UNIX/Linux C 简要复习

陈龙

目录

一、 库的创建和使用 ...... - 1 -

1. 静态库 ...... - 1 -
2. 动态库 ...... - 1 -

二、 标准 IO ...... - 3 -

1. 创建、打开、关闭与删除函数族 ...... - 3 -
2. 文件读写函数族 ...... - 4 -
3. 文件读写位置定位 ...... - 4 -
4. 文件的状态 ...... - 5 -
5. 文件的缓冲 ...... - 5 -

三、 低级 IO ...... - 6 -

1. 文件打开、关闭、删除函数族 ...... - 6 -
2. 文件读写函数族 ...... - 6 -
3. 文件定位 ...... - 6 -
4. 文件缓冲 ...... - 6 -
5. 复制文件描述符 ...... - 6 -
6. 文件控制 FCNTL ...... - 6 -

四、 目录编程 ...... - 6 -

1. 目录的读取、定位 ...... - 6 -
2. 目录的创建、删除 ...... - 7 -
3. 工作目录 ...... - 7 -

五、 进程控制 ...... - 8 -

1. 进程标识符 ...... - 8 -
2. FORK、EXEC ...... - 8 -
3. 进程的休眠、终止、同步 ...... - 9 -

六、 信号 ...... - 10 -

1. 信号函数 ...... - 10 -
2. 显示发送信号 ...... - 10 -
3. 定时器 ...... - 10 -

七、 进程间通讯之——管道 - 10 -

1. 无名管道 - 10 -
2. POPEN - 10 -
3. 命名管道 FIFO - 11 -

八、 进程间通讯之——消息队列 - 11 -

1. 消息队列数据结构 - 12 -
2. 消息队列的创建 - 13 -
3. 消息队列的发送和接收 - 13 -
4. 消息队列的控制 - 14 -

九、 进程间通讯之——信号量 - 14 -

1. 信号量的基本操作 - 14 -
2. 信号量数据结构 - 15 -
3. 信号量的创建 - 15 -
4. 信号量控制 - 15 -
5. P/V/Z 操作 - 15 -

十、 进程间通讯之——共享内存 - 18 -

6. 共享内存数据结构 - 18 -
7. 共享内存的创建 - 18 -
8. 映射、释放共享内存 - 18 -
9. 共享内存控制 - 20 -

十一、 进程间通讯之——SOCKET - 20 -

1. SOCKET 地址结构 - 20 -
2. IP 地址转换 - 21 -
3. 字节序转换 - 21 -
4. SOCKET 端口 - 21 -
5. TCP C/S 通讯结构 - 21 -
6. UDP C/S 通讯结构 - 25 -

UNIX/Linux C 简要复习

一、 库的创建和使用

1. 静态库

```
/* pr1.c */
#include <stdio.h>

void print1() {
     printf("This is the first lib src!\n");
}
```

```
/* pr2.c */
#include <stdio.h>

void print2() {
     printf("This is the second lib src!\n");
}
```

```
/* main.c */
#include <stdio.h>

int main(void)
{
     print1();
     print2();
     return 0;
}
```

```
$ cc -O -o pr1.c pr2.c

$ ar -crsv libpr.a pr1.o pr2.o

$ cc -O -o main main.c -L./ -lpr

$ ./main
```

2. 动态库

1) 创建动态库

```
/* d1.c */
#include <stdio.h>

int  p = 2;
void print() {
     printf("This is the first dll src!\n");
}
```

```
/* d2.c */
#include <stdio.h>

int p = 3;
void print() {
     printf("This is the second dll src!\n");
}
```

```
/* td1.c */
#include <stdio.h>

int main(void)
{
     print();
     return 0;
}
```

动态库的编译可分为三个步骤，设计源代码、编译位置无关的代码（PIC）型.o 文件和链接动态库。链接动态库的命令包含特殊标志，与链接静态库和链接可执行目标文件有区别，而且不同的 UNIX 系统，其实现的细节也不相同。

编译 PIC 型.o 中间文件的方法一般是采用 C 编译器的“-KPIC”或者“-fpic”选项，有的 UNIX 版本的 C 编译器默认带上了 PIC 标志。创建最终动态库的方法一般是采用 C 编译器的“-G”或“-shared”选项，或者使用工具 ld 创建。

Linux 上编译以上代码：

分步编译：

```
$ gcc -fpic -c d1.c d2.c
$ gcc -shared -o d1.so d1.o
$ gcc -shared -o d2.so d2.o
```

一步到位：

```
$ cc -O -fpic -shared -o d1.so d1.c
$ cc -O -fpic -shared -o d2.so d2.c
```

2) 动态库的使用

```
$ cp d1.so dll.so
$ gcc -O -o td1 td2.c ./dll.so
```

3) 动态库的查找

通过环境变量 LD_LIBRARY_PATH

4) 动态库的显示调用

打开动态库：

```
#include <dlfcn.h>
void *dlopen(const char *pathname, int mode);
```

获取动态库对象地址：

```
#include <dlfcn.h>
void *dlsym(void *handle, const char *name);
```

关闭动态库：

```
#include <dlfcn.h>
int dlclose(void *handle);
```

错误检查：

```
#include <dlfcn.h>
char *dlerror(void);
```

5) 实例

获取动态库中的函数：

```
void (*pfunc)();
if (pfunc = (void (*)())dlsym(pHandle, “print”)) {
```

```
        pfunc();
}
```

获取动态库中的变量：

```
int *p;
if (p = (int *)dlsym(pHandle, “p”)) {
        printf(“p=%d”, p);
}
```

