计算机网络

实验（三）

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年11月3日
学　　院	信息学院
课程名称	计算机网络

实验（三）

目录

1 编译和运行代码

实验代码在以下环境中进行了编译和运行：

- 操作系统：Ubuntu 22.04 (on Windows Subsystem for Linux)
- 编译器：GCC 11.3.0
- CMake：3.22.1

为了编译代码，需要使用CMake，在项目目录下建立目录build，然后依次运行：

1. cd build
2. cmake ..
3. make

就可以得到server和client两个可执行文件。运行方式都是"**可执行文件名 IP PORT**"。分别为程序指定IP和端口。

2 完善Socket客户机和服务器

2.1 程序实现

2.1.1 通过命令行参数指定IP和端口

首先，为了实现通过参数指定服务器IP和端口，需要处理命令行参数。先检查主函数的*argc*，如果不是3，说明参数不对，打印提示信息并退出。如代码 1。

代码 1: 检查*argc*

```
if (argc != 3)
{
        printf("Usage: %s <IP> <PORT>", argv[0]);
        return -1;
}
```

然后在构造$sever_addr$结构体的时候根据输入的IP和端口号进行赋值。用户输入的端口号是一个数字字符串，首先使用 *atoi*转换成整数，然后通过*htons*转换成网络字节序。而用户输入的IP地址是一个点分十进制字符串，需要使用*inet_aton*转换成一个整形。如代码 2。

代码 2: 构造$sever_addr$结构体

```
server_addr.sin_port = htons(atoi(argv[2]));
inet_aton(argv[1], &server_addr.sin_addr);
```

这样，就可以通过命令行参数指定服务器IP和端口了。

2.1.2 服务器端处理SIGINT信号

由于服务器端使用*bind*绑定了端口，如果退出时没有正确释放资源，端口就会被占用。之后的运行就需要指定其他端口，造成了资源的浪费。使用Ctrl+C退出时，服务器端会收到SIGINT信号，可以在主函数中捕获这个信号，然后在信号处理函数中调用*close*释放资源。否的程序会直接退出，因而无法释放资源，造成问题。如代码 3。

代码 3: 捕获SIGINT信号

```
void release_socket(int signo)
{
        close(server_sock_listen);
        exit(-1);
}

// in int main()
signal(SIGINT, release_socket);
```

2.1.3 为Socket添加错误处理

为了给Socket添加错误处理，需要在每次调用Socket函数的时候检查返回值， Socket如果返回-1，说明调用失败，那么就打印错误信息并退出。在退出时需要注意释放已经获取的资源。为了更好地解决资源释放的问题，可以采用Linux内核中“向下goto”的方式跳转到相应的异常处理部分。这种方式是对goto的一种合理使用。

正如Linux Device Drivers Book中 [1]所说的：

The goto is useful *in a routine that allocates resources, performs operations on those resources, and then deallocates the resources. With a goto, you can clean up in one section of the code. The goto* reduces the likelihood of your forgetting to deallocate the resources in each place you detect an error.

虽然goto语句在C语言中被认为是不好的编程风格 [2]，但是在错误处理这种特殊场景中是一种正确的使用。这种风格的例子如代码 4。

代码 4: 向下goto

```
int foo(int bar)
{
    int return_value = 0;
    if (!do_something( bar )) {
        goto error_1;
    }
    if (!init_stuff( bar )) {
        goto error_2;
    }
    if (!prepare_stuff( bar )) {
        goto error_3;
    }
    return_value = do_the_thing( bar );
error_3:
    cleanup_3();
error_2:
    cleanup_2();
error_1:
    cleanup_1();
    return return_value;
}
```

在附录：代码清单中，代码 8、代码 9、代码 10 都用了这种方式来处理多个fd的释放。

2.1.4 允许用户输入字符串

为了允许用户输入字符串，可以开辟一定的缓冲区，然后使用$fgets$输入任意字符。使用$fgets$的好处是它可以指定缓冲区长度，可以防止缓冲区溢出，更为安全。另外，使用$fgets$输入的字符串末尾会有一个'\0'，所以不需要再手动添加'\0'。也不会因为忘记添加该符号而造成问题。

