

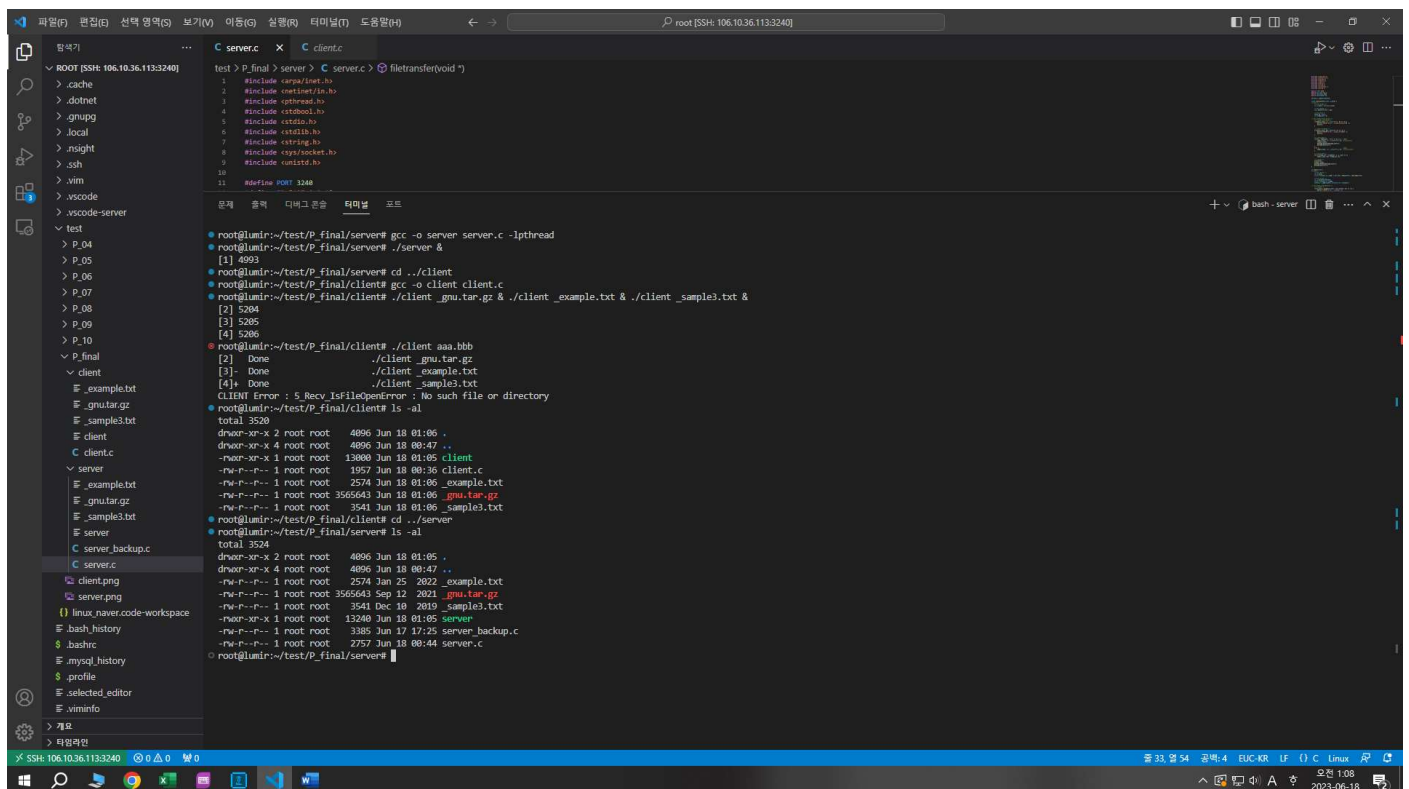
'23-1 리눅스 프로그래밍 기말 프로젝트 결과보고서

학번/이름 : 201818716 김용현

0. 요약

| | |
|-------------------------|---------------------------------|
| 현재 개발을 <u>완성한</u> 단계는? | 3단계 |
| 개발을 완료하지 못한 단계는? | 없음 |
| 개발한 프로그램을 실행한 시스템의 환경은? | Ubuntu 18.04.5 LTS & VScode SSH |

1. 실행 스크린샷



#. gcc -o server server.c -lpthread

-> server.c를 컴파일하여 server라는 이름의 실행파일 생성. (이때, -lpthread 옵션으로 링크를 해주었다.)