完整实例：

```
/* td1.c  */
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(void)
{
     void *phandle;
     void (*pfunc)();
     int *p;
     if (!(phandle = dlopen("./dll.so", RTLD_NOW))) {
         printf("Cannot find dll.so\n");
         exit(1);
     }
     if ((pfunc = (void (*)())dlsym(phandle, "print")) != NULL) {
         pfunc();
     } else {
         printf("cannot find function print\n");
     }

     if ((p = (int *)dlsym(phandle, "p")) != NULL) {
         printf("p=%d\n", *p);
     } else {
         printf("cannot find variable p\n");
     }

     dlclose(phandle);
     return 0;
}
```

```
$ cc -O -o td1 td1.c -ldl

$ ./td1
```

二、 标准 IO

1. 创建、打开、关闭与删除函数族

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *type);
FILE *freopen(const char *filename, const char *type, FILE *stream);
int fclose(FILE *stream);
int remove(const char *filename);
int rename(const char *oldname, ocnst char *newname);
```

- freopen 实现文件流的替换，它首先关闭原文件流 stream，然后再以 fopen 的方式发开一个新的文件流，此后对原文件流的任意操作都自动转化为对新文件流的操作。成功时返回值先新文件的 FILE 型指针，否则返回 NULL。

实例

```
if ((fp = freopen(“/tmp/1”, “w”, stderr)) == NULL) {
        printf(“stderr == /tmp/1 failed.\n”);
        return ;
}
```

2. 文件读写函数族

字符输入输出函数族

```
#include <stdio.h>
int getc(FILE *stream);
int getchar(*void);
int fgetc(FILE *stream);

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

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);
```

- gets 从标准输入流（stdin）中读取一字符存储到参数 s 所指向的内存空间中，当文件结束或者错误发生时返回 NULL，否则将返回参数 s 所指向的内存地址。fgets 加入防溢出控制。
- puts 参数 s 只想一串以字符串结束符“0”结尾的字符，将该字符（不包括“0”）写到 stdout，并自动输出“\n”。不输出字符串的结尾符，失败是返回 EOF。

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);
```

格式化输入输出函数族

```
#include <stdio.h>
int printf(const char *fromat, /* [arg,] */ …);
int fprintf(FILE **stream, const char *fromat, /* [arg,] */ …);
int sprintf(char *s, const char *fromat, /* [arg,] */ …);
int scanf(const char *format, /* [pointer,] */ …);
int fscanf(FILE *stream, const char *format, /* [pointer,] */ …);
int sscanf(const char *s, const char *format, /* [pointer,] */ …);
```

3. 文件读写位置定位

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
void rewind(FILE *stream);
long int ftell(FILE *stream);
```

- noffset：

  SEEK_SET　　文件头开始

  SEEK_CUR　　当前位置开始

  SEEK_END　　文件未开始
- rewind 重置 stream，将文件流定位于文件开始处。

- ftell 获取文件流的当前访问位置，调用成功时将该值返回，否则返回-1。

4. 文件的状态

```
#include <stdio.h>
int ferror(FILE *stream);
int feof(FILE *stream);
void clearer(FILE *stream);

extern int errno;
#include <string.h>
char *strerror(int errnum);
```

- 当文件 I/O 发生错误时，调用 ferror 函数将返回非0值，否则返回0值。
- 文件结束时（读文件函数返回 EOF），feof 返回非0值，否则返回0 。
- clearer 清楚文件流错误标志和 EOF 标志。
- 在错误发生后立马调用 strerror(errno)可以获得当前的错误提示信息。

实例

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

int main(void)
{
     FILE *fp;
     char str[1024];

     if ((fp = fopen("/etc/passwd", "r")) == NULL) {
         perror("open file passwd ERROR !\n");
         exit(-2);
     }
     while (!feof(fp)) {
         fgets(str, 1024, fp);
         fputs(str, stdout);
     }
     return 0;
}
```

5. 文件的缓冲

标准 IO 函数和低级 IO 函数相比就是，标准 IO 在应用级增加了缓冲功能（低级 IO 函数只能使用文件系统自带的缓冲功能，可视为无缓冲），标准 IO 的缓冲模式分为三种：

- 全缓冲　　读写普通磁盘文件采用全缓存模式
- 行缓冲　　**stdin、stdout**
- 无缓冲　　**stderr**

缓冲函数

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
int fflush(FILE *stream);
```

- setbuf 设置 stream 的缓冲区为 buf，若 buf 为 NULL 则关闭 stream 的缓冲。
- setvbuf 可设置缓冲类型：_IOFBF 全缓冲，_IOLBF 行缓冲，_IONBF 无缓冲。

实例

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

```
int main(void)
{
     printf(" 1---1 ");
     fprintf(stderr, " 2---2 ");
     printf(" 3---3 \n");
     fprintf(stderr, " 4---4\n");
     return 0;
}
```

三、 低级 IO

1. 文件打开、关闭、删除函数族

```
#include <fcntl.h>
int open(const char *filename, int oflag, … /* [mode_t mode] */);

#include <unistd.h>
int close(int fildes);
int unlink(char *path);
```

2. 文件读写函数族

```
#include <unistd.h>
ssize_t read(int filedes, void *buff, size_t nbyte);
ssize_t write(int filedes, const void *buf, size_t nbytes);
```

3. 文件定位

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

4. 文件缓冲

```
#include <unistd.h>
int fsync(int fildes);
```

5. 复制文件描述符

```
#include <unistd.h>
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

6. 文件控制 fcntl

```
#include <fcntl.h>
int fcntl(int fildes, int cmd);
int fcntl(int fildes, int cmd, int arg);
int fcntl(int fildes, int cmd, struct flock *arg);
```

