

# 网络程序设计报告

## 实验一 linux 平台上的 TCP 并发服务

### 1 实验内容

掌握基本套接字函数使用方法、TCP 协议工作原理、并发服务原理和编程方法。实验内容：

在 linux 平台上实现 1 个 TCP 并发服务器，至少可以为 10 个客户端同时提供服务。

(1) 基于 TCP 套接字编写服务器端程序代码，然后编译和调试；

(2) 服务器程序要达到：可以绑定从终端输入的 IP 地址和端口；可以显示每一个进程的进

程号；可以显示当前并发执行的进程数量；可以根据客户机要求的服务时间确定进程的生存

时间。

(3) 基于 TCP 套接字编写客户端程序代码，然后编译和调试；

(4) 客户端程序要达到：可以从终端输入服务器的 IP 地址和端口；可以从终端输入对服务

器的服务时间要求。

(5) 联调服务器和客户端，服务器每收到一个连接就新建一个子进程，在子进程中接收客

户端的服务时间请求，根据所请求的时间进行延时，然后终止子进程。如：客户端请求服务

10s，则服务器的子进程运行 10s，然后结束。

(6) 服务器要清除因并发服务而产生的僵尸进程。

### 2 设计分析

(1) 要求处理 10 个以上客户端，服务端采用 TCP 并发服务器，每进行一次连接创建一个

子进程，子进程总数加 1。

(2) 需要实现端口的绑定和运行时间的设置，中国在客户端实现，argv[]

中输入需要的参数，

参数不正确的时候出错处理。

(3)延时功能我使用的 `sleep()`，直接延时从客户端发送过来的要求的时间，最后结束的时候

子进程需要减 1。

(4)处理僵尸进程，自己实现一个 `sigchld_handler` 实现僵尸进程的清理。

### 3 实现代码

server.c:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>

#define LISTENQ 1024
#define BUFSIZE 1024

int open_listenfd(int port);
void service(int connfd);
void sigchld_handler(int sig);

static int pcnt;
```

```
int main(int argc, char *argv[])
{
    int listenfd, connfd, port;
    unsigned int clientlen;
    struct sockaddr_in clientaddr;
    pid_t pid;

    if (argc != 2) {
        fprintf(stderr, "%s <port>\n", argv[0]);
        exit(-1);
    }
    port = atoi(argv[1]);

    signal(SIGCHLD, sigchld_handler);
    if ((listenfd = open_listenfd(port)) == -1) {
        fprintf(stderr, "%s\n", strerror(errno));
        exit(-1);
    }
    for (;;) {
        clientlen = sizeof(clientaddr);
        do {
            connfd = accept(listenfd,
                           (struct sockaddr *)&clientaddr.sin_addr.s_addr,
                           &clientlen);
        } while (connfd == -1 && errno == EINTR);

        if (connfd == -1) {
            fprintf(stderr, "%s\n", strerror(errno));
            exit(-1);
        }
    }
}
```

```

    }

    if ((pid = fork()) == -1) {
        fprintf(stderr, "%s\n", strerror(errno));
    } else {
        pcnt++;
        if (pid == 0) {
            close(listenfd);
            service(connfd);
            close(connfd);
            exit(EXIT_SUCCESS);
        }
    }
    close(connfd);
}

exit(EXIT_SUCCESS);
}

void service(int connfd)
{
    char buf[BUFSIZE];
    int seconds;
    time_t tmval;
    const struct tm *tmptr;

    read(connfd, buf, BUFSIZE);
    sscanf(buf, "%d", &seconds);
    if (seconds > 0) {
        tmval = time(0);
        tmptr = localtime(&tmval);
    }
}

```

```

        printf("process %d: %d seconds. total child_pid: %d\n",getpid(),
seconds, pcnt);

        sleep(seconds);
    }
}

void sigchld_handler(int sig)
{
    pid_t pid;
    time_t tmval;
    const struct tm *tmptr;

    while ((pid = (waitpid(-1, NULL, WNOHANG))) > 0) {
        tmval = time(0);
        tmptr = localtime(&tmval);
        printf("process %d exited. total child_pid: %d\n",pid, --pcnt);
    }
    signal(SIGCHLD, sigchld_handler);
}

int open_listenfd(int port)
{
    int listenfd, optval = 1;
    struct sockaddr_in serveraddr;

