <stdio.h>

#include <sys/stat.h>


#define VerifyErr(a, b) \

                if (a) fprintf(stderr, "%s failed.\n", (b)); \

                else fprintf(stderr, "%s success.\n", (b));


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

{

                int shmid, size;


                if(argc != 3)

                                exit(1);

                shmid = atoi(argv[1]);

                if(strcmp(argv[2], "v") == 0){

                                StatShm(shmid);
```

```
		}
		else if(strcmp(argv[2], "d") == 0){
			VerifyErr(shmctl(shmid, IPC_RMID, NULL) < 0, "Delete Shm");
		}
		else{
			size = atoi(argv[2]);
			VerifyErr(shmget(shmid, size, 0666|IPC_CREAT|IPC_EXCL) < 0, "Create Shm");
		}

		return 0;
}
```

其中用到了两个函数:

```
char * GetFileMode(mode_t st_mode, char *resp){
	if(resp == NULL)
		return 0;
	memset(resp, '-', 9);
	if(st_mode & S_IRUSR)	resp[0] = 'r';
	if(st_mode & S_IWUSR)	resp[1] = 'w';
	if(st_mode & S_IXUSR)	resp[2] = 'x';
	if(st_mode & S_IRGRP)	resp[3] = 'r';
	if(st_mode & S_IWGRP)	resp[4] = 'w';
	if(st_mode & S_IXGRP)	resp[5] = 'x';
	if(st_mode & S_IROTH)	resp[6] = 'r';
	if(st_mode & S_IWOTH)	resp[7] = 'w';
	if(st_mode & S_IXOTH)	resp[8] = 'x';

	return resp;
}

int StatShm(int shmid){
		char resp[10];
		struct shmid_ds buf;

		memset(&buf, 0, sizeof(buf));
		memset(resp, 0, sizeof(resp));
		shmctl(shmid, IPC_STAT, &buf);
		fprintf(stderr, "T	ID	KEY	MODE	OWNER	GROUP	NATTCH	SEGSZ	CPID
LPID\n");
```

```
                fprintf(stderr, "m %6d %#6x %s %6d %6d %6d %10d %10d %10d\n", shmid, buf.shm_perm.__key,
                        GetFileMode(buf.shm_perm.mode, resp), buf.shm_perm.uid, buf.shm_perm.gid,
                        buf.shm_nattch, buf.shm_segsz, buf.shm_cpid, buf.shm_lpid);


                return 0;
}
```

执行情况如下:

```
[bill@billstone Unix_study]$ gcc -o ipcshm ipcshm.c
[bill@billstone Unix_study]$ ./ipcshm 2000 100
Create Shm success.
[bill@billstone Unix_study]$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000007d0 229377     bill       666        100        0

[bill@billstone Unix_study]$ ./ipcshm 229377 v
T     ID     KEY        MODE       OWNER    GROUP   NATTCH     SEGSZ     CPID      LPID
m 229377   0x7d0 rw-rw-rw-      500      500        0          100        1796          0
[bill@billstone Unix_study]$ ./ipcshm 229377 d
Delete Shm success.
[bill@billstone Unix_study]$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

[bill@billstone Unix_study]$
```

(5) 共享内存实例

共享内存的应用可以分为打开(创建)共享内存, 映射共享内存, 读写共享内存和释放共享内存映射等四个步骤.

程序 shm1 想共享内存中指定位置写入数据.

```
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/stat.h>

#define VerifyErr(a, b) \
```

```
                if (a) { fprintf(stderr, "%s failed.\n", (b)); return; } \
                else fprintf(stderr, "%s success.\n", (b));


int main(void)
{
                int shmid, no;
                char *pmat = NULL, buf[1024];

                VerifyErr((shmid = shmget(0x1234, 10*1024, 0666|IPC_CREAT)) == -1, "Open(Create) Shm");
                VerifyErr((pmat = (char *)shmat(shmid, 0, 0)) == 0, "Link Shm");
                printf("Please input NO.(0-9): ");
                scanf("%d", &no);
                VerifyErr(no<0 || no>9, "Input No.");
                printf("Please input data: ");
                memset(buf, 0, sizeof(buf));
                getchar();                      // 读入'\n'回车符
                fgets(buf, sizeof(buf), stdin);
                memcpy(pmat+no*1024, buf, 1024);
                shmdt(pmat);

                return 0;
}
```

程序 shm2 从共享内存指定位置读取数据

```
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <sys/stat.h>

#define VerifyErr(a, b) \
                if (a) { fprintf(stderr, "%s failed.\n", (b)); return; } \
                else fprintf(stderr, "%s success.\n", (b));

int main(void)
{
                int shmid, no;
                char *pmat = NULL, buf[1024];
```