- cmd 为 F_DUPFD、F_GETFD/F_SETFD、F_GETFL/F_SETFL、F_GETOWN/F_SETOWN、F_GETLK/F_SETLK/F_SETLKW

四、 目录编程

1. 目录的读取、定位

```
#include <dirent.h>
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

```
void seekdir(DIR *dirp, long int loc);
void rewinddir(DIR *dirp);
long int telldir(DIR *dirp);
```

- opendir 返回一个目录流，类似 FILD。
- readdir 通过流读 dirp 指向的目录流，返回目录中的一项内容在 dirent 只想的结构中，并移动目录文件指针到下一目录项。ditent 结构一般包含以下两项

  ino_t d_ino; /* 文件对性的 i 节点编号 */

  chard_name[]; /* 文件名称（以字符串结尾符“0”结尾） */
- seekdir 不可随意定位，loc 必须是前面对同一目录流 dirp 调用函数 telldir 的返回值之一。
- telldir 返回目录流的当前访问位置。

实例

```
long int lo;
DIR *dirp;
…
lo = telldir(dirp);
…
seekdir(dirp, lo);
```

2. 目录的创建、删除

```
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
int rmdir(char *path);
```

- mkdir 的目录的权限由操作系统的 mask 和 mode 共同决定，为 mkde & (~mask)。

3. 工作目录

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

int chdir(const char *path);
int fchdir(int fildes);
```

- getwd 拷贝当前工作目录绝对地址到 pathname 处，最大长度为 PATH_MAX。

实例

```
/* ls1.c */
#include <stdio.h>
#include <string.h>
#include <dirent.h>

inline int opshowdir(char *pathname) {
     DIR *dirp;
     struct dirent *pent;
     if ((dirp = opendir(pathname)) == NULL) {
         perror("open dir ERROR \n");
         return 1;
     }
     while (1) {
         if ((pent = readdir(dirp)) == NULL)
              break;
         fprintf(stderr, "%5d %s\n", pent->d_ino, pent->d_name);
```

```
     }
     closedir(dirp);
     return 0;
}

int main(int argc, char *argv[])
{
     char pathstr[1024];
     int i = 1;

     if (argc == 1) {
         strcpy(pathstr, "./");
         opshowdir(pathstr);
     } else if (argc > 1) {
         while (i < argc) {
              opshowdir(argv[i]);
              ++i;
         }
     }
     return 0;
}
```

五、 进程控制

1. 进程标识符

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid();
pid_t getpgrp();
pid_t getppid();

uid_t getuid();
uid_t geteuid();
gid_t getgid();
gid_t getegid();
```

- getpid 返回当前进程 id，getpgrp 返回当前进程组 id，getppid 返回父进程 id。
- getuid 返回进程实际用户 id，geteuid 返回进程有效用户 id，getgid 返回进程实际组 id。

实例

```
/* id1.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
     printf("pid=[%d], gid=[%d], ppid=[%d]\n", getpid(), getpgrp(), getp-
pid());
     printf("uid=[%d], euid=[%d], gid=[%d], egid=[%d]\n", getuid(), geteu-
id(), getgid(), getegid());
     return 0;
}
```

2. fork、exec

```
#include <unistd.h>
pid_t fork();
pid_t vfork();
int execl(const char *path, const char *arg0, …, (char *)0);
int execle(const char *path, const char *arg0, …, (char *)0, char *const
evnp[]);
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[], char *const evnp[]);
int execvp(const char *file, const char *argv[]);
extern char **environ;
```

- vfork 和 fork 有两点区别：①创建子进程时不复制父进程数据，在 exec 后才复制父进程数据。②父进程以 vfork 方式创建子进程之后被阻塞，知道子进程退出或执行 exec 调用后才继续运行。

3. 进程的休眠、终止、同步

```
#include <unistd.h>
unsigned int sleep(unsigned int second);

#include <stdlib.h>
void exit(int status);

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

实例

父进程监控子进程并在任意子进程退出时重启所有子进程。

```
/* wait2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

int main(void)
{
     pid_t pid1, pid2, pid3;
     int status;
     while (1) {
         if ((pid1 = fork()) < 0) {
              perror("fork ERROR\n");
              exit(1);
         } else if (pid1 == 0) {
              sleep(30);
              exit(0);
         }

         if ((pid2 = fork()) < 0) {
              perror("fork ERROR\n");
              exit(1);
         } else if (pid2 == 0) {
              sleep(40);
              exit(0);
         }
         fprintf(stderr, "Fork child pid=[%d][%d]\n", pid1, pid2);
         pid3 = wait(&status);
         kill(pid1, SIGTERM);
         kill(pid2, SIGTERM);
         fprintf(stderr, "Kill child pid=[%d][%d], exitpid[%d], sta-
tus[%d]\n", pid1, pid2, pid3, status);
         pid3 = wait(&status);
         sleep(1);
     }
     return 0;
```

```
}
```

六、 信号

1. 信号函数

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

- f 的取值：①SIG_DEG 默认 ②SIG_IGN 忽略信号处理 ③函数地址
- 该函数可简化为

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

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

2. 显示发送信号

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

3. 定时器

```
#include <unistd.h>
unsigned int alarm(unsigned int second)
```

- 进程将在调用函数 alarm 后 seconds 秒，接收到内核发送的一个 SIGALRM 信号，该函数只产生一次定时操作，如需多次定时需要在信号处理函数中重新设定。