    /* creat socket */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* setsocket */

```

```

    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                    (const void *)&optval, sizeof(int)) < 0)
        return -1;

    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short) port);

    /*bind*/
    if (bind(listenfd, (struct sockaddr *)&serveraddr,
              sizeof(serveraddr)) < 0)
        return -1;

    if (listen(listenfd, LISTENQ) < 0)
        return -1;
    return listenfd;
}

```

client.c:

```

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

```

```
#include <unistd.h>

#include <sys/types.h>

int open_clientfd(char *hostname, int port);

int main(int argc, char *argv[])
{
    int clientfd, port;
    char *host, *seconds;

    if (argc != 4) {
        fprintf(stderr, "usage: %s <host> <port> <seconds>\n", argv[0]);
        exit(-1);
    }

    host = argv[1];
    port = atoi(argv[2]);
    seconds = argv[3];

    if ((clientfd = open_clientfd(host, port)) == -1) {
        fprintf(stderr, "%s\n", strerror(errno));
        exit(-1);
    }

    write(clientfd, seconds, strlen(seconds));
    close(clientfd);
    exit(EXIT_SUCCESS);
}

int open_clientfd(char *hostname, int port)
{
    int clientfd;
```

```

struct hostent *hp;

struct sockaddr_in serveraddr;

if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;

if ((hp = gethostbyname(hostname)) == NULL)
    return -2;

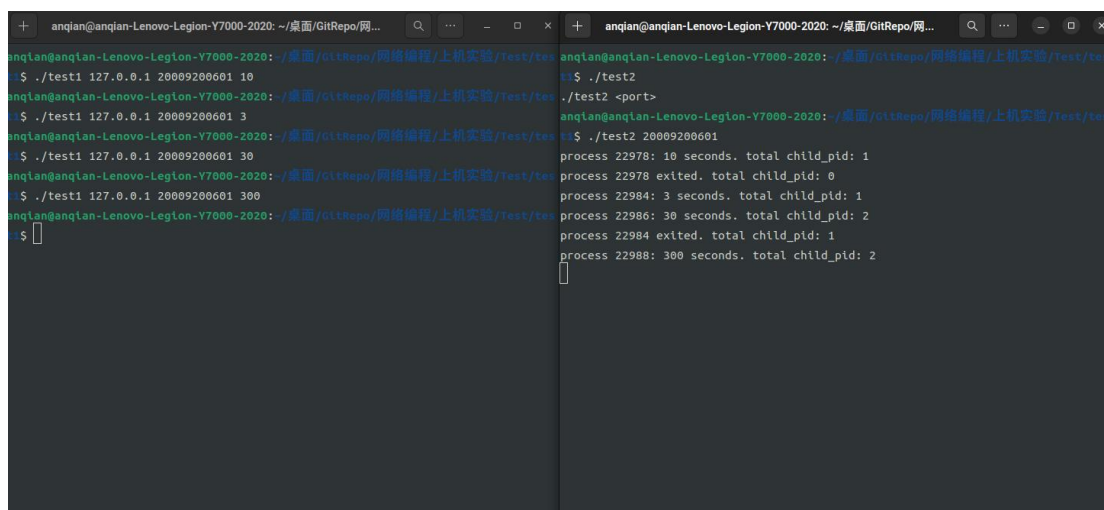
memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
memcpy((char *)&serveraddr.sin_addr.s_addr,
        (char *)hp->h_addr_list[0],
        hp->h_length);
serveraddr.sin_port = htons(port);

