

Ch16-More on Separation of Input/Output Stream

Intro CH 16.

Instructor: Limei Peng
Dept. of CSE, KNU

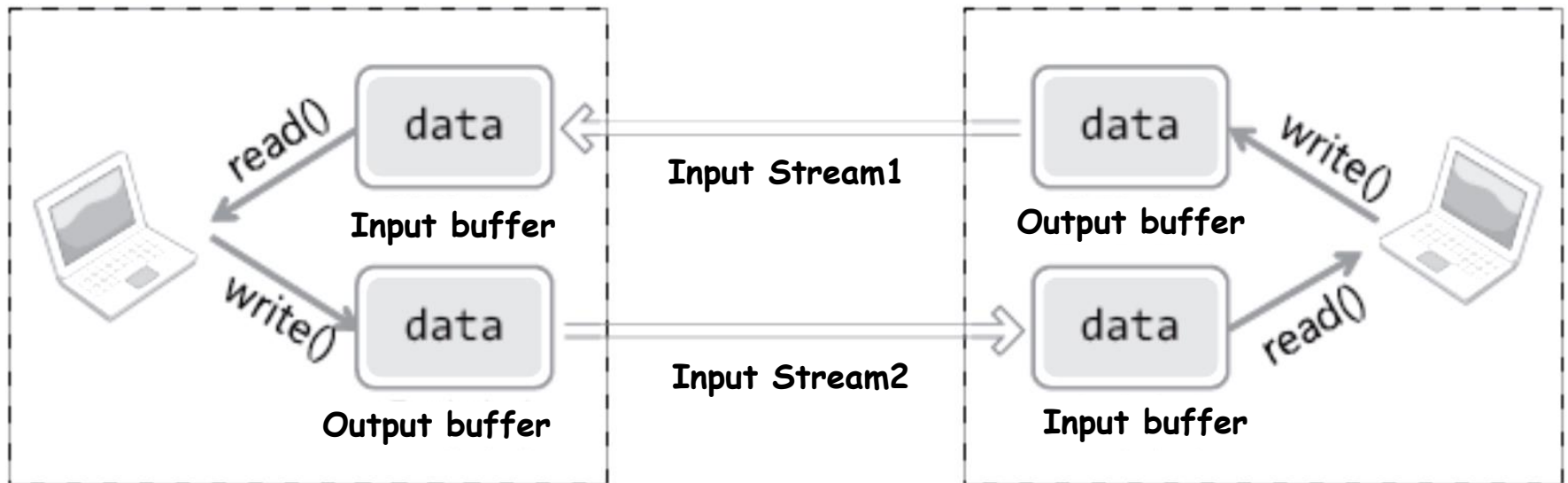
Outline

- ❑ Review on half-close: **shutdown()**
- ❑ Review on stream separation
 - Ch10: TCP routine separation, using **fork()**
 - Ch15: **fdopen()**, **FILE***
- ❑ Separated I/O streams with **half-close**

Review on Half-close: shutdown

□ Half-close

- A host may completed all the data transmission in its buffer, therefore, it intends to stop the output stream
- However, the connected partner host may still have data to transmit, and thus it does NOT want to close the output stream



Review on Half-close: **shutdown**(Cont.)

□ **shutdown** function

- Can only terminate one direction of data transfer

```
int shutdown(int sockfd , int howto)
```

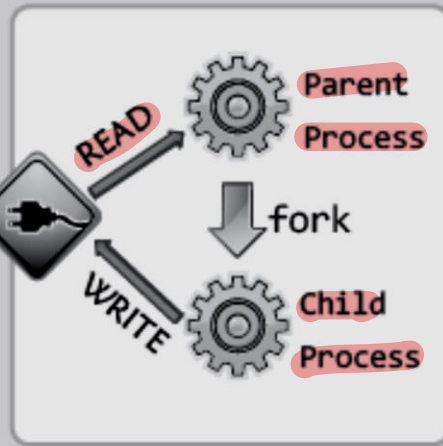
Returns: 0 if OK, -1 on error

- 2nd argument: *howto* (shutdown mode)
 - **SHUT_RD**: close input(read) stream
 - **SHUT_WR**: close output(write) stream
 - **SHUT_RDWR**: close input/output stream

