# 유닉스 프로그래밍 개인과제 12번

201619460 이성규

**개인과제 12-1)**

**코드:**

기본적인 동작은, 서버가 소켓을 열고, 클라이언트가 커넥트를 하여 메시지를 주고 받는 프로그램입니다.
먼저 서버 코드를 보겠습니다.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <sys/socket.h>
6   #include <netinet/in.h>
7   #include <arpa/inet.h>
8   #include <sys/types.h>
9   #include <sys/wait.h>
10
11  #define BUFFSIZE 4096
12  #define SERVERPORT 7799
13
14  int main(void) {
15      int i, s_sock, c_sock;
16      struct sockaddr_in server_addr, client_addr;
17      socklen_t c_addr_size;
18      char buf[BUFFSIZE] = {0};
19      char hello[] = "Hello~ I am Server!\n";
20      pid_t pid;
21      int wstatus;
22      ssize_t check; //QnA 내용처럼 send와 recv 반환값 검사용 변수
23
24      s_sock = socket(AF_INET, SOCK_STREAM, 0); // 소켓 열기
25
26      int option = 1;     //SO_REUSEADDR의 옵션 값을 TRUE로 : 프로그램 종료후 다시실행때 bind()에서 오류 무시하기위해 사용
27      setsockopt(s_sock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));
28
29      bzero(&server_addr,sizeof(server_addr)); //server_addr 초기화
30
31      server_addr.sin_family = AF_INET;
32      server_addr.sin_port = htons(SERVERPORT);
33      server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // 동작하는 컴퓨터 IP주소 자동 할당
34
35      if (bind(s_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
36          perror("[S] Can't bind a socket"); // 바인드 안된 경우 예외처리
37          exit(1);
38      }
```

서버코드는 소켓을 열고 ( 24 라인 ) 바인드를 합니다( 35 라인 ).
이때, IP주소는 INADDR_ARNY를 사용하여 동작하는 컴퓨터의 IP주소를 자동으로 할당 받습니다. ( 33 라인 )
위 코드에서 22 라인의 check변수는 send 와 recv 반환 값을 임시로 저장하여 검사하기 위한 변수입니다.

서버코드에서 바인드를 한 후, for문을 돌며 accept를 합니다. 만약 accept를 받게 되면 fork를 호출하여 자식프로세스를 생성하고 자식 프로세스에게 동작을 맡긴 뒤 다시 연결을 기다리게 됩니다.

```
40      listen(s_sock,1);
41      c_addr_size = sizeof(struct sockaddr);
42
43      for(i=0; i<3; i++) {
44          printf("[S] waiting for a client..#%02d\n",i);
45          c_sock = accept(s_sock, (struct sockaddr *) &client_addr, &c_addr_size);
46          if (c_sock == -1) {
47              perror("[S] Can't accept a connection"); //accept 예외처리
48              exit(1);
49          }
50
51          pid = fork();
52
53          if (pid < 0 ) // fork 에러
54          {
55              perror("[S] Fork Faild");
56              return 1;
```

아래의 코드가 자식 프로세스의 코드입니다.

```
57          } else if ( pid == 0 ) //자식 프로세스
58          {
59              close(s_sock);
60
61              printf("[S-%d] Connected: client IP addr=%s port=%d\n",getpid(), inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
62
63              //1. say hello to client
64              check = send(c_sock, hello, sizeof(hello)+1,0);
65              if ( (check == -1) || (check == 0) ) {
66                  printf("[S-%d] Can't send message\n",getpid());
67                  exit(1);
68              }
69
70              printf("[S-%d] I said Hello to Client!\n",getpid());
71
72              //2. recv msg from client
73              check = recv(c_sock,buf,BUFFSIZE, 0);
74              if ((check == -1) || (check == 0)) {
75                  printf("[S-%d] Can't receive message\n",getpid());
76                  exit(1);
77              }
78
79              printf("[S-%d] Client says: %s\n",getpid(), buf);
80
81              close(c_sock);
82
83              return 0;
```

화면상에 출력을 하는 주체가 부모 프로세스인지 자식 프로세스인지 구분을 하고자, 자식 프로세스는 getpid를 이용하여 자신의 pid번호를 앞쪽에 붙여서 출력을 하게 했습니다. ( 위 코드에서 printf("[S-%d] ....", getpid(), ....) 부분 )

fork로 만들어진 자식 프로세스에서는 일단 서버의 소켓을 닫았습니다.( 라인 59 )
자식 프로세스에서는 이제 클라이언트와만 전송을 하고 서버의 소켓은 필요가 없기 때문입니다.
서버 소켓은 자식 프로세스에서만 닫았기 때문에, 부모 프로세스에선 여전히 서버가 닫히지 않았습니다.
그 이후 자식 프로세스는 연결된 클라이언트의 IP와 포트를 출력하고, ( 라인 61 )
클라이언트에게 메시지를 보내게 됩니다. ( 라인 64 )
그 후, 자신이 메시지를 보냈단 것을 출력하고, ( 70 라인 ) 클라이언트로부터 메시지를 대기합니다. ( 73 라인)
그리고 마지막으로 클라이언트에게 메시지를 받게 된다면, 클라이언트로부터 받은 메시지를 출력하고 종료하게 됩니다.
( 79 라인 )

위 코드에서 send와 recv에 대한 예외처리로, LMS 질의응답을 참고하여 예전에 작성한 코드와 다르게 리턴 값이 -1인지 0인지 확인하였습니다.
( 관련 링크 : http://ieilmsold.jbnu.ac.kr/mod/ubboard/article.php?id=220436&bwid=88383 )