/*connect */
if (connect(clientfd, (struct sockaddr *)&serveraddr,
            sizeof(serveraddr)) < 0)
    return -1;
return clientfd;
}

```

#### 4. 实验结果:





The image shows two terminal windows side-by-side. The left window shows a loop of commands: `./test1 127.0.0.1 20009200601 10`, `./test1 127.0.0.1 20009200601 3`, `./test1 127.0.0.1 20009200601 30`, and `./test1 127.0.0.1 20009200601 300`. The right window shows `./test2` followed by `./test2 <port>` with values 10, 3, 30, and 300. The output for `./test2` shows process execution times and child\_pids: `process 22978: 10 seconds. total child_pid: 1`, `process 22978 exited. total child_pid: 0`, `process 22984: 3 seconds. total child_pid: 1`, `process 22986: 30 seconds. total child_pid: 2`, `process 22984 exited. total child_pid: 1`, and `process 22988: 300 seconds. total child_pid: 2`.

## 实验二 进程间的协调通信

### 1. 实验内容

掌握进程的概念、进程间通信的基本原理、集成间通信的主要类型和各自的特点。实验内容：

在 linux 平台上实现 1 个父进程、2 个子进程，利用管道和共享内存实现两个子进程之间数

据快速传送。

- (1) 创建一个进程，再创建一对管道、一块共享内存（大于 64kB）；
- (2) 通过 `fork()` 函数生成 2 个子进程；
- (3) 调试程序，确定父、子进程之间可以双向通信；
- (4) 调试程序，确定两个子进程之间可以通过父进程中转实现双向通信；
- (5) 调试程序，确定两个子进程都可访问共享内存；
- (6) 实现两个子进程之间无冲突地访问共享内存。传送的数据块不小于 32kB，为了能够看

到演示效果，读/写每个字节后都延时 0.5ms。

### 2. 设计分析

对于管道：

匿名管道实现起来较为简单方便，但是也有其局限。匿名管道只能实现单向通信，因此为了实现双向通信，需要建立两个管道。此外，匿名管道只能被应用于有亲缘关系的进程之间的通信，比如父子进程。下面是实现进程间匿名管道通

信的方式:

1. 创建管道: 使用系统调用 `pipe()` 来创建匿名管道。这个系统调用将会创建一个包含两个文件描述符的整数数组。第一个文件描述符用于读取管道输出, 第二个文件描述符用于写入管道输入。
2. 创建子进程: 使用系统调用 `fork()` 创建一个新的子进程。子进程将会成为管道的消费者, 而父进程将会成为管道的生产者。
3. 创建子进程: 使用系统调用 `fork()` 创建一个新的子进程。子进程将会成为管道的消费者, 而父进程将会成为管道的生产者。
4. 使用管道读写

对于共享内存:

进程直接读写内存, 不需要任何数据的拷贝

- 为了在多个进程间交换信息, 内核专门留出了一块内存区
- 由需要访问的进程将其映射到自己私有地址空间
- 进程直接读写这一内存区而不需要进行数据的拷贝, 提高了效率

多个进程共享一段内存, 需要依靠某种同步机制, 如互斥锁和信号量等

3. 代码实现

pipe.c

```
#include<unistd.h>
#include<iostream>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<wait.h>
#include<string.h>
#include<sys/types.h>
using namespace std;

int main(int argc, char **argv)
{
    int pipe1[2], pipe2[2];
```

```
//pid_t pid1,pid2;

int n;

char cstr[]="child1 data";

char buf[128];

//char buf2[128];

//memset(buf1, '\0', 128); //初始化

//memset(buf2, '\0', 128);


if(pipe(pipe1)<0||pipe(pipe2)<0)

    cout<<"pipe error"<<endl;


pid_t pid1=fork();

pid_t pid2=fork();

if(pid1==-1)

{

    printf("fork pid1 failed!");

}

else if(pid1 > 0)

{

    //父进程,用管道1 读数据,管道2 写数据

    cout<<"father PID:"<<getpid()<<endl;

    if(read(pipe1[0],buf,100)>0)

        cout<<"father received:"<<buf<<endl;

    write(pipe2[1],buf,strlen(buf));

}

else if(pid1==0)

{

    cout<<"\npid1:"<<getpid()<<endl;

    //子进程1 用管道1 写数据

    close(pipe1[0]); //关闭pipe1 读端口
```

```

        close(pipe2[0]); // 关闭 pipe2 读端口
        close(pipe2[1]); // 关闭 pipe2 写端口
        write(pipe1[1], cstr, strlen(cstr));
    }
    if(pid2 == -1)
    {
        printf("fork pid2 failed!");
    }
    else if(pid2 == 0)
    {
        // 子进程 2 用管道 2 读数据
        close(pipe1[0]); // 关闭 pipe1 读端口
        close(pipe1[1]); // 关闭 pipe1 写端口
        close(pipe2[1]); // 关闭 pipe2 写端口
        if(read(pipe2[0], buf, 128) > 0)
        {
            cout << "\npid2: " << getpid() << endl;
            // printf("\npid2 : pid=%d\n", getpid());
            // printf("pid2 : buf=%s\n", buf);
            cout << "pid2 receive: " << buf << endl;
            exit(0);
        }
    }
    return 0;
}

```

memshare.c:

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