Review on Separated I/O of TCP routine (Ch10)

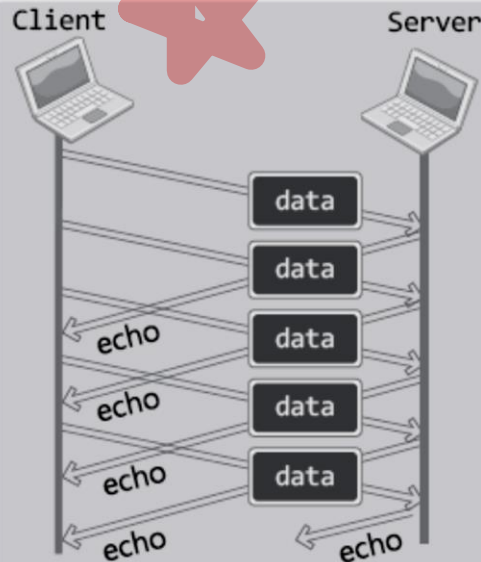
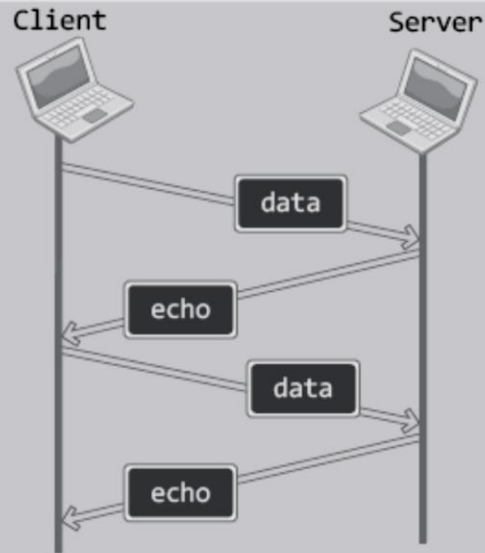


에코 서버



에코 클라이언트

Sockets support bidirectional communication. Therefore, as shown in the figure on the left, we can use two processes to take charge of the input and output operations respectively and separate them.



다중 동시성 가능

If the **input** and **output** stream is separated, it means the input and output operations can be proceeded simultaneously.

Review on Separated I/O of TCP routine (Ch10)

예제 echo_mpclient.c의 일부

```
if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))==-1)
    error_handling("connect() error!");
```

```
pid=fork();
```

```
if(pid==0)
```

```
    write_routine(sock, buf);
```

```
else
```

```
    read_routine(sock, buf);
```

입력을 담당하는 함수와 출력을 담당하는 함수를 구분 지어 정의했기 때문에, 구현의 용이성에도 좋은 점수를 줄 수 있다.