```
86          }else //부모 프로세스
87          {
88              printf("[S] I open the [S-%d] at #%02d\n",pid,i);
89              close(c_sock);
90          }
91
92      }
93      close(s_sock);
94
95      for(i=0; i<3; i++){
96          pid = wait(&wstatus);
97          printf("[S] My Child [S-%d] is finised with %d\n",pid,wstatus);
98      }
99
00      return 0;
01  }
```

accept이후 fork로 자식프로세스를 생성하고 부모 프로세스 코드에선,
만든 자식 프로세스의 pid를 출력하고, 자식 프로세스에게 넘겨준 클라이언트 소켓을 닫습니다. ( 89 라인 )
이는 위 자식프로세서에서 이야기한 바와 같습니다.
그 이후 for문 처음으로 돌아가 추가적인 연결을 받으며 위에서 설명한 동작을 총 3번 반복하게 됩니다.

그리고 for문이 끝나 3번의 accept 반복이 끝났다면, 서버소켓을 닫고,( 라인 93 )
for 문을 돌며 생성한 자식의 종료 상태 확인 및 출력을 하고 부모프로세스는 종료됩니다. ( 라인 95 )

그럼 이제 클라이언트 코드를 보겠습니다.

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

#define BUFFSIZE 4096
#define SERVERPORT 7799

int main(void) {
    int c_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t c_addr_size;
    char buf[BUFFSIZE] = {0};
    char hello[] = "Hi~ I am Client!!\n";
    ssize_t check; //QnA 내용처럼 send와 recv 반환값 검사용 변수

    c_sock = socket(AF_INET, SOCK_STREAM, 0); // 소켓열기

    bzero(&server_addr, sizeof(server_addr)); // server_addr 초기화

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);


    printf("[C] Connecting...\n");

    if (connect(c_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
        perror("[C] Can't connect to a Server"); // 커넥트 예외처리
        exit(1);
    }

    printf("[C] Connected!\n");
```

클라이언트 코드에서, 소켓을 열고 서버에 커넥트를 하는데, 동일한 컴퓨터상의 실행이라 가정하여 IP주소에 INADDR_ANY를 사용하였습니다. ( 라인 26 ) INADDR_ANY는 동작하는 컴퓨터의 IP주소를 자동으로 할당받게 해줍니다. 그 외 포트 번호도 서버코드에서와 동일하게 사용합니다.

```
38      //1. recv msg from server (maybe it's "hello")
39      check = recv(c_sock, buf, BUFFSIZE, 0);
40      if( (check == -1) || (check == 0)) {
41          perror("[C] Can't receive message");
42          exit(1);
43      }
44
45      printf("[C] Server says: %s\n", buf);
46
47      //2. say hi to server
48      check = send(c_sock, hello, sizeof(hello)+1, 0);
49      if ( (check == -1) || (check == 0)) {
50          perror("[C] Can't send message");
51          exit(1);
52      }
53
54      printf("[C] I said Hi to Server!!\n");
55
56
57      //printf("[C] I am going to sleep...\n");
58      //sleep(10);
59
60      close(c_sock);
61
62      return 0;
63
64  }
```

클라이언트가 서버와 커넥트가 된다면
서버로부터 메시지를 받고, ( 라인 38 )
서버가 전해준 메시지를 출력합니다.( 라인 45 )

그 이후 클라이언트도 서버에게 메시지를 전달하고
소켓을 닫고 종료를 하게 됩니다.