```
        VerifyErr((shmid = shmget(0x1234, 10*1024, 0666|IPC_CREAT)) == -1, "Open(Create) Shm");
        VerifyErr((pmat = (char *)shmat(shmid, 0, 0)) == 0, "Link Shm");
        printf("Please input NO.(0-9): ");
        scanf("%d", &no);
        VerifyErr(no<0 || no>9, "Input No.");
        memcpy(buf, pmat+no*1024, 1024);
        printf("Data: [%s]\n", buf);
        shmdt(pmat);

        return 0;
}
```

运行结果如下

```
[bill@billstone Unix_study]$ make shm1
cc      shm1.c    -o shm1
[bill@billstone Unix_study]$ make shm2
cc      shm2.c    -o shm2
[bill@billstone Unix_study]$ ./shm1
Open(Create) Shm success.
Link Shm success.
Please input NO.(0-9): 1
Input No. success.
Please input data: this is a test!
[bill@billstone Unix_study]$ ./shm2
Open(Create) Shm success.
Link Shm success.
Please input NO.(0-9): 1
Input No. success.
Data: [this is a test!
]
[bill@billstone Unix_study]$
```

第五篇 网络通信篇

IPC 对象只能实现在一台主机中的进程相互通信, 网罗通信对象则打破了这个限制, 它如同电话和邮件, 可以帮助不通屋檐下的人们相互交流.

套接字(Socket)是网络通信的一种机制, 它已经被广泛认可并成为事实上的工业标准.

第十五章 基于TCP的通信程序

TCP 是一种面向连接的网络传输控制协议. 它每发送一个数据, 都要对方确认, 如果没有接收到对方的确认, 就将自动重新发送数据, 直到多次重发失败后,才放弃发送.

套接字的完整协议地址信息包括协议, 本地地址, 本地端口, 远程地址和远程端口等内容, 不同的协议采用不同的结构存储套接字的协议地址信息.

Socket 是进程间的一个连接, 我们可以采用协议, 地址和端口的形式描述它:

{ 协议, 本地地址, 本地端口, 远程地址, 远程端口 }

当前有三种常见的套接字类型, 分别是流套接字(SOCK_STREAM), 数据报套接字(SOCK_DGRAM)和原始套接字(SOCK_RAW):

1) 流套接字. 提供双向的, 可靠的, 顺序的, 不重复的, 面向连接的通信数据流. 它使用了 TCP 协议保真了数据传输的正确性.

2) 数据报套接字. 提供一种独立的, 无序的, 不保证可靠的无连接服务. 它使用了 UDP 协议, 该协议不维护一个连接, 它只把数据打成一个包, 再把远程的 IP 贴上去, 然后就把这个包发送出去.

3) 原始套接字. 主要应用于底层协议的开发, 进行底层的操作.

TCP协议的基础编程模型

TCP 是面向连接的通信协议, 采用客户机-服务器模式, 套接字的全部工作流程如下:

首先, 服务器端启动进程, 调用 socket 创建一个基于 TCP 协议的流套接字描述符.

其次, 服务进程调用 bind 命名套接字, 将套接字描述符绑定到本地地址和本地端口上, 至此 socket 的半相关描述----{协议, 本地地址, 本地端口}----完成.

再次, 服务器端调用 listen, 开始侦听客户端的 socket 连接请求.

接下来, 客户端创建套接字描述符, 并且调用 connect 向服务器提交连接请求. 服务器端接收到客户端连接请求后, 调用 accept 接受, 并创建一个新的套接字描述符与客户端建立连接, 然后原套接字描述符继续侦听客户端的连接请求.

客户端与服务器端的新套接字进行数据传送, 调用 write 或 send 向对方发送数据, 调用 read 或 recv 接收数据.

在数据交流完毕后, 双方调用 close 或 shutdown 关闭套接字.

(1) Socket 的创建

在 UNIX 中使用函数 socket 创建套接字描述符, 原型如下:

```
#include <sys/types.h>

#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

其中, 参数 domain 指定发送通信的域, 有两种选择: AF_UNIX, 本地主机通信, 功能和 IPC 对象类似; AF_INET, Internet 地址 IPV4 协议. 在实际编程中, 我们只使用 AF_INET 协议, 如果需要与本地主机进程建立连接, 只需把远程地址设定为'127.0.0.1'即可.

参数 type 指定了通信类型: SOCK_STREAM, SOCK_DGRAM 和 SOCK_RAW. 协议 AF_INET 支持以上三种类型, 而协议 AF_UNIX 不支持原始套接字.

(2) Socket 的命名

函数 bind 命名一个套接字, 它为该套接字描述符分配一个半相关属性, 原型如下:

```
#include <sys/types.h>

#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

参数 s 指定了套接字描述符, 该值由函数 socket 返回, 指针 name 指向通用套接字的协议地址结构, namelen 参数指定了该协议地址结构的长度.

结构 sockaddr 描述了通用套接字的相关属性, 结构如下

```
typedef unsigned short int sa_family_t;

#define __SOCKADDR_COMMON(sa_prefix)   sa_family_t sa_prefix##family

struct sockaddr{

    __SOCKADDR_COMMON (sa_);      /* Common data: address family and length.   */

    char sa_data[14];              /* Address data.   */

};
```

不同的协议有不同的地址描述方式, 为了便于编码处理, 每种协议族都定义了自给的套接字地址属性结构, 协议族 AF_INET 使用结构 sockaddr_in 描述套接字地址信息, 结构如下:

```
struct sockaddr_in{

    __SOCKADDR_COMMON (sin_);

    in_port_t sin_port;                    /* Port number.   */

    struct in_addr sin_addr;               /* Internet address.   */

                                    /* Pad to size of `struct sockaddr'.   */

    unsigned char sin_zero[sizeof (struct sockaddr) - __SOCKADDR_COMMON_SIZE -    \

                           sizeof (in_port_t) - sizeof (struct in_addr)];

};

typedef uint32_t in_addr_t;

struct in_addr{

    in_addr_t s_addr;

};
```

这里有两点需要注意:

a. IP 地址转换

在套接字的协议地址信息结构中, 有一个描述 IP 地址的整型成员. 我们习惯使用点分方式描述 IP 地址, 所以需要将其转化为整型数据, 下列函数完成此任务

```
#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet_aton(const char *cp, struct in_addr *inp);

in_addr_t inet_addr(const char *cp);

char *inet_ntoa(struct in_addr in);
```

函数 inet_addr 将参数 ptr 指向的字符串形式 IP 地址转换为 4 字节的整型数据. 函数 inet_aton 同样完成此功能. 函数 inet_ntoa 的功能则恰好相反.

b. 字节顺序转换

网络通信常常跨主机, 跨平台, 跨操作系统, 跨硬件设备, 但不同的 CPU 硬件设备, 不同的操作系统对内存数据的组织结构不尽相同. 在网络通信中, 不同的主机可能采取了不同的记录顺序, 如果不做处理, 通信双方对相同的数据会有不同的解释. 所以需要函数实现主机字节顺序和网络字节顺序的转换

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);
```

```
uint16_t ntohs(uint16_t netshort);
```

函数 htons, htonl 分别将 16 位和 32 位的整数从主机字节顺序转换为网络字节顺序.

函数 ntohs, ntohl 分别将 16 位和 32 位的整数从网络字节顺序转换为主机字节顺序.

(3) Socket 的侦听

TCP 的服务器端必须调用 listen 才能使套接字进入侦听状态, 原型如下

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

参数 s 是调用 socket 创建的套接字. 参数 backlog 则确定了套接字 s 接收连接的最大数目.

在 TCP 通信模型中, 服务器端进程需要完成创建套接字, 命名套接字和侦听接收等一系列操作才能接收客户端连接请求. 下面设计了一个封装了以上三个操作的函数, 代码如下

```
int CreateSock(int *pSock, int nPort, int nMax){

        struct sockaddr_in addrin;

        struct sockaddr *paddr = (struct sockaddr *)&addrin;

        int ret = 0;          // 保存错误信息


        if(!((pSock != NULL) && (nPort > 0) && (nMax > 0))){

                printf("input parameter error");

                ret = 1;

        }

        memset(&addrin, 0, sizeof(addrin));


        addrin.sin_family = AF_INET;

        addrin.sin_addr.s_addr = htonl(INADDR_ANY);

        addrin.sin_port = htons(nPort);

                // 创建 socket, 在我本机上是 5

        if((ret == 0) && (*pSock = socket(AF_INET, SOCK_STREAM, 0)) <= 0){

                printf("invoke socket error\n");

                ret = 1;

        }

              // 绑定本地地址

        if((ret == 0) && bind(*pSock, paddr, sizeof(addrin)) != 0){

                printf("invoke bind error\n");

                ret = 1;

        }


        if((ret == 0) && listen(*pSock, nMax) != 0){

                printf("invoke listen error\n");

                ret = 1;

        }
```

```
        close(*pSock);
        return(ret);
}
```

(4) Socket 的连接处理

服务器端套接字在进入侦听状态后, 通过 accept 接收客户端的连接请求

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

函数 accept 一旦调用成功, 系统将创建一个属性与套接字 s 相同的新的套接字描述符与客户进程通信, 并返回该新套接字的描述符编号, 而原套接字 s 仍然用于套接字侦听. 参数 addr 回传连接成功的客户端地址结构, 指针 addrlen 回传该结构占用的字节空间大小.

下面封装了系统调用 accept, 代码如下

```
#include <fcntl.h>
int AcceptSock(int *pSock, int nSock){
        struct sockaddr_in addrin;
        int lSize, flags;

        if((pSock == NULL) || (nSock <= 0)){
                printf("input parameter error!\n");
                return 2;
        }
        flags = fcntl(nSock, F_GETFL, 0);           // 通过 fcntl 函数确保 nSock 处于阻塞方式
        fcntl(nSock, F_SETFL, flags & ~O_NONBLOCK);
        while(1){
                lSize = sizeof(addrin);
                memset(&addrin, 0, sizeof(addrin));       // 通过调试, 问题应该出在 accept 函数
                if((*pSock = accept(nSock, (struct sockaddr *)&addrin, &lSize)) > 0)
                        return 0;
                else if(errno == EINTR)
                        continue;
                else{
                        fprintf(stderr, "Error received! No: %d\n", errno);
                        return 1;
                }
        }
}
```

(5) Socket 的关闭

套接字可以调用 close 函数关闭, 也可以调用下面函数

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

函数 shutdown 是强制性地关闭所有套接字连接, 而函数 close 只将套接字访问计数减 1, 当且仅当计数器值为 0 时, 系统才真正的关闭套接字通信.