물론, 인터랙티브 방식의 데이터 송수신을 진행하는 경우에는 이러한 분할이 큰 의미를 부여하지 못한다. 즉, 이러한 형태의 구현이 어울리는 상황이 있고, 또 어울리지 않는 상황도 있다.

```
void read_routine(int sock, char *buf)
{
    while(1)
    {
        int str_len=read(sock, buf, BUF_SIZE);
        if(str_len==0)
            return;

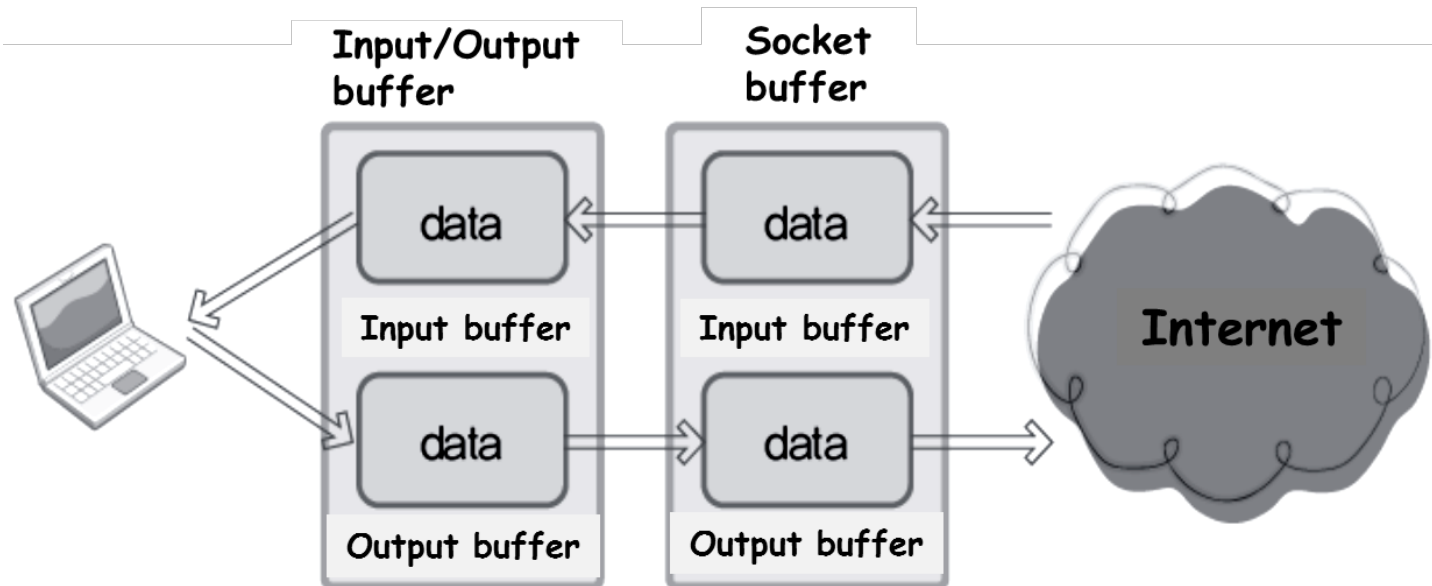
        buf[str_len]=0;
        printf("Message from server: %s", buf);
    }
}

void write_routine(int sock, char *buf)
{
    while(1)
    {
        fgets(buf, BUF_SIZE, stdin);
        if(!strcmp(buf, "q\n") || !strcmp(buf, "Q\n"))
        {
            shutdown(sock, SHUT_WR);
            return;
        }
        write(sock, buf, strlen(buf));
    }
}
```

Review on Separated I/O stream (Ch15)

□ Features of standard I/O

- Two types of buffers: socket buffer & I/O buffer
 - Socket buffer: for TCP reliability
 - Input/output buffer: for performance improvement, the more data, the better performance: transmission in bundle.
- I/O buffer
 - Separated I/O stream transmissions



스트림 분리의 이점

Chapter 10에서 설명한 스트림 분리의 목적

- 입력루틴(코드)과 출력루틴의 독립을 통한 구현의 편의성 증대
- 입력에 상관없이 출력이 가능하게 함으로 인해서 속도의 향상 기대

Chapter 10에서 설명한 스트림의 분리는 멀티 프로세스 기반의 분리였다.

Chapter 15에서 보인 스트림 분리의 이점

- FILE 포인터는 읽기모드와 쓰기모드를 구분해야 하므로,
- 읽기모드와 쓰기모드의 구분을 통한 구현의 편의성 증대
- 입력버퍼와 출력버퍼를 구분함으로 인한 버퍼링 기능의 향상

Chapter 15에서 보인 스트림의 분리는 FILE 구조체 포인터 기반의 분리였다.