#. ./server &

-> server 실행파일을 후면 처리하여 실행.

#. cd ../client

-> client 디렉터리로 이동.

#. gcc -o client client.c

-> client.c를 컴파일하여 client라는 이름의 실행파일 생성.

#. ./client _gnu.tar.gz & ./client _example.txt & ./client _sample3.txt &

-> client 실행파일에 대해 3개의 요청을 동시처리. (멀티쓰레드 동작 확인 용도.) (3개의 파일은 server 디렉터리에 존재하는 파일.)

-> 3개의 요청 처리 이후, _gnu.tar.gz _example.txt _sample3.txt 3개의 파일이 client 디렉터리로 복사(전송)됨.

```
#. ./client aaa.bbb
```

-> 존재하지 않는 파일(aaa.bbb)에 대한 실행 요청.

-> 실행 요청 처리 이후, 'CLIENT Error : 5_Recv_IsFileOpenError : No such file or directory' 라는 내용의 실행 오류 메시지 출력.

(client 쪽에서 출력된 메시지.)

```
#. ~/test/P_final/client# ls -al
```

-> client 디렉터리 내에서의 'ls -al' 명령 실행.

```
#. cd ../server
```

-> server 디렉터리로 이동.

```
#. ~/test/P_final/server# ls -al
```

-> server 디렉터리 내에서의 'ls -al' 명령 실행.

-> server와 client 디렉터리에서 각각, _gnu.tar.gz _example.txt _sample3.txt 3개의 파일이 모두 일치함을 알 수 있다.

-> 즉, 파일 복사(전송)이 정상적으로 진행되었음을 알 수 있다.

2. 개발한 코드 스크린샷 및 설명 : 전체 로직은 주석에 잘 정리되어 있습니다.

[server.c]

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <pthread.h>
4  #include <stdbool.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <sys/socket.h>
9  #include <unistd.h>
10
11 #define PORT 3240
12 #define IP "127.0.0.1"
13 #define BUFFSIZE 1024
14 #define MAXTHREADS 16
15
16 pthread_t tid[MAXTHREADS];
17
18 void* filetransfer(void* _c_sock) {
19     /* Init */
20     /* Init_Socket */
21     int c_sock = *(int*)_c_sock;
22
23     /* Init_Buffer */
24     char buf[BUFFSIZE] = {0};
25
26     /* Init_File */
27     FILE* fp = NULL;
28     int fread_cnt = 0;
29
30     /* IPC_Socket_FileTransfer */
31     /* 1_Send_IsConnected */
32     if(send(c_sock, "t", sizeof("t"), 0) == -1) {
33         perror("SERVER Error : 1_Send_IsConnected ");
34         exit(1);
35     }
36
37     /* 2_Recv_FileName */
38     if(recv(c_sock, buf, BUFFSIZE, 0) == -1) {
39         perror("SERVER Error : 2_Recv_FileName ");
40         exit(1);
41     }
42
43     /* 3_File_Open */
44     if((fp = fopen(buf, "rb")) == NULL) { //실패
45         send(c_sock, "f", sizeof("f"), 0); //false(fail)
46         close(c_sock);
47         pthread_detach(pthread_self());
48         pthread_exit(NULL);
49     }
50     else { //성공
51         send(c_sock, "t", sizeof("t"), 0); //true(success)
52     }
53
54     /* 4_File_Send */
55     while((fread_cnt = fread(buf, 1, 1, fp)) == 1)
56         send(c_sock, buf, fread_cnt, 0);
57
58     /* 5_Close */
59     fclose(fp);
60     close(c_sock);
61     pthread_detach(pthread_self());
62     pthread_exit(NULL);
63 }
64
```