(6) Socket 的连接申请

TCP 客户端调用 connect 函数向 TCP 服务器端发起连接请求, 原型如下

```
#include <sys/types.h>

#include <sys/socket.h>

int   connect(int   sockfd,   const   struct sockaddr *serv_addr, socklen_t addrlen);
```

其中, serv_addr 指针指定了对方的套接字地址结构.

(7) TCP 数据的发送和接收

套接字一旦连接上, 就可以发送和接收数据. 原型如下

```
#include <sys/types.h>

#include <sys/socket.h>

int send(int s, const void *msg, size_t len, int flags);

int recv(int s, void *buf, size_t len, int flags);
```

函数 send(recv)应用于 TCP 协议的套接字通信中, s 是与远程地址连接的套接字描述符, 指针 msg 指向待发送的数据信息(或接收数据的缓冲区), 此信息共 len 个字节(或最大可接收 len 个字节).

如果函数 send 一次性发送的信息过长, 超过底层协议的最大容量, 就必须分开调用 send 发送, 否则内核将不予发送信息并且置 EMSGSIZE 错误.

简单服务器端程序

这里设计了一个 TCP 服务器端程序的实例, 它创建 Socket 侦听端口, 与客户端建立连接, 然后接收并打印客户端发送的数据,代码如下

```
#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdio.h>

#include <errno.h>


#define VerifyErr(a, b) \

                if (a) { fprintf(stderr, "%s failed.\n", (b)); return 0; } \

                else fprintf(stderr, "%s success.\n", (b));


int main(void)

{

                int nSock, nSock1;

                char buf[2048];


                //CreateSock(&nSock, 9001, 9);
```

```
        nSock = nSock1 = 0;                    // 这里只是为了调试所用
        VerifyErr(CreateSock(&nSock, 9001, 9) != 0, "Create Listen SOCKET");
        //VerifyErr(AcceptSock(&nSock1, nSock) != 0, "Link");
        AcceptSock(&nSock1, nSock);
        memset(buf, 0, sizeof(buf));
        recv(nSock1, buf, sizeof(buf), 0);
        fprintf(stderr, buf);
        close(nSock1);
        close(nSock);

        return 0;
}
```

运行程序, 并在在浏览器中输入http://xxx.xxx.xxx.xxx:9001/, 你将得到一条来自客户端的http报文. 但是在这里却出现了问题, 问题如下:

```
[bill@billstone Unix_study]$ make tcp1
cc      tcp1.c    -o tcp1
[bill@billstone Unix_study]$ ./tcp1
Create Listen SOCKET success.
Error received! No: 9                          // 出现错误, 为'bad file number'错误
[bill@billstone Unix_study]$
```

第十六章 基于UDP的通信程序设计

UDP 协议使用函数 sendto 发送数据, 使用函数 recvfrom 接收数据.

函数 sendto 的原型如下

```
#include <sys/types.h>
#include <sys/socket.h>
int   sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```

通过函数 sendto 发送数据时总能立即成功返回. 注意, 这里'成功'的含义是指数据被成功的发送, 并不保证对方套接字能成功的接收, 甚至不保证对方套接字协议信息的正确性.

函数 recvfrom 的原型如下

```
#include <sys/types.h>
#include <sys/socket.h>
int   recvfrom(int   s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

函数 recvfrom 默认以阻塞方式读取数据, 成功时返回接收数据的字节长度. 成功时返回 0, 否则返回-1 并置 errno.

UDP协议的基础编程模型

UPD 是无连接的通信协议, 也采用客户机-服务器模式, 几个关键步骤如下:

(1) 创建 UDP 服务器套接字

在 UDP 通信模型中,服务器端进程需要完成创建套接字, 命名套接字, 等操作后才能调用接收客户端的数据, 这里整合成一个函数 CreateUdpSock.

```
int CreateUdpSock(int *pnSock, int nPort){
        struct sockaddr_in addrin;
        struct sockaddr *paddr = (struct sockaddr *)&addrin;

        assert(pnSock != NULL && nPort > 0);
        memset(&addrin, 0, sizeof(addrin));
        addrin.sin_family = AF_INET;
        addrin.sin_addr.s_addr = htonl(INADDR_ANY);
        addrin.sin_port = htons(nPort);

        assert((*pnSock = socket(AF_INET, SOCK_DGRAM, 0)) > 0);
        if(bind(*pnSock, paddr, sizeof(addrin)) >= 0)
                return 0;
        close(*pnSock);

        return 1;
}
```

其中, 参数 pSock 回传创建的侦听套接字描述符, 整型 nPort 指定了套接字的侦听端口. 调用成功时返回 0, 否则关闭套接字并返回其他值.