Chapter 10에서 보인 스트림의 분리와 Chapter 15에서 보인 스트림의 분리는 그 방법에 있어서 차이점을 보이고, 이로 인해서 기대하는 장점에도 차이가 있다.



스트림 분리 이후의 EOF에 대한 문제점

Chapter 07의 복습

- half-close shutdown(sock, SHUT_WR)
- 출력 스트림에 대해서 half-close 진행 시 EOF 전달

writefp를 대상으로 fclose 함수를 호출하면 half-close가 진행될까?

```
readfp=fdopen(clnt_sock, "r");
writefp=fdopen(clnt_sock, "w");
fputs("FROM SERVER: Hi~ client? \n", writefp);
fputs("I love all of the world \n", writefp);
fputs("You are awesome! \n", writefp);
fflush(writefp);

fclose(writefp);
fgets(buf, sizeof(buf), readfp);
```

하나의 소켓을 대상으로 입력용 그리고 출력용 FILE 구조체 포인터를 얻었다 해도, 이 중 하나를 대상으로 fclose 함수를 호출하면, half-close가 아닌, 완전종료가 진행된다. 따라서, 위의 코드에서 마지막 행의 fgets 함수 호출은 성공하지 못한다.

Problems with EOF in stream separation

: 전송이 끝난 이후 EOF에 대한 문제

□ Half-close: **shutdown** (sock, SHUT_WR)

- Ch10: no problem

- Ch15: problem?

- We close file pointers using **fclose()**, aren't we?
- If **EOF** is transmitted, we expect that we can receive data but cannot transmit data, don't we? Isn't this concept very similar to half-close? → are they the same?

Example#1: **sep_serv.c** (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[]){
    int serv_sock, clnt_sock;
    FILE *readfp, *writefp;

    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t clnt_adr_sz;
    char buf[BUF_SIZE] = {0, };
    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family = AF_INET;
    serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_adr.sin_port = htons(atoi(argv[1]));

    bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
    listen(serv_sock, 5);
    clnt_adr_sz = sizeof(clnt_adr);
    clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);

    readfp = fdopen(clnt_sock, "r");
    writefp = fdopen(clnt_sock, "w");
```

Example#1: **sep_serv.c** (2/3)

```
fputs("FROM SERVER: Hi~ client? \n", writefp);  
fputs("I love all of the world \n", writefp);  
fputs("You are awesome! \n", writefp);  
fflush(writefp);
```

```
fclose(writefp);  
fgets(buf, sizeof(buf), readfp);  
fputs(buf, stdout);  
fclose(readfp);  
return 0;
```

```
}
```

Example#1: sep clnt.c (3/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[]){
    int sock;
    FILE *readfp, *writefp;

    struct sockaddr_in serv_adr;
    socklen_t clnt_adr_sz;
    char buf[BUF_SIZE] = {0, };
    sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family = AF_INET;
    serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
    serv_adr.sin_port = htons(atoi(argv[2]));

    connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));

    readfp = fdopen(sock, "r");
    writefp = fdopen(sock, "w");
    while(1){
        if(fgets(buf, sizeof(buf), readfp) == NULL)
            break;
        fputs(buf, stdout);
        fflush(stdout);
    }
    fputs("FROM CLIENT: Thank you! \n", writefp);
    fflush(writefp);

    fclose(writefp);
    fclose(readfp);
    return 0;
}
```

File Edit View Search Terminal Help

```
np2019@ubuntu:~/NP$ ./sep_serv 1234
```

```
np2019@ubuntu:~/NP$ ./sep_clnt 127.0.0.1 1234
FROM SERVER: Hi~ client?
I love all of the world
You are awesome!
```

Not echoed. Why?