```

64
65 int main(void) {
66     /* Init */
67     /* Init_Server */
68     int s_sock;
69     struct sockaddr_in s_addr = {AF_INET, htons(PORT), inet_addr(IP)};
70
71     /* Init_Client */
72     int c_sock[MAXTHREADS];
73     struct sockaddr_in c_addr;
74     socklen_t c_addr_size = sizeof(struct sockaddr);
75
76     /* IPC_Socket_StartRoutine */
77     /* 1_Socket */
78     if((s_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
79         perror("SERVER Error : 1_Socket ");
80         exit(1);
81     }
82     int option = 1; setsockopt(s_sock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));
83
84     /* 2_Bind */
85     if(bind(s_sock, (struct sockaddr *)&s_addr, sizeof(struct sockaddr)) == -1) {
86         perror("SERVER Error : 2_Bind ");
87         exit(1);
88     }
89
90     /* 3_Listen */
91     listen(s_sock, 1);
92
93     /* IPC_Socket_AcceptRoutine */
94     for(int i = 0; ; i = (i + 1) % MAXTHREADS) {
95         /* 1_Accept */
96         if((c_sock[i] = accept(s_sock, (struct sockaddr *)&c_addr, &c_addr_size)) == -1) {
97             perror("SERVER Error : 1_Accept ");
98             exit(1);
99         }
100
101         /* 2_Threads */
102         if(pthread_create(&tid[i], NULL, filetransfer, &c_sock[i]) != 0) {
103             perror("SERVER Error : 2_Threads ");
104             exit(1);
105         }
106     }
107
108     /* IPC_Socket_CloseRoutine */
109     /* 1_Close */
110     close(s_sock);
111
112     /* 2_Return */
113     return 0;
114 }

```

[매크로]

- #. PORT, IP : IPC-Socket에서 사용하기 위하여, Port번호와 IP주소를 매크로 상수로 지정하였다. (이는 client.c 에서도 동일하게 사용된다.)
- #. BUFSIZE : BUFFER의 SIZE를 1024Byte로 지정하기 위하여 선언하였다.
- #. MAXTHREADS : 서버가 한번에 수용 가능한 쓰레드의 수를 16개로 제한하기 위하여 선언하였다.

[전역변수]

#. pthread_t tid[MAXTHREADS]

→ 최대 16개의, pthread_create를 통해 만들어진 쓰레드들을 관리하기 위한, pthread_t 타입의 tid 배열을 선언. 이때, 모든 위치에서 해당 변수에 접근할 수 있게 하기 위해 전역변수로 선언하였다. 전역변수로 선언할 경우, Data영역에 배치가 되어, 다른 모든 쓰레드들도 여기에 접근하여 데이터를 공유하며 쓸 수 있게 된다. (단, 일반적인 경우, 동시화 문제가 발생할 수 있음에 유의하자.)