2.1.5 使用bye退出

服务器端需要把处理和返回消息的部分加入一个循环中，直到收到bye消息才退出。客户端在服务器返回bye消息后再释放资源退出。

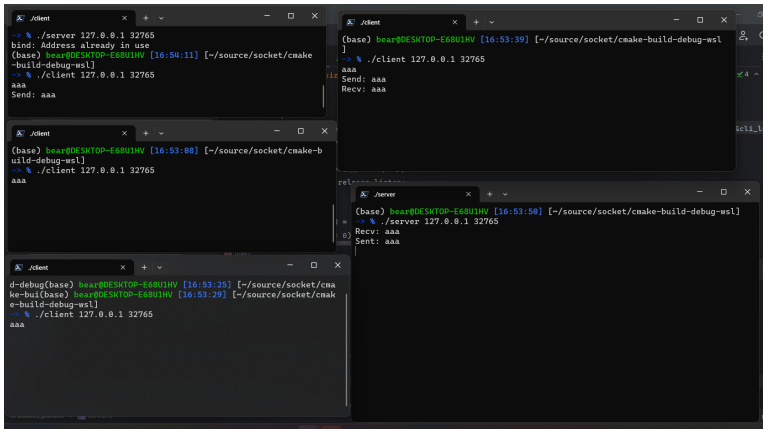
2.1.6 将服务器端改为迭代服务器

迭代服务器可以依次对所有的客户机进行服务。在服务器端，需要使用一个循环来接收客户机的连接请求。所有的请求都在服务器的单个进程中依次处理。为此，可以把服务器端的代码改为如代码 9所示。将$accept$调用以及对消息的处理放进一个循环中，这样就可以依次接收多个客户机的连接请求了。当客户机输入bye的时候进入下一次循环，服务器端就会关闭当前的连接，然后$accept$等待下一个客户机的连接请求。

2.2 客户机和服务器的运行情况

如图 1，运行迭代版的服务器，用4个客户端进行连接。可以看到，服务器端只能依次处理4个客户端的请求。每次只有一个服务端能得到处理。当该客户端输入bye结束后，下一个客户端才能得到处理。

图 1: 迭代版的服务器



2.3 Server端的Backlog

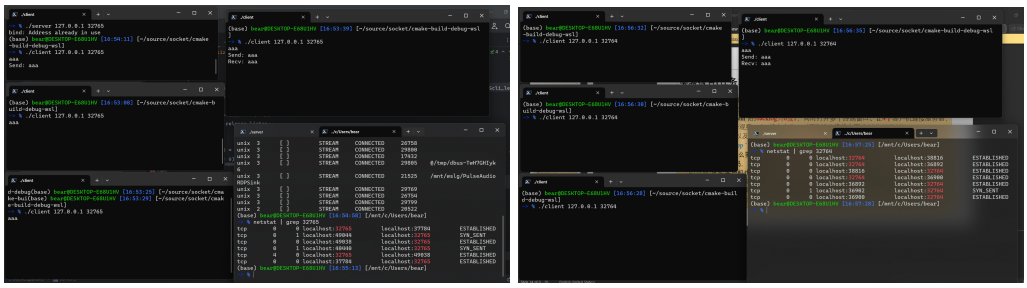
在服务器端，Socket建立连接主要分为以下几步：

1. 将TCP状态设置为LISTEN状态，开启监听客户端的连接请求
2. 收到客户端发送的SYN报文后，TCP状态切换为SYN RECEIVED，并发送SYN ACK报文
3. 收到客户端发送的ACK报文后，TCP三次握手完成，状态切换为ESTABLISHED

其中，第一步通过调用$listen$完成。其中$backlog$是已完成的连接队列(ESTABLISHED)与未完成连接队列(SYN_RCVD)之和的上限。将参数设置为0和1的运行结果使用netstat进行统计，如图 **??**所示。

可以发现，对于4个尝试建立连接的客户端，当$backlog = 0$时，使用netstat可以发现有 2对连接处于ESTABLISHED状态，当$backlog = 1$时，使用netstat则有3对。可见， $backlog$确实限制了服务器端的连接数。

图 2: 不同的*backlog*



(a) $backlog = 0$ (b) $backlog = 1$

3 TCP并发服务器

3.1 程序实现

为了实现多进程，需要使用$fork$系统调用。首先，为了方便建立多个连接，需要将$listen$的参数n设置成较大的数，例如5。然后循环进行$accept$调用，接收客户机的连接请求。