스트림 종료 시 half-close가 진행되지 않는 이유

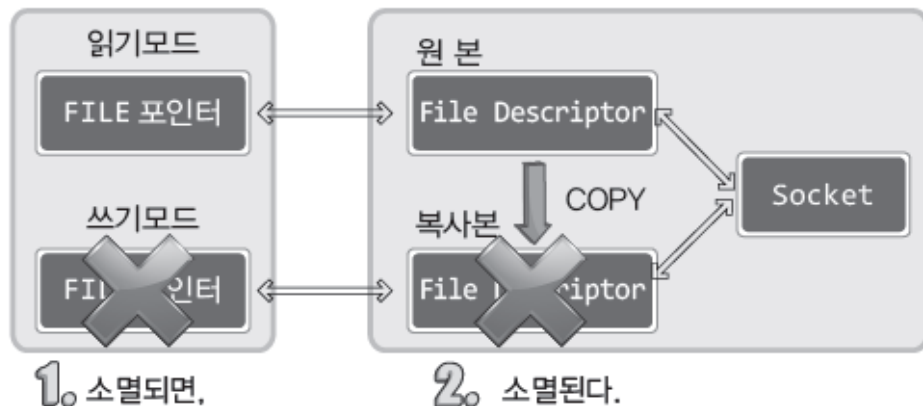
1. 둘 중 하나가 소멸되면,



2. 소멸되고,

3. 소멸된다.

왼쪽 그림과 같이 하나의 파일 디스크립터를 대상으로 FILE 포인터가 생성되었으니, FILE 포인터가 종료되면, 연결된 FILE 디스크립터도 종료된다.



왼쪽 그림과 같이 파일 디스크립터를 복사한 다음에 각각의 파일 디스크립터를 대상으로 FILE 포인터를 만들면, FILE 포인터 소멸 시 해당 파일 포인터에 연결된 파일 디스크립터만 소멸된다.

하지만 위의 경우에도 half-close는 진행되지 않는다. 왜? 여전히 하나의 FILE 디스크립터가 남아있고, 이를 이용해서 입출력이 가능해야 하기 때문에!

파일 디스크립터의 복사

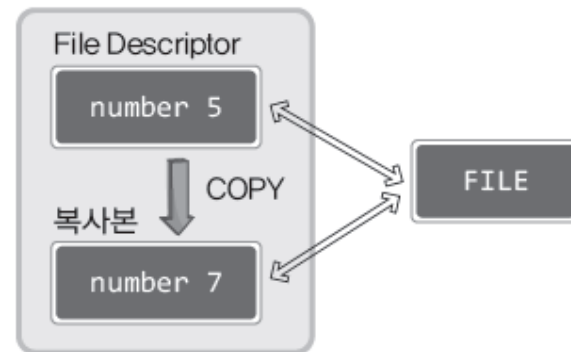
```
#include <unistd.h>
```

```
int dup(int fildes);
```

```
int dup2(int fildes, int fildes2);
```

→ 성공 시 복사된 파일 디스크립터, 실패 시 -1 반환

- fildes 복사할 파일 디스크립터 전달.
- fildes2 명시적으로 지정할 파일 디스크립터의 정수 값 전달.



```

cfd1=dup(1);
cfd2=dup2(cfd1, 7);
printf("fd1=%d, fd2=%d \n", cfd1, cfd2);
write(cfd1, str1, sizeof(str1));
write(cfd2, str2, sizeof(str2));

close(cfd1);
close(cfd2);
write(1, str1, sizeof(str1));
close(1);
write(1, str2, sizeof(str2));
    
```

dup.c의 일부

왼쪽 코드에서 총 두 개의 파일 디스크립터를 복사한다. 그리고 복사된 파일 디스크립터까지 모두 종료를 한다. 때문에 **마지막 행에 존재하는 write 함수의 호출은 성공하지 못한다.**

```

root@my_linux:/tcpip# gcc dup.c -o dup
root@my_linux:/tcpip# ./dup
fd1=3, fd2=7
Hi~
It's nice day~
Hi~
    
```