七、 进程间通讯之——管道

1. 无名管道

```
#include <unistd.h>
int pipe(int fildes[2]);
```

- fildes 为管道的两端，fildes[0]为入端，fildes[1]为出端。管道为单向数据流。
- 双向管道需要建立两个无名管道。

实例

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

2. popen

```
#include <stdio.h>
FILE *popen(const char *command, char *type);
int pclose(FILE *stream);
```

- popen 类似函数 system，先 fork 一个子进程，然后调用 exec 执行参数 command 中给定的 shell 命令。
- type 参数：r 创建于子进程的标准输出连接的管道。w 创建与子进程的标准输入连接的管道。

实例

模拟命令 ps –ef | grep init。

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

```
int main(void)
{
     FILE *out, *in;
     char buf[1024];
     if ((out = popen("grep init", "w")) == NULL) { /* 创建写管道流 */
         fprintf(stderr, "error!\n");
         exit(-1);
     }
     if ((in = popen("ps -ef", "r")) == NULL) {  /* 创建读管道流  */
         fprintf(stderr, "error!\n");
         exit(-1);
     }
     while (fgets(buf, sizeof buf, in) != NULL) { /* 读取ps -ef的结果  */
         fputs(buf, out);  /* 转发到grep init  */
     }
     pclose(out);
     pclose(in);
     return 0;
}
```

3. 命名管道 FIFO

1) 创建命名管道 FIFO

a) 命令 mknode：mknod name p (管道名为 name)

mknod name [b | c] major minor 创建块设备或字符设备文件

mknod name p 创建管道文件

mknod name s 创建信号量

mknod name m 创建共享内存

b) 命令 mkfifo：mkfifo [-m Mode] file ... Mode 为创建的管道文件的权限。

c) 系统调用

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(char *path, mode_t mode);
```

- mode 类似 open 的第三个参数。
- 管道文件本身就是文件，对普通文件的操作也适合于管道文件（read、write……）。

八、 进程间通讯之——消息队列

本章开始的三种 IPC 机制，一般称为 XSI IPC，消息队列适用于进程之间少量数据的顺序共享，信号量适用于进程之间的同步与互斥的控制，共享内存则适用于进程之间大批量数据的随机共享访问。

操作 IPC 对象：

a) 查询 IPC 对象

ipcs [options]

-q 查询消息队列 IPC 对象

-s 查询信号量 IPC 对象

-m 查询共享内存 IPC 对象

-a 查询 IPC 对象的全部属性

默认 查询消息队列、信号量、共享内存的基本属性，类似于“-qsm”。

IPC ID 代表 IPC 对象的标识符，此外每个 IPC 对象还有一个关键字（KEY），两者的关系相当于文件系统 i 节点号，和文件名。通过文件名找到此 IPC 对象 ID，再通过此 ID 访问相应的 IPC 对象。

b) 删除 IPC 对象

ipcrm [options]

-q msqid

-Q msgkey

-s semid

-S semkey

-m shmid

-M shmkey

1. 消息队列数据结构

消息队列本质上就是内核中的一个队列结构，每一个消息队列用结构 struct msqid_ds 表示，该结构中每项对应于消息队列的每个属性，其中 msg_first 和 msg_last 分别指向此消息队列中的队首和队尾，队列中每项的结构为 struct msg，一般结构如下：

```
struct msg {
    struct msg *msg_next;
    long msg_type;          /* 本消息的类型 */
    short msg_ts;               /* 本消息的长度 */
    short msg_spot;         /* 本条消息的数据地址 */
}
```

msg_spot 指向消息数据，表示消息数据的结构为 msgbuf，该结构不是固定的，而是一个大致的模板，几例如下：

```
/* 消息数据是一个字符 */
struct msgbuf {
    long mtype;     /* 消息类型 */
    char mtext[1];  /* 消息数据 */
};
```

```
/* 消息数据是一个整型数据 */
struct msgbuf {
    long mtype;
    int ntext;
};
```

```
/* 消息数据是一个字符数组和一个整型数据 */
struct msgbuf {
    long mtype;
    char ctext[100];
    int ntext;
};
```

```
/* 消息数据是一个结构 */
struct msgtext {
    char ctext[200];
    int ntext;
};
struct msgbuf {
```

```
    long mtype;
    struct msgtext stext;
};
```

2. 消息队列的创建

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- msgget 创建一个新消息队列，或访问一个已经存在的消息队列

3. 消息队列的发送和接收

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, void *msgp, int msgsz, int msgflg);
                                                        返回：成功0；出错-1
int msgrcv(int msqid, void *msgp, int msgsz, long msgtype, int msgflg);
返回：成功实际接收的消息数据字节数
```