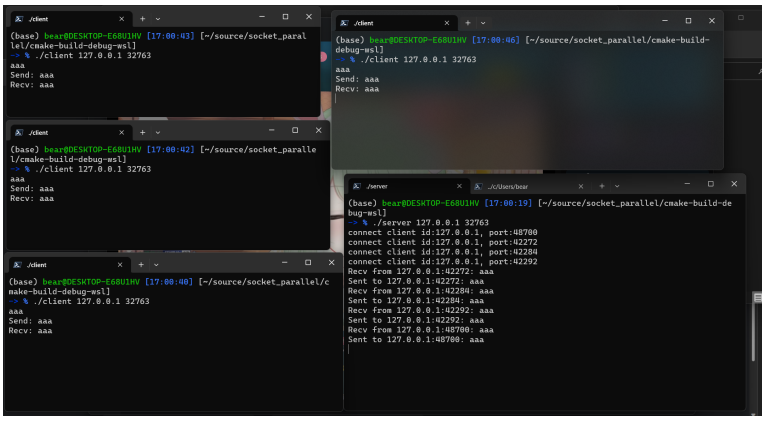
一旦接受了客户机的连接请求，就需要创建一个子进程来处理客户机的请求。在父进程中，需要关闭服务器端的套接字，在子进程中，执行正常的TCP Echo Server的执行流程。当收到Bye结束时，子进程需要关闭套接字，释放资源，然后退出。

判断父进程和子进程通过$fork$的返回值来判断。如果$fork$返回值为0，说明是子进程，如果返回值大于0，说明是父进程。代码见附录：代码清单中的代码 10。

3.2 并发服务器的运行情况

运行并发服务器，结果如图 3所示。四个客户端都能分别发送消息，服务器端也能正常接收。

图 3: 并发的服务器



3.3 父进程分支中是否需要关闭Socket

从结论上来说，父进程分支中必须关闭$Socket$，否则会造成系统资源的浪费。

3.3.1 理论分析

为了分析这个问题，首先要理解*fork*的功能。*fork*的作用是创建一个新的进程，新的进程是原进程的一个副本。这个系统调用是所有类Unix系统进程模型的中心。因此我们可以通过分析一个相对简单的Unix操作系统XV6的*fork*系统调用实现来分析这个问题。

XV6的*fork*系统调用实现如代码 5所示。在*fork*系统调用中，首先调用了*allocproc*函数来分配一个进程结构体。然后对这个进程结构体进行初始化，包括设置进程的状态，设置进程的父进程，设置进程的内存空间等。此时，父进程的整个内存内容都通过 *copyuvm*进行了复制，然后所有父进程打开的文件都通过*filedup*进行了复制。最后，子进程*trapframe*的*eax*设置成了1，这样在子进程中*fork*的返回值就会成为0。

代码 5: XV6的*fork*系统调用实现

```
1
2  int
3  fork(void)
4  {
5    int i, pid;
6    struct proc *np;
7
8    // Allocate process.
9    if((np = allocproc()) == 0)
10     return -1;
11
12   // Copy process state from p.
13   if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
14     kfree(np->kstack);
15     np->kstack = 0;
16     np->state = UNUSED;
17     return -1;
18   }
19   np->sz = proc->sz;
20   np->parent = proc;
21   *np->tf = *proc->tf;
22   np->mode = proc->mode;
23   np->parentfds = proc->parentfds;
24
25   // Clear %eax so that fork returns 0 in the child.
26   np->tf->eax = 0;
27
28   for(i = 0; i < NOFILE; i++)
29     if(proc->ofile[i])
30       np->ofile[i] = filedup(proc->ofile[i]);
31   np->cwd = idup(proc->cwd);
32   np->thr = proc->thr;
33
34   pid = np->pid;
35   np->state = RUNNABLE;
36   safestrcpy(np->name, proc->name, sizeof(proc->name));
37   return pid;
38 }
```

由于Socket API的设计遵从UNIX“一切皆文件”的设计理念，因此Socket调用也是基于文件系统的，打开一个Socket就会创建一个文件描述符。因此，为了确认是否需要在父进程中关闭相应的Socket，需要进一步探究*fork*系统调用中的*filedup*函数的实现。

*filedup*函数的实现如代码 6所示。可见，复制文件描述符的时候，只是将文件描述符的引用计数加1，而不是真正意义上的复制。

代码 6: *filedup*函数的实现

```
struct file *
filedup(struct file *f)
{
        acquire(&ftable.lock);
        if (f->ref < 1)
        {
                panic("filedup");
        }
        f->ref++;
        release(&ftable.lock);
        return f;
}
```

而当关闭文件的时候，系统会检查文件的引用计数，如果引用计数为0，才会真正释放文件，否则不进行任何操作。如代码 7所示。

代码 7: *fileclose*函数的实现

```
void
fileclose(struct file *f)
{
        ...
        if (--f->ref > 0)
        {
                release(&ftable.lock);
                return;
        }
        ...
}
```