(2)发送 UDP 协议数据

这里设计了一个封装了 sendto 系统调用的函数 SendMsgByUdp.

```
int SendMsgByUdp(void *pMsg, int nSize, char *szAddr, int nPort){
        int nSock, ret;
        struct sockaddr_in addrin;

        assert(pMsg != NULL && nSize > 0 && szAddr != NULL && nPort > 0);

        assert((nSock = socket(AF_INET, SOCK_DGRAM, 0)) > 0);
        memset(&addrin, 0, sizeof(struct sockaddr));
        addrin.sin_family = AF_INET;
        addrin.sin_addr.s_addr = inet_addr(szAddr);
        addrin.sin_port = htons(nPort);
        if(sendto(nSock, pMsg, nSize, 0, (struct sockaddr *)&addrin, sizeof(addrin)) > 0)
                ret = 0;
        else
                ret = 1;
        close(nSock);

        return(ret);
}
```

该函数向目的地址发送 UDP 数据包, 其中指针 pMsg 指向待发送的数据, 该数据长 nSize 字节, 目的地址的 IP 是 szAddr, 目的端口是 nPort. 调用成功时返回 0, 否则返回其他值.

(3) 接收 UDP 协议数据

这里同样封装了系统调用 recvfrom, 函数名为 RecvMsgByUdp.

```
int RecvMsgByUdp(int nFile, void *pData, int *pnSize){
        int nSize;
```

```
        assert(nFile > 0);
        if((*pnSize = recvfrom(nFile, pData, *pnSize, 0, NULL, NULL)) > 0)
                return 0;
        else
                return 1;
}
```

其中, nFile 是服务器端的 UDP 套接字描述符, 指针 pData 指向接收数据的内存缓冲区, 参数 pnSize 是该缓冲区可容纳的最大容量. 调用成功时返回 0, 并将接收到的数据存储在 pData 中, 参数 pnSize 回传接收到的字节长度; 否则返回其他值.

(4) 实例

这里设计一个基于 UDP 的客户端-服务器端的例子. 客户端进程每隔一秒向服务器发送数据报信息. 服务器进程先创建端口号为 9999 的 UDP 套接字, 然后循环接收数据报信息并打印在屏幕上.

这里, 为了编程方便我们定义了一个头文件 udp.h 封装了上面三个函数.

```
[bill@billstone Unix_study]$ cat udp.h
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

int CreateUdpSock(int *pnSock, int nPort);
int SendMsgByUdp(void *pMsg, int nSize, char *szAddr, int nPort);
int RecvMsgByUdp(int nFile, void *pData, int *pnSize);
[bill@billstone Unix_study]$
```

UDP 客户端程序如下

```
[bill@billstone Unix_study]$ cat udpk1.c
#include "udp.h"

int main(void)
{
        int ret, i = 0;
        char szBuf[100];

        while(1){
                sprintf(szBuf, "第%d 次发送", i);
                ret = SendMsgByUdp(szBuf, strlen(szBuf), "127.0.0.1", 9999);
                if(ret == 0)
                        printf("Send UDP Success: %s\n", szBuf);
                else
                        printf("Send UDP Failed: %s\n", szBuf);

                sleep(1);
                i++;
        }
}
[bill@billstone Unix_study]$
```

UDP 服务器端程序如下

```
[bill@billstone Unix_study]$ cat udps1.c
#include "udp.h"

int main(void)
{
	int nSock, nSize;
	char szBuf[256];

	CreateUdpSock(&nSock, 9999);
	nSize = sizeof(szBuf);
	memset(szBuf, 0, sizeof(szBuf));
	while(RecvMsgByUdp(nSock, szBuf, &nSize) == 0){
		printf("Recv UDP Data: [%d] [%s]\n", strlen(szBuf), szBuf);
		nSize = sizeof(szBuf);
		memset(szBuf, 0, sizeof(szBuf));
	}

	close(nSock);

	return 0;
}
[bill@billstone Unix_study]$
```

首先运行客户端程序 udpk1

```
[bill@billstone Unix_study]$ gcc -o udpk1 udpk1.c udp.c
[bill@billstone Unix_study]$ ./udpk1
Send UDP Success: 第 0 次发送
Send UDP Success: 第 1 次发送
Send UDP Success: 第 2 次发送
Send UDP Success: 第 3 次发送
Send UDP Success: 第 4 次发送
Send UDP Success: 第 5 次发送
Send UDP Success: 第 6 次发送
Send UDP Success: 第 7 次发送
Send UDP Success: 第 8 次发送
Send UDP Success: 第 9 次发送
```

接着在另一个终端运行服务器程序 udps1

```
[bill@billstone Unix_study]$ gcc -o udps1 udps1.c udp.c
[bill@billstone Unix_study]$ ./udps1
Recv UDP Data: [9] [第 0 次发送]
Recv UDP Data: [9] [第 1 次发送]
Recv UDP Data: [9] [第 2 次发送]
Recv UDP Data: [9] [第 3 次发送]
Recv UDP Data: [9] [第 4 次发送]
Recv UDP Data: [9] [第 5 次发送]
Recv UDP Data: [9] [第 6 次发送]
Recv UDP Data: [9] [第 7 次发送]
Recv UDP Data: [9] [第 8 次发送]
```

第十七章 并发Socket程序设计

非阻塞并发模型

I/O 阻塞是影响进程并发的重要原因, 进程一旦进入阻塞, 就不能再执行任何操作. 比如进程调用输入函数后, 在默认情况下必须一直阻塞到产生满足条件的数据为止.