- 进程在发送或接收消息队列的信息时接收到信号，将终止消息的发送或接收并返回 EINTR 错误，此时重新发送即可。

实例

循环读取键盘输入，将输入的字符串写入到消息队列（关键字为0x1234）中。并在接收程序中显示。

```
/* msg1.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/errno.h>
/* 定义消息结构 */
struct mymsgbuf {
     long mtype;
     char ctext[100];
};

int main(void)
{
     struct mymsgbuf buf;  /* 消息缓冲区 */
     int msgid;
     /* 打开（或创建）消息队列 */
     if ((msgid = msgget(0x1234, 0666 | IPC_CREAT)) < 0) {
         fprintf(stderr, "open msg %X failed.\n", 0x1234);
         exit(1);
     }
     /* 循环读取键盘输入，并将读取的字符串写到消息队列 */
     while (strncmp(buf.ctext, "exit", 4)) {
         memset(&buf, 0, sizeof buf);
         fgets(buf.ctext, sizeof buf.ctext, stdin);
         buf.mtype = getpid();
         while ((msgsnd(msgid, &buf, strlen(buf.ctext), 0)) < 0) {
              if (errno == EINTR) /* 信号中断，重新发送 */
                  continue;
```

```
                return;
            }
        }
        return 0;
}
```

```
/* msg2.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/errno.h>

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

int main(void)
{
        struct mymsgbuf buf;
        int msgid;
        int ret;
        if ((msgid = msgget(0x1234, 0666 | IPC_CREAT)) < 0) {
            fprintf(stderr, "open msg %X failed.\n", 0x1234);
            exit(1);
        }
        while (strncmp(buf.ctext, "exit", 4)) {
            memset(&buf, 0, sizeof buf);

            while ((ret = msgrcv(msgid, &buf, sizeof buf.ctext, 0, 0)) < 0) {
                if (errno = EINTR) continue;
                return;
            }
            fprintf(stderr, "Msg: Type=%d, Len=%d, Text:%s", buf.mtype,ret,
buf.ctext);
        }
        return 0;
}
```

4. 消息队列的控制

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

- 对消息队列执行各种控制，包括查看消息队列数据结构、改变消息队列的访问权限、改变消息队列的属主、删除消息队列等。

九、 进程间通讯之——信号量

信号量本质上是一组整型变量，用于进程之间的互斥与同步（通过 P、V、Z 操作）。

1. 信号量的基本操作

a) P 操作

检查信号量的取值，如果该值大于0，则分配临界资源，信号量减1；否则代表当前无空闲资源可用，进程阻塞，知道制定资源到达位置。

b) V 操作

释放临界资源，信号量取值加1。

c) Z 操作

又称测试操作，等待当前信号量取值为0，如果成立，进程返回。

2. 信号量数据结构

IPC 对象中的“信号量”通常指的是“信号量集合”，内核采用结构 semid_ds 来管理信号量，它的数据成员与命令“ipcs –a -s”显示的结果一一对应，该结构中有值先信号量集合中的信号量指针 sem_base 和信号量集合中信号量成员的数目 sem_nsems。sem_base 值先一个信号量数组，数组中的每项为结构 sem，如下：

```
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 */
};
```

3. 信号量的创建

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

- nsems 为创建的信号量集合中的信号量个数。
- semflg 为权限和 IPC_CREAT、IPC_EXCL。

实例

创建关键字0x1234，权限0666，10个信号量的信号量集合。

```
int semid = semget(0x1234, 10, 0666 | IPC_CREAT);
```

4. 信号量控制

读取、设置、删除等操作

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

5. P/V/Z 操作

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

- sops 指向的结构如下

```
struct sembuf {
    short sem_num;  /* 信号量中的信号序号 */
    short sem_op;       /* 信号操作 */
    short sem_flg;      /* 操作方式，取值有 IPC_NOWAIT 和 SEM_UNDO 等 */
};
```

sem_num 指定操作信号量集合中的信号序号，第一个信号的序号是0，sem_op 为操作类型，如下：

a) 正整数（>0）→V 操作，信号量数值增加 sem_op。

b) 0 →Z 操作，判断信号量数值是否等于0。

c) 负整数（<0）→ P 操作，信号量数值增加 sem_op。

实例

生产者-消费者：KEY 为1000，第0、1信号量分别为信号量 A 和 B。

信号量 A 代表当前可生产的数目，初始为最大值；信号量 B 代表当前的产品数，初始为0；

生产者进程：

P(A)

production;

V(B)

消费者进程：

P(B)

custom;

V(A)

A 信号量初始值为5，B 信号量初始值为0.

```
/* initsem1000.c */
/* 初始化A，B两个信号量，最好在生产者和消费者进程运行之前运行该初始化程序 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>

/* /usr/include/bits/sem.h中有如下注释内容 */
/* The user should define a union like the following to use it for arguments
   for `semctl'.

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

   Previous versions of this file used to define this union but this is
   incorrect.  One can test the macro _SEM_SEMUN_UNDEFINED to see whether
   one must define the union or not.  */
/* 并在该文件中定义了以下宏 */
// #define _SEM_SEMUN_UNDEFINED   1 //

#ifdef _SEM_SEMUN_UNDEFINED
union semun
{
     int val;
     struct semid_ds *buf;
     unsigned short int *array;
     struct seminfo *__buf;
};
#endif
```

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

int main(void)
{
     int semid;
     union semun sem;
     VERIFYERR((semid = semget(1000, 2, 0666|IPC_CREAT)) < 0, "open sem 
1000");
     /* Init sem A=5 */
     sem.val = 5;
     VERIFYERR(semctl(semid, 0, SETVAL, sem) == -1, "semctl set sem 
1000:0=5");
     /* Init sem B=0 */
     sem.val = 0;
     VERIFYERR(semctl(semid, 1, SETVAL, sem) == -1, "semctl set sem 
1000:1=0");
     return 0;
}
```

```
/* sema.c 生产者进程 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.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 semid;
     struct sembuf sb;
     VERIFYERR((semid = semget(1000, 2, 0666|IPC_CREAT)) < 0, "open sem 
1000"); //打开信号量

     // P(A)
     sb.sem_num = 0;
     sb.sem_op = -1;
     sb.sem_flg = sb.sem_flg & ~IPC_NOWAIT;
     VERIFYERR(semop(semid, &sb, 1) != 0, "P sem 1000:0");
     // 生产
     fprintf(stderr, "[%d]producing ... \n", getpid());
     sleep(1); //生产耗时1s
     fprintf(stderr, "[%d]producted \n", getpid());