这样一来，如果父进程中的Socket没有被关闭，那么子进程中对Socket关闭时，内核首先对相应文件管理结构的引用计数-1，然后发现引用计数不为0，因此不会真正释放文件。这样一来，子进程中的Socket就会一直保持打开状态，没有被关闭。因此，父进程中如果不关闭该Socket，就会对系统资源产生浪费，因为这个Socket将永远不会得到关闭。

由于类UNIX操作系统的*fork*都有相同的功能，所以这一分析可以很好地推广到常见的Linux内核中。当*fork*发生时，相应的描述符会发生逻辑上的复制，所以如果父进程不关闭相应的Socket，就会造成资源的浪费。

3.3.2 实验验证

使用netstat工具来对是否关闭Socket的服务器进行信息的获取，如图 ??。

图 4: 是否关闭

(a) 关闭 (b) 不关闭

可以发现如果不关闭，那么在进程结束之后也会残留CLOSE_WAIT状态的连接。这大大浪费了系统资源。因此，需要在父进程中关闭相应的Socket。

4 实验总结

通过这次实验，我学习了如何调用Socket API通过TCP连接收发网络数据。并实现了迭代版和并发版的不同服务器端。在实验过程中，我学习了如何使用netstat 工具获得所需要的信息。同时，通过理论分析和实验验证，我学到了在$fork$之后，必须关闭父进程中的Socket这一原则。这一原则对于防止网络程序对系统资源的浪费有很大的帮助。

参考文献

References

[1] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers.* " O'Reilly Media, Inc.", 2005.

[2] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

附录：代码清单

代码 8: 客户机

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <error.h>
#include <stdlib.h>
#include <signal.h>

char recv_msg[255];
char send_msg[255];

int client_sock;

void release_socket(int signo)
{
        close(client_sock);
}

int main(int argc, char *argv[])
{
        signal(SIGINT, release_socket);
        struct sockaddr_in server_addr;

        if (argc != 3)
        {
                printf("Usage: %s <IP> <PORT>", argv[0]);
                return -1;
        }

        client_sock = socket(AF_INET, SOCK_STREAM, 0);
        if (client_sock < 0)
        {
                perror("socket");
                return -1;
        }

        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(atoi(argv[2]));
        inet_aton(argv[1], &server_addr.sin_addr);
        memset(server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));

        long err = connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
        if (err < 0)
        {
                perror("connect");
                goto release;
        }

```

```
52          for (;;)
53          {
54                  fgets(send_msg, 255, stdin);
55
56                  printf("Send: %s", send_msg);
57                  err = send(client_sock, send_msg, strlen(send_msg), 0);
58                  if (err < 0)
59                  {
60                          perror("send");
61                          goto release;
62                  }
63
64                  memset(recv_msg, 0, sizeof(recv_msg));
65                  err = recv(client_sock, recv_msg, sizeof(recv_msg), 0);
66                  if (err < 0)
67                  {
68                          perror("recv");
69                          goto release;
70                  }
71
72                  printf("Recv: %s", recv_msg);
73
74                  if (strncmp(recv_msg, "bye", 3) == 0)
75                  {
76                          break;
77                  }
78          }
79
80          return 0;
81  release:
82          close(client_sock);
83
84          return -1;
85  }
```

代码 9: 服务器（迭代）

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <sys/types.h>
4   #include <sys/socket.h>
5   #include <netinet/in.h>
6   #include <arpa/inet.h>
7   #include <unistd.h>
8   #include <error.h>
9   #include <stdlib.h>
10  #include <signal.h>
11
12  int server_sock_listen, server_sock_data;
13
14  void release_socket(int signo)
15  {
16          close(server_sock_listen);
17          close(server_sock_data);
18  }
```

```
19
20  int main(int argc, char *argv[])
21  {
22          signal(SIGINT, release_socket);
23
24          struct sockaddr_in server_addr;
25          char recv_msg[255];
26
27          if (argc < 3)
28          {
29                  printf("Usage: %s <ip> <port>", argv[0]);
30          }
31
32          server_sock_listen = socket(AF_INET, SOCK_STREAM, 0);
33          if (server_sock_listen == -1)
34          {
35                  perror("socket");
36                  return -1;
37          }
38
39          server_addr.sin_family = AF_INET;
40          server_addr.sin_port = htons(atoi(argv[2]));
41  //      server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
42          inet_aton(argv[1], &server_addr.sin_addr);
43          memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
44
45          long err = bind(server_sock_listen, (struct sockaddr *)&server_addr, sizeof(server_addr));
46          if (err == -1)
47          {
48                  perror("bind");
49                  goto release_listen;
50          }
51
52          err = listen(server_sock_listen, 0);
53          if (err == -1)
54          {
55                  perror("listen");
56                  goto release_listen;
57          }
58
59          for (;;)
60          {
61                  server_sock_data = accept(server_sock_listen, NULL, NULL);
62                  if (server_sock_data == -1)
63                  {
64                          perror("accept");
65                          goto release_listen;
66                  }
67
68                  for (;;)
69                  {
70                          memset(recv_msg, 0, sizeof(recv_msg));
71                          err = recv(server_sock_data, recv_msg, sizeof(recv_msg), 0);
72                          if (err == -1)
73                          {
```

```
                                perror("recv");
                                goto release_data;
                        }

                        printf("Recv: %s", recv_msg);

                        err = send(server_sock_data, recv_msg, strlen(recv_msg), 0);
                        if (err == -1)
                        {
                                perror("send");
                                goto release_data;
                        }
                        printf("Sent: %s", recv_msg);

                        if (strncmp(recv_msg, "bye", 3) == 0)
                        {
                                break;
                        }
                }
        release_data:
                close(server_sock_data);
        }
        return 0;
release_listen:
        close(server_sock_listen);
        return -1;
}
```