套接字也使一种 I/O 设备, 它有四类阻塞性交易, 分别是输入类交易(read, recv 和 recvfrom 等), 输出类交易(write 和 send 等), 连接申请交易(connect)和连接处理交易(accept), 其中 UDP 协议数据发送函数 sendto 不产生阻塞.

当套接字调用以上函数时, 将导致进程阻塞. 因此把套接字函数的阻塞模式更改为非阻塞模式, 是实现并发套接字程序的一种方法.

(1) 非阻塞套接字系统调用

函数 fcntl 设置套接字描述符的 O_NONBLOCK 标志后, 即可将 I/O 方式更改为非阻塞方式.

下面代码将套接字描述符 nSock 设置为非阻塞方式.

```
int val;
val = fcntl(nSock, F_GETFL, 0);
fcntl(nSock, F_SETFL, val | O_NONBLOCK);
```

此时, 当调用函数的成功条件不满足时, 将立即返回-1 并置 errno 为 EAGAIN 或 EINPROGRESS 错误.

(2) 非阻塞套接字程序设计流程

非阻塞套接字的程序一般包含一段循环代码, 在相互中采取轮询的方式, 分别调用套接字的输入, 输出, 申请连接或连接处理函数, 从而得到并发处理多个套接字的目的.

这里设计一个运行在非阻塞方式下的服务器端套接字程序, 说明通过非阻塞方式实现并发处理的流程. 它首先在端口 PORT 上创建一个 TCP 侦听套接字, 然后一边处理客户端的套接字连接申请, 一边从已连接的客户端接收数据信息.

```
   /*------------------- 定义设置非阻塞模式宏 --------------------*/
#define Fsetnonblock(a) \
                { val = fcntl(a, F_GETFL, 0); fcntl(nSock, F_SETFL, val | O_NONBLOCK); }

int nLisSock;
int i, n = 0, nSockVar[MAX];
int val;
char buf[1024];

   /*------------------- 主流程 --------------------*/
CreateSock(&nLisSock, PORT, MAX);
Fsetnonblock(nLisSock);
   /*--- 循环代码, 论询 nLisSock 套接字的 accept 函数和连接套接字的 read 函数 ---*/
while(1){
        /*--- 创建套接字描述符 nSockVar[n]与客户端套接字建立连接 ---*/
    if((nSockVar[n] = accept(nLisSock, NULL, NULL)) > 0){
            /*--- 设置新创建的套接字描述符的非阻塞属性 ---*/
        Fsetnonblock(nSockVar[n++]);
    }
       /*--- 遍历每一个套接字, 并从客户端套接字中读入数据 ---*/
    for(i=0;i<n;i++){
        read(nSockVar[i], buf, sizeof(buf));
```

```
            /*--- 其他的处理代码---*/
        ... ... ... ...
    }
}
```

从上面可以看到, 进程虽然能够同时处理一个套接字的侦听和 n 个套接字的数据接收操作, 却存在两个缺陷:

a. 此代码采用了非阻塞轮询的方式, 极大地浪费了 CPU 时间.

b. 此代码没有解决循环的跳出问题, 进程很有可能一直循环下去.

信号驱动并发模型

有过 Windows 下开发经验的读者都知道消息驱动的概念. 比如使用 Win32 程序, 平时挂起或执行其他的操作, 当套接字有输入发生时, Windows 产生消息并将此消息发送到目标进程. 进程接到此消息, 自动调用响应的处理代码, 完成套接字数据的读取过程.

当文件描述符就绪时, UNIX 内核会向进程发送 SIGIO 信号, 进程获取该信号并调用预先准备的处理函数执行 I/O 操作. 当函数调用完毕后, 进程回到接收信号前的代码处继续工作. 在一个套接字上实现信号驱动的步骤如下:

(1) 为信号 SIGIO 设计处理函数被捕获信号

```
Void func(int sig){                    // 信号处理函数
      ... ... ... ...
    signal(SIGIO, func);
}
... ... ... ...
signal(SIGIO, func);                   // 捕获信号 SIGIO
```

(2) 设定套接字的归属主, 设置接收 SIGIO 信号的进程或进程组.

```
Fcntl(nSock, F_SETOWN, getid());            // 设置接收信号的进程
或者
fcntl(nSock, F_SETOWN, 0 – getpgrp());   // 设置接收信号的进程组
```

(3) 设置套接字的异步 I/O 属性.

```
int val;
val = fcntl(nSock, F_GETFL, 0);
fcntl(nSock, F_SETFL, val | O_ASYNC);
```

然而, 套接字信号驱动的设置尽管比较简单, 但在实践中却难以实现. 因为 UNIX 内核并不仅仅在套接字有输入时才发送 SIGIO 信号, 在套接字连接, 输出, 错误和其他状态变化时均发送该信号, 这样就导致了进程在接收信号后无法正确地判断下一步的行为.