```

```
#include <iostream>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <signal.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define MEM_SIZE 1024

using namespace std;
int shmid;
int status;
main(int argc, char** argv)
{
    char* p_addr, *c1_addr, *c2_addr;
    key_t key;
    char* name = (char* )"/dev";
    char* text = (char* )"hello";
    int pid1,pid2;

    key=ftok(name,'a');
    if(key==-1)
    {
        perror("ftok error");
        exit(1);
    }
}
```

```

}

shmid=shmget(key, MEM_SIZE, 0777 | IPC_CREAT);

if(shmid==-1)
{
    perror("shmget error");
    exit(1);
}

if((pid1=fork())==-1)
{
    printf("childprocess1 failed!");
}
else if(pid1>0)
{
    cout<<"father PID:"<<getpid()<<endl;
    wait(&status);
    p_addr = (char* )shmat(shmid, 0, 0);
    cout<<"father received:"<<p_addr<<endl;
}
else
{
    cout<<"child1 PID:"<<getpid()<<endl;
    c1_addr = (char* )shmat(shmid, 0, 0);
    memset(c1_addr, '\0', 1024);
    strncpy(c1_addr, text, 1024);
    exit(0);
}

if((pid2=fork())==-1)
{
    printf("childprocess2 failed!");
}

```

```
    }  
    else if(pid2==0)  
    {  
        c2_addr = (char* )shmat(shmid,0,0);  
        cout<<"\nchild2 PID:"<<getpid()<<endl;  
        cout<<"child2 received:"<<c2_addr<<endl;  
        exit(0);  
    }  
  
    return 0;  
}
```

#### 4. 实验结果

```
anqian@anqian-Lenovo-Legion-Y7000-2020: ~/桌面/GitRepo/网...
anqian@anqian-Lenovo-Legion-Y7000-2020:~/桌面/GitRepo/网络编程/上机实验/Test/tes
t2$ ./test1
father PID:23142

father PID:23144

pid1:23143
father received:child1 data♦♦
pid1:23145
father received:child1 data♦♦

pid2:23144
pid2 receive:child1 data♦♦child1 data♦♦♦♦
anqian@anqian-Lenovo-Legion-Y7000-2020:~/桌面/GitRepo/网络编程/上机实验/Test/tes
t2$ ./test2
father PID:23169
child1 PID:23170
father received:hello

child2 PID:23171
child2 received:hello
anqian@anqian-Lenovo-Legion-Y7000-2020:~/桌面/GitRepo/网络编程/上机实验/Test/tes
t2$
```

### 实验三 Windows 平台上的 TCP 并发服务

#### 1. 实验内容

编程内容与实验 1 相同，操作系统为 windows。了解 Windows 与 Linux 平台编程环境的差异，掌握 Winsock 编程接口及编程方法。

#### 2. 实验试剂分析

服务控制管理器（SCM）是 Windows 操作系统中负责管理和控制系统服务的组件。它提供了一组 API，允许应用程序与 SCM 进行通信以管理服务。下面是一个基本的服务控制管理器通信流程：

连接到 SCM：应用程序通过调用 OpenSCManager 函数来连接到 SCM。这个函数返回一个句柄，表示与 SCM 的连接。

查询服务：通过调用 EnumServicesStatus 函数或相关函数，应用程序可以



向 SCM 查询系统中安装的服务的信息。这可以用来获取服务的名称、状态和其他属性。

打开服务：通过调用 `OpenService` 函数，应用程序可以打开一个已注册的服务。这个函数返回一个句柄，表示对服务的访问权限。

控制服务：应用程序可以使用打开的服务句柄调用 `StartService`、`StopService`、`PauseService`、`ResumeService` 等函数来控制服务的状态。这些函数允许应用程序启动、停止、暂停、恢复服务的运行。

监视服务状态：通过调用 `QueryServiceStatus` 或相关函数，应用程序可以获取服务的当前状态信息。这可以用来检查服务是否正在运行、已停止或处于其他状态。

关闭服务和 SCM 连接：在完成对服务的操作后，应用程序应调用 `CloseServiceHandle` 来关闭服务句柄，以释放相关资源。最后，应用程序还需要调用 `CloseServiceHandle` 来关闭与 SCM 的连接句柄。使用 `winsocket` 编程实现即可，需要查看许多函数，与 `linux` 稍微不同