```
[void* filetransfer(void* _c_sock)]
```

#. 멀티 쓰레드를 통한 파일 전송을 위해 만들어진 함수. main 함수에서 accept를 통해 만들어진 c_sock에 대한 정보를 인자로 전달받는다.

/* Init */ : 변수 초기화를 진행한다.

/* Init_Socket */

#. c_sock : 함수의 인자로 전달받은 _c_sock 값을 int형으로 형변환하여 저장한다.

/* Init_Buffer */

#. buf[BUFFSIZE] : 파일 전송 시, 버퍼 용도로 사용하기 위한 char형 배열.

/* Init_File */

#. fp : FILE 구조체. 파일을 다루기 위한 용도로 선언.

#. fread_cnt : fread시, 읽어들이는 데이터의 수를 count 하기 위한 용도로 선언.

/* IPC_Socket_FileTransfer */ : 파일 전송(복사)에 대한 실질적인 역할을 담당하는 코드부분. (코드의 핵심파트)

/* 1_Send_IsConnected */

#. server가 먼저 client쪽으로 send를 보내어, 서버와의 연결이 정상적으로 체결되었다는 것을 클라이언트에게 알려준다. 이는 IPC-Socket에서 Signal과 같은 역할을 수행한다. 이때, send를 서버가 먼저 진행하는 이유는, listen(server) -> connect(client) -> accept(server) 순서로 연결이 진행되기 때문에, server쪽에서 먼저 소켓 연결이 완료되었다는 사실을 알게 되기 때문이다. (client 쪽에서 먼저 send를 하게 되면, 이는 소켓 연결이 되었는지 안되었는지 여부를 모르는 상태에서 send를 하는 것이기 때문에 예상치 못한 문제가 발생할 수 있다.)

#. 이때, 클라이언트는 /* IPC_Socket_StartRoutine */ - /* 3_Recv_IsConnected */에 해당하는 라인에서 recv를 통해 send가 도착할 때까지 대기하고 있게 된다.

#. 추가적으로, send(c_sock, "t", sizeof("t"), 0)을 하게되면, server->client쪽 버퍼에 't'와 '\0'이라는 2개의 문자가 넘어가게 된다. (나는 't'이라는 char 문자 하나만 보내고 싶었는데, 't'과 '\0' 2개의 문자가 넘어가는 상황.) 따라서, client쪽에서 1byte의 문자만 receive하는 방식으로 recv함수를 구현하면, 't'은 client에서 읽히지만, '\0'은 server->client쪽 버퍼에 남아 문제가 생긴다.

#. 함수에 오류가 발생한 경우, 오류 처리 루틴을 통해 이를 알려준다.

/* 2_Recv_FileName */

#. 클라이언트가 /* IPC_Socket_StartRoutine */ - /* 4_Send_FileName */에서 send를 통해 보내온 argv[1]에 해당하는 값을 서버의 buf에 저장한다. 이는 전송할 경로명(파일명)을 나타낸다. (뒤쪽에서 여기 buf에 저장된 값을 통해 fopen을 수행할 것이다.)

#. 함수에 오류가 발생한 경우, 오류 처리 루틴을 통해 이를 알려준다.

/* 3_File_Open */

#. /* 2_Recv_FileName */에서, recv를 통해 전달받은 경로명(파일명)이 실제로 존재하는지 여부를 판단한다. 이후, 해당 경로명을 fopen을 통해 연결을 시도한다. 이때, 연결이 실패할 경우 해당하는 경로명이 존재하지 않는다는 뜻이므로, 실패했다는 의미의 "f"라는 문자열을 client쪽으로 보낸다. (이때, client는 /* IPC_Socket_StartRoutine */ - /* 5_Recv_IsFileOpenError */에서 recv를 통해 여러 여부에 대한 문자열이 전달되기를 대기하고 있게 된다.) 이외의 경우 연결에 성공했다는 것이고, 해당하는 경로명(파일명)이 존재한다는 뜻이므로, "t"라는 문자열을 client쪽으로 보내고 파일 전송을 정상적으로 실행한다.

#. 텍스트 모드가 아닌, 바이너리 모드로 파일을 열어 전송한다.

#. 연결에 실패할 경우에도, 서버는 종료되지 않고 계속 실행상태에 놓여있게 된다. 단지 쓰레드만이 종료될 뿐이다.

/* 4_File_Send */

#. fread와 send의 조합을 통해 파일을 client쪽으로 전송한다. 파일의 모든 내용이 정상적으로 전부 전송된다.

/* 5_Close */

#. 마무리 작업에 해당하는 부분. fclose를 통해 파일 포인터를 닫고, close를 통해 소켓을 닫으며, pthread_detach()를 통해 현재 쓰레드를 분리한 후, pthread_exit()를 통해 쓰레드를 종료한다. 이때, main 함수에서 pthread_join()을 통한 쓰레드의 종료를 대기하지 않기 위해 pthread_detach()를 사용하였다.

#. 이때, pthread_detach()는 쓰레드를 종료시키는 함수가 아니다. 단지, 쓰레드를 '분리'된 상태로 만드는 역할을 할 뿐이다.

[main]

```
/* Init */ : IPC-Socket 통신을 위해 필요한 변수들에 대해 선언 및 초기화를 진행한다.

/* Init_Server */

#. s_sock, s_addr : 서버측과 관련된 변수들을 할당.

/* Init_Client */

#. c_sock, c_addr, c_addr_size : 클라이언트측과 관련된 변수들을 할당.

#. c_sock의 경우, MAXTHREADS 개수만큼 할당하며, pthread_t tid와 index를 맞추어 사용할 수 있게 하였다.

/* IPC_Socket_StartRoutine */

#. Socket, Bind, Listen에 대한 수행. 이는, IPC-Socket을 위한 서버측의 가장 기본적인 세팅이다.

#. 해당 부분은 반복 및 쓰레드와 연관성이 없으며, 서버측에서 1회만 수행되면 된다.

#. 함수에 오류가 발생한 경우, 오류 처리 루틴을 통해 이를 알려준다.

/* IPC_Socket_AcceptRoutine */

#. for문에 반복문 탈출 조건을 없애, 무한루프를 돌 수 있게 만들었다. 이때, i = (i + 1) % MAXTHREADS 라는 증가조건을 통해, 클라이언트가 아무리 많이 들어온다 하더라도, index 범위를 초과하는 일이 발생하지 않게 만들었다.