     // 提交产品V(B)
     sb.sem_num = 1;
     sb.sem_op = 1;
     sb.sem_flg = sb.sem_flg & ~IPC_NOWAIT;
     VERIFYERR(semop(semid, &sb, 1) != 0, "V sem 1000:1");

     return 0;
}
```

```
/* sem.b 消费者进程 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.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 semid;
	struct sembuf sb;
	VERIFYERR((semid = semget(1000, 2, 0666|IPC_CREAT)) < 0, "open sem 
1000");

	sb.sem_num = 1;
	sb.sem_op = -1;
	sb.sem_flg = sb.sem_flg & ~IPC_NOWAIT;
	VERIFYERR(semop(semid, &sb, 1) != 0, "P sem 1000:1");

	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 = sb.sem_flg & ~IPC_NOWAIT;
	VERIFYERR(semop(semid, &sb, 1) != 0, "V sem 1000:0");

	return 0;
}
```

十、 进程间通讯之——共享内存

共享内存就是物理内存中的一段可以由两个以上的进程共享访问的区域。其最重要的两个方面分别是大小和地址，进程在访问共享内存之前需要将共享内存映射到进程空间一个虚拟地址中，然后就可以使用普通的内存访问函数操作这块内存区域。

6. 共享内存数据结构

内核中使用结构 shmid_ds（Linux 下在/usr/include/bits/shm.h）来表示一个共享内存。

7. 共享内存的创建

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
                                                    返回：共享内存 ID
```

- 该函数创建一个新的共享内存，大小为 size。shmflg 和消息队列类似包含权限和 IPC_CREAT、IPC_EXCL。

8. 映射、释放共享内存

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(cont void *shmaddr);
```

- shmat 将标识号为 shmid 的共享内存映射到调用进程的地址空间中，在大多数情况下我们让系统自动为我们选择映射的地址。

shmaddr	shmflg	映射地址
NULL		自动
非 NULL	未置 SHM_RND	shmaddr
非 NULL	置 SHM_RND	shmaddr-(shmaddr%SHMLAB)

- shmat 成功返回映射的地址，否则返回-1，并置 errno。
- shmdt 参数 shmaddr 必须为 shmat 返回的地址，成功返回0，否则返回-1。

实例

创建大小10×1024字节的共享内存，分10块，写入、读取。

```
/* shm1.c 写如共享内存 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/ipc.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];
     // 打开共享内存
     VERIFYERR((shmid = shmget(0x1234, 10*1024, 0666|IPC_CREAT)) == -1,
"open shm");
     // 映射共享内存
     VERIFYERR((pmat = (char *)shmat(shmid, 0, 0)) == 0, "link shm");
     // 写入的块号
     printf("please input No.(0~9):");
     scanf("%d", &no);
     VERIFYERR(no < 0 || no > 9, "input No.");
     // 写入数据
     printf("please input data: ");
     memset(buf, 0, sizeof buf);
     scanf("%s", buf);
     memcpy(pmat+no*1024, buf, sizeof buf);
     // 释放映射
     shmdt(pmat);
     return 0;
}
```

```
/* shm2.c 读取 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/ipc.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 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;
}
```

9. 共享内存控制

```
#include <sys/sem.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- 读取/设置共享内存信息，删除，锁定，解锁。

十一、 进程间通讯之——Socket

1. socket 地址结构

socket 地址结构定义在 sys/socket.h 中，如下：

```
struct sockaddr {
  u_short sa_family;
  char sa_data[14];
};
```

sa_family 指定协议，sa_data 制定协议地址，这个结构只是一个大致的框架。每种协议都有不同的地址结构，如 AF_INET 地址结构为 sockaddr_in，定义在 netinet/in.h 中，如下：

```
struct soackaddr_in {
    short sin_family;   /* 16位协议（AF_INET）  */
    u_short sin_port;   /*  16位端口地址 */
    struct in_addr sin_addr;    /* 32位 IP 地址 */
    char sin_zero[8];   /* 填充位 */
};
struct in_addr {
    u_long s_addr;
};
```

填充位的作用是使其和 sockaddr 大小相同，sizeof(sockaddr) == sizeof(sockaddr_in)。这样使用通用地址结构 sockaddr 为参数的函数可直接使用 sockaddr_in 类型的参数。

获取套接字地址：

```
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *name, int *namelen);
int getpeername(int s, struct sockaddr *name, int *namelen);
                                              成功返回0，否则-1、置errno
```

- getsockname 获取套接字描述符 s 在本地的主机别名。

- getpeername 获取与描述符 s 相连接的对方的套接字的协议地址信息。

2. IP 地址转换

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
unsigned long inet_addr(char *ptr);
int inet_aton(char *ptr, struct in_addr *addrptr);
char *inet_ntoa(struct in_addr inaddr);
```

3. 字节序转换

```
#include <arpa/inet.h>
u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