代码 10: 服务器（并发）

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

int server_sock_listen, server_sock_data;

void release_socket(int signo)
{
        close(server_sock_listen);
        close(server_sock_data);
        exit(-1);
}

int main(int argc, char *argv[])
{
        signal(SIGINT, release_socket);

        struct sockaddr_in server_addr;
```

```
26          char recv_msg[255];
27
28          if (argc < 3)
29          {
30                  printf("Usage: %s <ip> <port>", argv[0]);
31          }
32
33          server_sock_listen = socket(AF_INET, SOCK_STREAM, 0);
34          if (server_sock_listen == -1)
35          {
36                  perror("socket");
37                  return -1;
38          }
39
40          server_addr.sin_family = AF_INET;
41          server_addr.sin_port = htons(atoi(argv[2]));
42  //      server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
43          inet_aton(argv[1], &server_addr.sin_addr);
44          memset(&server_addr.sin_zero, 0, sizeof(server_addr.sin_zero));
45
46          long err = bind(server_sock_listen, (struct sockaddr *)&server_addr, sizeof(server_addr));
47          if (err == -1)
48          {
49                  perror("bind");
50                  goto release_listen;
51          }
52
53          err = listen(server_sock_listen, 5);
54          if (err == -1)
55          {
56                  perror("listen");
57                  goto release_listen;
58          }
59
60          struct sockaddr_in client_addr;
61          memset(&client_addr, 0, sizeof(client_addr));
62          socklen_t cli_len;
63          cli_len = sizeof(client_addr);
64
65          for (;;)
66          {
67                  server_sock_data = accept(server_sock_listen, (struct sockaddr *)&client_addr, &
                        cli_len);
68                  if (server_sock_data == -1)
69                  {
70                          perror("accept");
71                          goto release_listen;
72                  }
73
74                  pid_t pid = fork();
75                  if (pid < 0)
76                  {
77                          perror("fork");
78                  }
79                  else if (pid == 0)
```

```
80                {
81                        printf("Connect client %s:%d\n",
82                                   inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
83                        for (;;)
84                        {
85                                memset(recv_msg, 0, sizeof(recv_msg));
86                                err = recv(server_sock_data, recv_msg, sizeof(recv_msg), 0);
87                                if (err == -1)
88                                {
89                                        perror("recv");
90                                        goto release_data;
91                                }
92
93                                printf("Recv from %s:%d: %s",
94                                           inet_ntoa(client_addr.sin_addr),
95                                           ntohs(client_addr.sin_port), recv_msg);
96
97                                err = send(server_sock_data, recv_msg, strlen(recv_msg), 0);
98                                if (err == -1)
99                                {
100                                       perror("send");
101                                       goto release_data;
102                               }
103                               printf("Sent to %s:%d: %s",
104                                          inet_ntoa(client_addr.sin_addr),
105                                          ntohs(client_addr.sin_port), recv_msg);
106
107                               if (strncmp(recv_msg, "bye", 3) == 0)
108                               {
109                                       close(server_sock_data);
110                                       close(server_sock_listen);
111                                       return 0;
112                               }
113                       }
114               }
115               else
116               {
117                       close(server_sock_data);
118                       continue;
119               }
120       }
121
122       close(server_sock_data);
123       close(server_sock_listen);
124       return 0;
125
126release_data:
127       close(server_sock_data);
128release_listen:
129       close(server_sock_listen);
130       return -1;
131}
```