/* 1_Accept */ & /* 2_Threads */

#. Accept은 Main 함수에서 진행된다. 서버는 Accept을 수행한 후, 해당 소켓 파일 디스크립터를 쓰레드로 넘겨주고 다시 Accept에서 대기한다. 무한루프 안에서 이러한 과정이 일어나기 때문에, 클라이언트의 요청을 무한히 받아들여 동시에 처리할 수 있게 해준다.

#. 결국 Main 함수에서의 서버의 역할은, 클라이언트의 connect 요청을 받아, accept를 진행하고, 쓰레드를 만들어 해당 파일전송요청을 쓰레드에게 넘겨준 뒤 다시 connect에서 대기하는 일의 무한 반복이다.

/* IPC_Socket_CloseRoutine */

#. 소켓 close와 main 함수 return을 진행하는 마무리 과정이다.
```

[client.c]



```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdbool.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/socket.h>
8  #include <unistd.h>
9
10 #define PORT 3240
11 #define IP "127.0.0.1"
12 #define BUFSIZE 2
13
14 int main(int argc, char* argv[]) {
15     /* Error */
16     if(argc != 2) {
17         printf("Usage : ./client file\n");
18         exit(1);
19     }
20
21     /* Init */
22     /* Init_Server */
23     struct sockaddr_in s_addr = {AF_INET, htons(PORT), inet_addr(IP)};
24
25     /* Init_Client */
26     int c_sock;
27
28     /* Init_Buffer */
29     char buf[BUFSIZE];
30
31     /* Init_File */
32     FILE* fp = NULL;
33 }
```