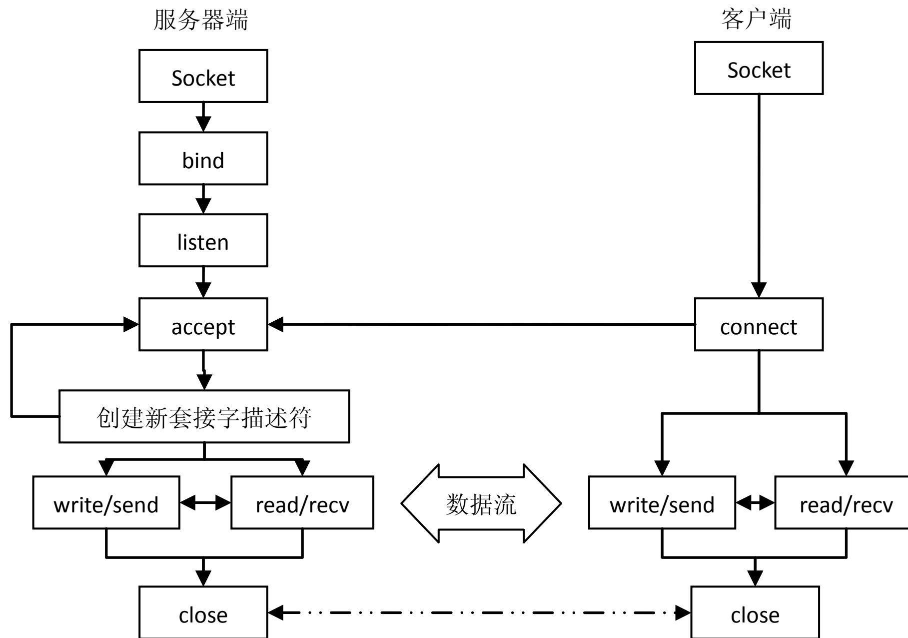
4. socket 端口

```
#include <netdb.h>
struct servent {
    char *s_name; /* 服务器名 */
    char **s_aliases; /* 服务器别名 */
    int s_port; /* 服务器端口号 */
    char *s_proto; /* 使用协议 */
};
struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

实例

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
     struct servent *serv;
     if (argc != 2) exit(1);
     if ((serv = getservbyname(argv[1], "tcp")) == NULL) {
         perror("getservbyname ERROR");
         exit(1);
     }
     printf("Serv name: %s\nServ_aliase name: %s\nServ port: %d\nServ pro-
to: %s\n", serv->s_name, serv->s_aliases, ntohs(serv->s_port), serv-
>s_proto);
     return 0;
}
```

5. TCP C/S 通讯结构



socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- domain 取值通常为 AF_UNIX、AF_INET。
- type 通常为 SOCK_STREAM、SOCK_DGRAM、SOCK_RAW。
- protocol 通常为0，自动选择协议。

bind

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int s, const struct sockaddr *name, int namelen);
                                        成功返回0，否则-1、置 errno
```

listen

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int s, int backlog);
                                        成功返回0，否则-1、置 errno
```

- backlog 确定了套接字 s 接收链接的最大数目。

accept

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, int *addrlen);
                                        成功返回新描述符，否则-1
```

- 返回一个新的描述符和客户端通信，客户端信息在 addr 中。

connect

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s, const struct sockaddr *name, int namelen);
                                        成功返回0，否则-1
```

- SOCK_STREAM 调用 connect，必须等待服务器端的应答结果函数才返回，

SOCK_DGRAM 调用 connect，函数仅仅记录对方套接字地址而不建立真正的链接。

shutdown

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

send

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int s, const void *buf, size_t len, int flags);
                                   成功返回发送长度，套接字关闭返回0，否则-1、置 errno
```

recv

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int s, const void *buf, size_t len, int flags);
                                   成功返回接收到长度，套接字关闭返回0，否则-1、置 errno
```

实例一

返回客户端 IP 信息

```
/* getpeereg.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(void)
{
     int sockfd, tsock;
     struct sockaddr_in addr;
     struct sockaddr *paddr = (struct sockaddr *)&addr;
     socklen_t lenaddr = sizeof(struct sockaddr);
     if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
         perror("socket ERROR");
         exit(1);
     }
     memset(paddr, 0, sizeof(struct sockaddr_in));
     addr.sin_family = AF_INET;
     addr.sin_port = htons(9991);
     addr.sin_addr.s_addr = htonl(INADDR_ANY);
     if (bind(sockfd, paddr, sizeof(struct sockaddr)) == -1 || lis-
ten(sockfd, 10) == -1) {
         perror("bind listen ERROR");
         exit(1);
     }
     memset(paddr, 0, sizeof(struct sockaddr));
     if ((tsock = accept(sockfd, paddr, &lenaddr)) == -1) {
         perror("accept ERROR !");
         exit(1);
     }
     printf("accept return:\n\tClient: %s:%d\n", inet_ntoa(((struct sock-
addr_in *)paddr)->sin_addr), ntohs(addr.sin_port));
     memset(paddr, 0, sizeof(struct sockaddr));
     if(getpeername(tsock, paddr, &lenaddr) == -1) {
         perror("accept ERROR !");
         exit(1);
     }
     printf("getpeername return:\n\tClient: %s:%d\n",  inet_ntoa(((struct
sockaddr_in *)paddr)->sin_addr), ntohs(addr.sin_port));
```

```
    return 0;
}
```

```
Client:
$ telnet 192.168.56.101 9991
```

```
Serv:
$ make getpeereg
$ ./ getpeereg
accept return:
       Client: 192.168.56.1:60150
getpeername return:
       Client: 192.168.56.1:60150
```