超时并发模型

超时也是防止阻塞的一种手段, 它可以保证进程不被永远挂起. 阻塞函数的超时时间长度决定了进程的并发程度, 超时时间越小, 并发度越高, 超时时间越大, 则并发度越低.

在 UNIX 下的阻塞函数一般都有在进程接收到任意信号后终端返回的传统, 套接字函数也不例外, 利用这个特性使用定时器信号 SIGALRM 实现套接字的超时处理.

下面我们使用信号加跳转方式设置超时:

(1) 定义超时标志和跳转结构.

(2) 为信号 SIGALRM 设计处理函数

(3) 记录跳转点
(4) 超时判断
(5) 捕获信号 SIGALRM 并设置定时器
(6) 调用套接字阻塞处理函数, 比如套接字接收, 发送, 连接申请或者接收连接等
(7) 取消定时器并忽略信号 SIGALRM.

本处设计一个套接字连接函数 connect 超时处理的例子 tcpto1.c, 它从命令行输入 IP 地址和端口, 程序向该 IP 地址和端口建立连接, 如果连接失败, 10 秒钟后超时退出, 代码如下

```
#include "tcp.h"

static int nTimeout = 0;                        // (1) 定义超时标志设置
jmp_buf env;                                    // (1) 定义跳转结构

void OnTimeOut(int nSignal){                    // (2) 信号处理函数
        signal(nSignal, SIG_IGN);
        nTimeout = 1;
        longjmp(env, 1);
        return;
}

int main(int argc, char *argv[])
{
        int nSock = -1, ret;

        if(argc != 3)
                return 1;
        nTimeout = 0;
        setjmp(env);                            // (3) 记录跳转点
        if(nTimeout == 1)                       // (4) 超时判断
                printf("Connect Timeout.\n");
        else{
                signal(SIGALRM, OnTimeOut);     // (5) 捕获信号 SIGALRM
                alarm(3);                       // (5) 发送定时器信号 SIGALRM
                ret = ConnectSock(&nSock, atoi(argv[2]), argv[1]);    // (6) 执行函数
                alarm(0);                       // (7) 取消定时器
                signal(SIGALRM, SIG_IGN);       // (7) 忽略信号 SIGALRM
                if(ret == 0)
                        printf("Connect Success.\n");
                else
                        printf("Connect Error.\n");
        }
        if(nSock != -1)
                close(nSock);

        return 0;
}
```

其中, 用到了一个头文件 tcp.h, 其中整合了 ConnectSock 函数, 代码如下:

```
[root@billstone Unix_study]# cat tcp.h
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <assert.h>

int ConnectSock(int *pSock, int nPort, char *pAddr){
        struct sockaddr_in addrin;
        long lAddr;
        int nSock;

        assert(pSock != NULL && nPort > 0 && pAddr != NULL);
        assert((nSock = socket(AF_INET, SOCK_STREAM, 0)) > 0);
        memset(&addrin, 0, sizeof(addrin));

        addrin.sin_family = AF_INET;
        addrin.sin_addr.s_addr = inet_addr(pAddr);
        addrin.sin_port = htons(nPort);
        if(connect(nSock, (struct sockaddr*)&addrin, sizeof(addrin)) >= 0){
                *pSock = nSock;
                return 0;
        }
        else{
                close(nSock);
                return 1;
        }
}

[root@billstone Unix_study]#
```

编译程序, 分别连接百度网站上的已打开和未打开的端口, 结果如下

```
[root@billstone Unix_study]# gcc -o tcpto1 tcpto1.c
[root@billstone Unix_study]# ping www.baidu.com
PING www.a.shifen.com (202.108.22.5) 56(84) bytes of data.

--- www.a.shifen.com ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

[root@billstone Unix_study]# ./tcpto1 202.108.22.5 80              // 连接存在的 80 HTTP 端口
Connect Success.
[root@billstone Unix_study]# ./tcpto1 202.108.22.5 1234            // 连接不存在的 1234 端口
Connect Timeout.                                                   // 延迟 3 秒
[root@billstone Unix_study]#
```

多路复用并发模型

多路复用函数 select 把一些文件描述符集合在一起, 如果某个文件描述符的状态发生编号, 比如进入"写就绪"或者"读就绪"状态, 函数 select 会立即返回, 并且通知进程读取或写入数据; 如果没有 I/O 到达, 进程将进入阻塞, 知道函数 select 超时退出为止.

利用多路复用, 进程可以同时监控多个套接字信息, 在多个套接字上并发地执行操作.

几种常见的使用 select 的套接字进程如下:

(1) 交互式进程

程序一边处理客户的交互式输入输出, 一边使用套接字. 多路复用标准输入, 标准输出和套接字文件描述符.