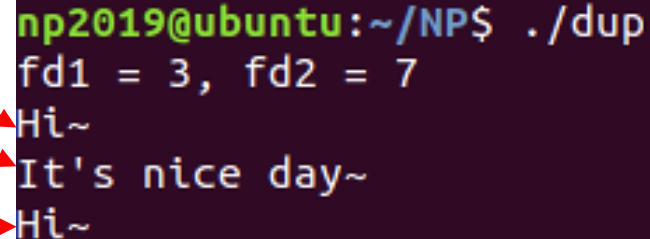
실행 결과

Example#2: dup.c (1/1)

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

```
int main(int argc, char *argv[]){
    int cfd1, cfd2;
    char str1[] = "Hi~ \n";
    char str2[] = "It's nice day~ \n";
    cfd1 = dup(1);
    cfd2 = dup2(cfd1, 7);
    printf("fd1 = %d, fd2 = %d \n", cfd1, cfd2);
    write(cfd1, str1, sizeof(str1));
    write(cfd2, str2, sizeof(str2));

    close(cfd1); close(cfd2);
    write(1, str1, sizeof(str1));
    close(1);
    write(1, str2, sizeof(str2));
    return 0;
}
```



```
np2019@ubuntu:~/NP$ ./dup
fd1 = 3, fd2 = 7
Hi~
It's nice day~
Hi~
```


Separate streams after copying file descriptors

파일 디스크립터 복사 후 스트림 분리

Example#3:

sep_serv2.c (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define BUF_SIZE 1024

int main(int argc, char *argv[]){
    int serv_sock, clnt_sock;
    FILE *readfp, *writefp;

    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t clnt_adr_sz;
    char buf[BUF_SIZE] = {0, };
    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family = AF_INET;
    serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_adr.sin_port = htons(atoi(argv[1]));

    bind(serv_sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));
    listen(serv_sock, 5);
    clnt_adr_sz = sizeof(clnt_adr);
    clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr, &clnt_adr_sz);

    readfp = fdopen(clnt_sock, "r");
    writefp = fdopen(dup(clnt_sock), "w");
```

dup 함수 호출을 통해서 복사된 파일 디스크립터를 대상으로
파일 구조체 포인터 형성

Example#3: `sep_serv2.c` (2/3)

```
fputs("FROM SERVER: Hi~ client? \n", writefp);  
fputs("I love all of the world \n", writefp);  
fputs("You are awesome! \n", writefp);  
fflush(writefp);  
shutdown(fileno(writefp), SHUT_WR);  
fclose(writefp);  
fgets(buf, sizeof(buf), readfp);  
fputs(buf, stdout);  
fclose(readfp);  
return 0;
```

파일 디스크립터로 변환해서 shutdown 함수를 호출한다
이제 half-close가 진행되고
이로 인해 상대방 영역으로 EOF 전달됨

```
np2019@ubuntu:~/NP$ ./sep_serv2 1234  
FROM CLIENT: Thank you!
```

Conclusion: no matter how many file descriptors are copied, we need to call **shutdown()** to realize half-close to transmit EOF.

Example#3: **sep_clint.c** (3/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[]){
    int sock;
    FILE *readfp, *writefp;

    struct sockaddr_in serv_adr;
    socklen_t clnt_adr_sz;
    char buf[BUF_SIZE] = {0, };
    sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family = AF_INET;
    serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
    serv_adr.sin_port = htons(atoi(argv[2]));

    connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr));

    readfp = fdopen(sock, "r");
    writefp = fdopen(sock, "w");
    while(1){
        if(fgets(buf, sizeof(buf), readfp) == NULL)
            break;
        fputs(buf, stdout);
        fflush(stdout);
    }
    fputs("FROM CLIENT: Thank you! \n", writefp);
    fflush(writefp);

    fclose(writefp);
    fclose(readfp);
    return 0;
}
```