实例二

C/S 通讯，client 端通过命令行向 server 传送 msg，并在 server 端控制台上显示这些 msg。

```
/* recvmsgserv.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

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

int main(void)
{
     int sockfd, tsock;
     struct sockaddr_in addr;
     struct sockaddr *paddr = (struct sockaddr *)&addr;
     socklen_t lenaddr = sizeof(struct sockaddr);
     char strbuf[1024];

     VERIFYERR((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1, "socket");

     memset(paddr, 0, sizeof(struct sockaddr_in));
     addr.sin_family = AF_INET;
     addr.sin_port = htons(9999);
     addr.sin_addr.s_addr = htonl(INADDR_ANY);
     VERIFYERR(bind(sockfd, paddr, sizeof(struct sockaddr)) == -1 || lis-
ten(sockfd, 10) == -1, "bind and listen");

     memset(paddr, 0, sizeof(struct sockaddr));
     VERIFYERR((tsock = accept(sockfd, paddr, &lenaddr)) == -1, "accept");

     printf("accept return:\n\tClient: %s:%d\n", inet_ntoa(((struct sock-
addr_in *)paddr)->sin_addr), ntohs(addr.sin_port));
     while(recv(tsock, strbuf, sizeof strbuf, 0) > 0) {
         printf("%s", strbuf);
     }

     return 0;
}
```

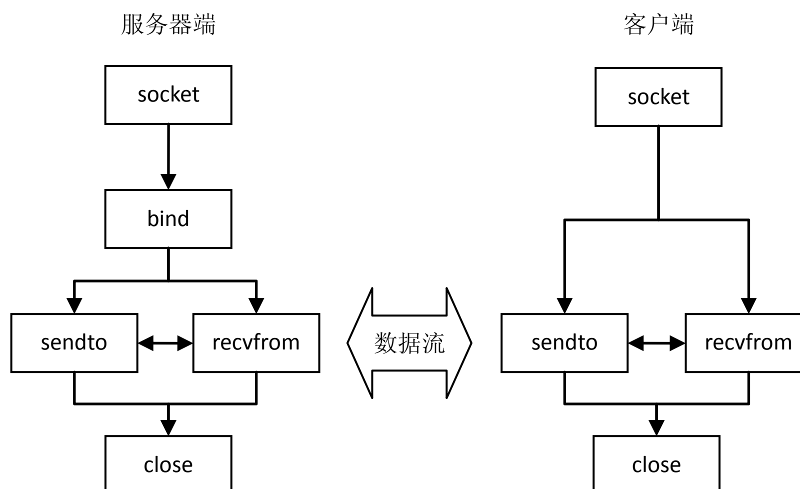
```
/* sndmsgcli.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

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

```
int main(int argc, char *argv[])
{
      int sockfd, tsock;
      struct sockaddr_in addr;
      struct sockaddr *paddr = (struct sockaddr *)&addr;
      socklen_t lenaddr = sizeof(struct sockaddr);
      char strbuf[1024];
      int i = 0;
      if (argc == 1) {printf("usage: a.out arg1 arg2 arg3 ...\n"); exit(0);}

      VERIFYERR((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1, "socket");

      memset(paddr, 0, sizeof(struct sockaddr_in));
      addr.sin_family = AF_INET;
      addr.sin_port = htons(9999);
      addr.sin_addr.s_addr = htonl(INADDR_ANY);
      VERIFYERR(connect(sockfd, paddr, lenaddr) == -1, "connect");

      for (i = 0; i < argc; ++i) {
          sprintf(strbuf, "argv[%d]: %s\n", i, argv[i]);
          VERIFYERR(send(sockfd, strbuf, sizeof strbuf, 0) <= 0, "send");
      }
      return 0;
}
```

```
$ ./sndmsgcli 1 asdf 23 asf  43 我的通讯
```

```
$ ./recvmsgserv
accept return:
       Client: 127.0.0.1:42733
argv[0]: ./sndmsgcli
argv[1]: 1
argv[2]: asdf
argv[3]: 23
argv[4]: asf
argv[5]: 43
argv[6]: 我的通讯
```

6. UDP C/S 通讯结构



sendto

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

```
成功返回发送字节数，失败返回-1、置errno
```

recvfrom

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr
*from, socklen_t *fromlen);
                    成功返回接收字节数，套接口关闭返回0，失败返回-1、置errno
```

实例一

UDP 实现 C/S 结构通讯。

```
/* udpserv.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

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

int main(void)
{
     int sockfd, tsock;
     struct sockaddr_in addr;
     struct sockaddr *paddr = (struct sockaddr *)&addr;
     socklen_t lenaddr = sizeof(struct sockaddr);
     char strbuf[1024];

     VERIFYERR((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) == -1, "socket");

     memset(paddr, 0, sizeof(struct sockaddr_in));
     addr.sin_family = AF_INET;
     addr.sin_port = htons(9999);
     addr.sin_addr.s_addr = htonl(INADDR_ANY);
     VERIFYERR(bind(sockfd, paddr, sizeof(struct sockaddr)) == -1, "bind and
listen");

     while (recvfrom(sockfd, strbuf, sizeof strbuf, 0, paddr, &lenaddr) > 0)
{
         printf("%s\n", strbuf);
     }
     shutdown(sockfd, 2);
     return 0;
}
```

```
/* udpcli.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

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

int main(int argc, char *argv[])
{
     int sockfd, tsock;
     struct sockaddr_in addr;
     struct sockaddr *paddr = (struct sockaddr *)&addr;
     socklen_t lenaddr = sizeof(struct sockaddr);
```

```
    char strbuf[1024];
    int i;
    VERIFYERR((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) == -1, "socket");
    memset(paddr, 0, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9999);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    for (i = 0; i < argc; ++i) {
        sprintf(strbuf, "argv[%d]=%s\n", i, argv[i]);
        VERIFYERR(sendto(sockfd, strbuf, sizeof strbuf, 0, paddr, lenaddr)
<= 0, "recvfrom");
        //printf("sendto: %s\n", argv[i]);
    }
    shutdown(sockfd, 2);
    return 0;
}
```

socket 地址结构 sockaddr 如下：

阿德发大水

sfaf