我没有实现并发，只是单纯实现了这个模型，还有延时功能。

### 3. 代码实现

server.cpp

```
#include <stdio.h>
#include <winsock2.h>
#include <unistd.h>
#include <time.h>
#pragma comment (lib, "ws2_32.lib")

int main()
{
    WSADATA wsaData;
    WSStartup( MAKEWORD(2, 2), &wsaData);

    fputs("port: ", stdout);

    int port;
```

```
scanf("%d", &port);

SOCKET servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

struct sockaddr_in sockAddr;
memset(&sockAddr, 0, sizeof(sockAddr));
sockAddr.sin_family = PF_INET;
sockAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
sockAddr.sin_port = htons(port);
bind(servSock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

listen(servSock, 20);

SOCKADDR cIntAddr;
int nSize = sizeof(SOCKADDR);

for (;;) {
    SOCKET cIntSock = accept(servSock, (SOCKADDR*)&cIntAddr, &nSize);
    char recvBuf[MAXBYTE] = {0};
    recv(cIntSock, recvBuf, MAXBYTE, 0);
    int sec;
    time_t tmval;
    struct tm *tmptr;
    sscanf(recvBuf, "%d", &sec);
    if (sec > 0) {
        tmval = time(0);
        tmptr = localtime(&tmval);
        printf("[%02d:%02d:%02d] server will sleep %d seconds\n",
            tmptr->tm_hour, tmptr->tm_min,
            tmptr->tm_sec, sec);
    }
}
```

```

        sleep(sec);

        tmval = time(0);
        tmptr = localtime(&tmval);

        printf("[%02d:%02d:%02d] server is wake up\n",
                tmptr->tm_hour, tmptr->tm_min, tmptr->tm_sec);
    }

    closesocket(clntSock);
}

closesocket(servSock);

WSACleanup();

return 0;
}

```

client.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#pragma comment(lib, "ws2_32.lib")

int main()
{
    WSADATA wsaData;

    WSStartup(MAKEWORD(2, 2), &wsaData);

    fputs("ipaddress: ", stdout);

    char host[64];
}

```

```
scanf("%s", host);

fputs("port: ", stdout);
int port;
scanf("%d", &port);

SOCKET sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

struct sockaddr_in sockAddr;
memset(&sockAddr, 0, sizeof(sockAddr));
sockAddr.sin_family = PF_INET;
sockAddr.sin_addr.s_addr = inet_addr(host);
sockAddr.sin_port = htons(port);
connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR));

char msg[MAXBYTE];
fputs("input serve second: ", stdout);
int time;
scanf("%d", &time);
snprintf(msg, MAXBYTE, "%d\n", time);
send(sock, msg, strlen(msg) + 1, 0);

closesocket(sock);

WSACleanup();

return 0;
}
```

#### 4. 实验结果

```
Suggestion [3,General]: 找不到命令 server.exe, 但它确实存在于当前位置。默认情况下, Windows PowerShell 不会从当前位置命令。如果信任此命令, 请改为键入 ".\server.exe"。有关详细信息, 请参阅 "get-help about_Command_Precedence"。
PS D:\GitRepo\网络编程\上机实验\Test\test3> .\server
port: 9000
[22:36:18] server will sleep 10 seconds
[22:36:28] server is wake up
[22:36:33] server will sleep 5 seconds
[22:36:38] server is wake up
[22:36:44] server will sleep 6 seconds
[22:36:50] server is wake up

Windows PowerShell
PS D:\GitRepo\网络编程\上机实验\Test\test3> .\client
ip: 127.0.0.1
port: 9000
input serve second: 10
PS D:\GitRepo\网络编程\上机实验\Test\test3> .\client
ip: 127.0.0.1
port: 9000
input serve second: 5
PS D:\GitRepo\网络编程\上机实验\Test\test3> .\client
ip: 127.0.0.1
port: 9000
input serve second: 6
PS D:\GitRepo\网络编程\上机实验\Test\test3>
```

从输出的结果来看基本实现了所要求的功能。

实验中遇到了一些 Windows 的报错, 最开始无法执行编译出来的 .exe 程序, 原因是 powershell 与 cmd 命令行的指令格式有差异, 更换指令后即可成功。