(2) 多套接字进程

程序同时使用侦听套接字和大量的连接套接字.

(3) 多协议进程

程序同时使用 TCP 套接字和 UDP 套接字

(4) 多服务进程

程序同时应用多种服务, 完成多种应用协议, 比如 inetd 守护进程等.

下面以多路复用读套接字文件描述符为例, 说明并发程序设计的步骤:

(1) 创建套接字文件描述符集合

```
fd_set fdset;                       // 文件描述符集合
FD_ZERO(&fdset);                    // 清空集合中的元素
FD_SET(nLisSock, &fdset);           // 监控侦听套接字
FD_SET(nSockVal, &fdset);           // 监控连接套接字
```

(2) 准备超时时间

```
Struct timeval wait;                // 定义超时时间
wait.tv_sec = 0;                    // 超时时间为 0.1 秒
wait.tv_usec = 100000;
```

(3) 调用函数 select 并检测应答结果

假设只复用读集合中的套接字描述符, 其中 MAXSOCK 是集合 fdset 中最大的描述符号.

```
Int ret;
ret = select(MAXSOCK+1, &fdset, NULL, NULL, &wait);
if (ret == 0)   ... ... ... ...                    // 超时
else if (ret == -1)   ... ... ... ...              // 错误
else ... ... ... ...                               // 产生了套接字连接或数据发送请求
```

(4)检测套接字文件描述符的状态并处理之

如果是侦听套接字, 其操作流程一般为:

```
If (FS_ISSET(nLisSock, &fdset)) {                  // 侦听套接字, 处理连接申请
    if ((nSockVal = accept(nLisSock, NULL, NULL)) > 0)   ... ... ... ...
}
```

如果是连接套接字操作, 其流程一般为:

```
If (FS_ISSET(nLisSock, &fdset)) {                  // 连接套接字, 读取传输数据
    read (nSockVar[i], buf, sizeof(buf));
}
```

采用 select 实现套接字的并发处理, 有如下优点:

(1) 在监控套接字描述符状态变化的过程中, 函数 select 以阻塞的方式执行, 这样可以节省 CPU 时间.

(2) 当规定的时间到达后, 套接字仍然没有连接申请, 接收或发送, 函数 select 将自动返回, 这样可以预防进程一直阻塞下去.

(3) 函数 select 能够同时监控多个套接字描述符的状态, 实现套接字的并发处理.

多进程并发模型

服务器套接字进程经常既需要接收客户端的连接申请, 又要接收客户端发送的数据信息, 遗憾的是函数 accept 和函数 recv 或 read 都会引起进程阻塞, 服务器宽进程常常顾此失彼.

多进程方法正好可以弥补这个缺陷, 它创建专门的进程来处理每一个阻塞的套接字函数, 比如父进程只执行函数 accept 等待并完成客户端的连接申请, 子进程则执行函数 recv 等待客户端的信息发送. 虽然每个进程都处于阻塞状态, 但一旦某个套接字描述符的状态发送变化时, 它所在的进程都能在第一时间被激活并完成响应操作, 这样救灾整个进程组中实现了套接字的并发处理.

(1) 不固定进程数的并发模型

多进程实现套接字并发处理最常见的方式是 accept 后创建子进程, 父进程继续 accept, 子进程完成后续工作.

不固定进程数的并发模型的服务器端流程如下:

a. 创建侦听套接字 nLisSock (socket, bind 和 listen)

b. 进程转后台运行

c. 等待客户端的连接申请(accept), 创建与客户端的通信连接 nSock

d. 创建子进程

e. 父进程关闭套接字 nSock, 此时用于子进程仍然打开此套接字, 故父进程的关闭操作并不真正断开连接, 只是把连接上减少 1

f. 父进程回到步骤 c 继续进行

g. 子进程关闭侦听套接字 nLisSock, 用于父进程仍然打开着侦听套接字, 故实际上此套接字仅仅把连接数减少 1, 并不真正关闭它

h. 子进程执行 recv 和 send 等操作与客户端进行数据交换

i. 数据交换完毕, 关闭套接字, 子进程结束

(2) 固定进程数的并发模型

固定进程数的并发模型是一种介于单进程与多进程之间的这种方案, 服务器父进程在创建侦听套接字后 fork 子进程, 由子进程等待客户端的 connect 并完成与客户端的通信交换等工作, 父进程的功能只是维持子进程的数量不变.

1) 父进程流程

固定进程数并发模型的服务器端父进程的流程如下:

a. 创建侦听套接字 nLisSock

b. 进程转后台运行

c. 创建子进程

d. wait 子进程. 如果有子进程退出, 则立即创建子进程, 保证子进程在数量上不变.

2) 子进程流程

子进程继续父进程的侦听套接字, 按下面流程运行:

a. 等待客户端的连接申请(accept), 并与客户端建立通信套接字 nSock 连接

b. 执行 recv 和 send 等操作与客户端进行数据交换

c. 数据交换完毕, 关闭套接字 nSock

d. 回到步骤 a, 继续执行