```

33
34  /* IPC_Socket_StartRoutine */
35      /* 1_Socket */
36      if((c_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
37          perror("CLIENT Error : 1_Socket ");
38          exit(1);
39      }
40
41      /* 2_Connect */
42      if(connect(c_sock, (struct sockaddr *)&s_addr, sizeof(struct sockaddr)) == -1) {
43          perror("CLIENT Error : 2_Connect ");
44          exit(1);
45      }
46
47      /* 3_Recv_IsConnected */
48      if(recv(c_sock, buf, sizeof(buf), 0) == -1) {
49          perror("CLIENT Error : 3_Recv_IsConnected ");
50          exit(1);
51      }
52
53      /* 4_Send_FileName */
54      if(send(c_sock, argv[1], strlen(argv[1]) + 1, 0) == -1) {
55          perror("CLIENT Error : 4_Send_FileName ");
56          exit(1);
57      }
58
59      /* 5_Recv_IsFileOpenError */
60      if(recv(c_sock, buf, sizeof(buf), 0) == -1) {
61          perror("CLIENT Error : 5_Recv_IsFileOpenError ");
62          exit(1);
63      }
64
65      if(buf[0] == 'f') {
66          printf("CLIENT Error : 5_Recv_IsFileOpenError : No such file or directory\n");
67          exit(1);
68      }
69
70  /* IPC_Socket_FileTransfer */
71      /* 1_File_Open */
72      if((fp = fopen(argv[1], "wb")) == NULL) {
73          perror("CLIENT Error : 1_File_Open ");
74          exit(1);
75      }
76
77      /* 2_File_Recv */
78      while(recv(c_sock, buf, 1, 0) != 0)
79          fwrite(buf, 1, 1, fp);
80
81  /* IPC_Socket_CloseRoutine */
82      /* 1_Close */
83      fclose(fp);
84      close(c_sock);
85
86      /* 2_Return */
87      return 0;
88  }

```


[매크로]

#. PORT, IP : IPC-Socket에서 사용하기 위하여, Port번호와 IP주소를 매크로 상수로 지정하였다. (이는 server.c 에서도 동일하게 사용된다.)

#. BUFSIZE : BUFFER의 SIZE를 2Byte로 지정하기 위하여 선언하였다.

[main]

/* Error */

#. "Usage : ./client fileWn" 형식으로 main함수 인자값이 들어오지 않았을 경우, 오류를 출력.

/* Init */

#. server, client, buffer, file과 관련된 변수를 선언. 해당 변수들의 용도는 server.c에서 설명한 용도와 동일하다.

/* IPC_Socket_StartRoutine */

/* 1_Socket */

#. socket을 생성한다.

/* 2_Connect */

#. 서버측에 connect 요청을 진행한다. 이때, connect요청에 대한 서버측의 처리는 순차처리이다. 즉, a b c 3개의 프로세스가 서버에 순서대로 connect 요청을 진행하면, 서버측에서는 a b c 순서대로 일을 처리해준다. 즉, a먼저 accept하고 해당하는 쓰레드를 만들고, 그 다음에 b를 accept하고 해당하는 쓰레드를 만들고, 마지막으로 c를 accept하고 해당하는 쓰레드를 만드는 순서이다. 이때, 멀티 쓰레드 형식으로 서버를 구현하였으므로 실제 파일 전송에 대한 처리는 a b c 모두 동시에 발생할 수 있다. (단지, accept하고 thread를 만드는 순서만 순차적인 것이다.)

/* 3_Recv_IsConnected */

#. 서버측의 /* IPC_Socket_FileTransfer */ - /* 1_Send_IsConnected */ 에서 server가 먼저 client쪽으로 send를 보내어, 서버와의 연결이 정상적으로 체결되었다는 것을 클라이언트에게 알려준다. 이는, IPC-Socket에서 Signal과 같은 역할을 수행한다. 이때, send를 서버가 먼저 진행하는 이유는, listen(server) -> connect(client) -> accept(server) 순서로 연결이 진행되기 때문에, server쪽에서 먼저 소켓 연결이 완료되었다는 사실을 알게되기 때문이다. (client 쪽에서 먼저 send를 하게 되면, 이는 소켓 연결이 되었는지 안되었는지 여부를 모르는 상태에서 send를 하는 것이기 때문에 문제가 발생할 수 있다.)

/* 4_Send_FileName */

#. 서버는 클라이언트가 /* IPC_Socket_StartRoutine */ - /* 4_Send_FileName */ 에서 send를 통해 보내온 argv[1]에 해당하는 값을 서버의 buf에 저장한다. 이는 전송할 경로명(파일명)을 나타낸다. 서버 측에서는 클라이언트 측에서 경로명(파일명)이 send되기 전까지 recv에서 대기한다.

/* 5_Recv_IsFileOpenError */

#. 서버는 /* IPC_Socket_FileTransfer */ - /* 2_Recv_FileName */에서 앞서 전달받은 경로명(파일명)이 실제로 존재하는지 여부를 판단한다. 이후, 해당 경로명을 fopen을 통해 연결을 시도한다. 이때, 연결이 실패할 경우 해당하는 경로명이 존재하지 않는다는 뜻이므로, 실패했다는 의미의 "f"라는 문자열을 client쪽으로 보낸다. (이때, client는 /* IPC_Socket_StartRoutine */ - /* 5_Recv_IsFileOpenError */에서 recv를 통해 여러 여부에 대한 문자열이 전달되기를 대기하고 있게 된다.) 이외의 경우 연결에 성공했다는 것이고, 해당하는 경로명(파일명)이 존재한다는 뜻이므로, "t"라는 문자열을 client쪽으로 보내고 파일 전송을 정상적으로 실행한다. 즉, 서버측에서는 오직 "t" 또는 "f"에 해당하는 문자열을 클라이언트에게 보내게 되는데(이는 각각 파일이 존재함과 존재하지 않음을 나타낸다.), 만약 해당 문자열이 "f"라면 해당 파일이 서버측에 존재하지 않는다는 뜻이므로, 클라이언트는 오류메시지를 출력하고 프로그램을 종료한다.

/* IPC_Socket_FileTransfer */ : 실제로 파일에 대한 전송을 담당하는 부분.

/* 1_File_Open */

#. argv[1]의 인자로 전달된 문자열에 해당하는 파일을 새로 만든다. 이때, 서버측에서 바이너리 모드로 읽은 내용을 가지고 전송을 시도하므로, 클라이언트측도 마찬가지로 바이너리 모드로 파일을 fopen한다.

/* 2_File_Recv */

#. 서버측에서 보내는 파일의 내용을 recv와 fwrite의 조합으로 수신한다. 이렇게 수신한 파일의 결과는 원본 파일과 완전히 동일하다. (정상적으로 동작한다는 뜻.)

/* IPC_Socket_CloseRoutine */

#. 파일 포인터를 닫고, 소켓을 닫고, main 함수를 return 하여 종료시킨다.

3. 최종 개발을 실패한 내용 (있다면) : 없음