**실행 결과:**

클라이언트 코드를 n2 로 컴파일을 하고, 예외적으로 비정상 실행의 경우도 실험하기 위해 클라이언트의 코드 뒷부분을 다음과 같이 수정하여 n2_s로 컴파일하였습니다.

```
47      //2. say hi to server
48      /*check = send(c_sock, hello, sizeof(hello)+1, 0);
49      if ( (check == -1) || (check == 0)) {
50          perror("[C] Can't send message");
51          exit(1);
52      }
53
54      printf("[C] I said Hi to Server!!\n");
55      */
56
57      printf("[C] I am going to sleep...\n");
58      sleep(10);
59
60      close(c_sock);
61
62      return 0;
63
64  }
```

n2_s로 컴파일 되는 코드는 오른쪽 코드 외의 다른 부분은 모두 기존 클라이언트 코드와 동일합니다.

n2_s 코드는 서버로부터 메시지를 받고, 메시지를 전하지 않은 채 10초간 멈추었다가 소켓을 닫고 종료합니다.

실행을 하면 처음 다음과 같이 출력이 되었다가,
( 왼쪽 이미지는 서버, 오른쪽 이미지는 클라이언트 실행입니다.)

```
ubuntu@201619460:~/hw12$ ./p-server
[S] waiting for a client..#00
[S] I open the [S-1874484] at #00
[S] waiting for a client..#01
[S] I open the [S-1874485] at #01
[S] waiting for a client..#02
[S-1874484] Connected: client IP addr=127.0.0.1 port=56424
[S-1874485] Connected: client IP addr=127.0.0.1 port=56426
[S-1874484] I said Hello to Client!
[S-1874485] I said Hello to Client!
[S-1874484] Client says: Hi~ I am Client!!

[S] I open the [S-1874486] at #02
[S-1874486] Connected: client IP addr=127.0.0.1 port=56430
[S-1874486] I said Hello to Client!
[S] My Child [S-1874484] is finised with 0
[S-1874486] Client says: Hi~ I am Client!!

[S] My Child [S-1874486] is finised with 0
```

```
ubuntu@201619460:~/hw12$ ./n2_s & ./n2 & ./n2
[1] 1874481
[2] 1874482
[C] Connecting...
[C] Connecting...
[C] Connected!
[C] Server says: Hello~ I am Server!

[C] I am going to sleep...
[C] Connected!
[C] Server says: Hello~ I am Server!

[C] Connecting...
[C] Connected!
[C] Server says: Hello~ I am Server!

[C] I said Hi to Server!!
[C] I said Hi to Server!!
ubuntu@201619460:~/hw12$
```

일정 시간이 지난 후 추가적으로 다음과 같이 출력 후 종료가 됩니다.

```
ubuntu@201619460:~/hw12$ ./p-server
[S] waiting for a client..#00
[S] I open the [S-1874484] at #00
[S] waiting for a client..#01
[S] I open the [S-1874485] at #01
[S] waiting for a client..#02
[S-1874484] Connected: client IP addr=127.0.0.1 port=56424
[S-1874485] Connected: client IP addr=127.0.0.1 port=56426
[S-1874484] I said Hello to Client!
[S-1874485] I said Hello to Client!
[S-1874484] Client says: Hi~ I am Client!!

[S] I open the [S-1874486] at #02
[S-1874486] Connected: client IP addr=127.0.0.1 port=56430
[S-1874486] I said Hello to Client!
[S] My Child [S-1874484] is finised with 0
[S-1874486] Client says: Hi~ I am Client!!

[S] My Child [S-1874486] is finised with 0
[S-1874485] Can't receive message
[S] My Child [S-1874485] is finised with 256
ubuntu@201619460:~/hw12$
```

```
ubuntu@201619460:~/hw12$ ./n2_s & ./n2 & ./n2
[1] 1874481
[2] 1874482
[C] Connecting...
[C] Connecting...
[C] Connected!
[C] Server says: Hello~ I am Server!

[C] I am going to sleep...
[C] Connected!
[C] Server says: Hello~ I am Server!

[C] Connecting...
[C] Connected!
[C] Server says: Hello~ I am Server!

[C] I said Hi to Server!!
[C] I said Hi to Server!!
ubuntu@201619460:~/hw12$
[1]-  Done                    ./n2_s
[2]+  Done                    ./n2
ubuntu@201619460:~/hw12$
```

위 서버의 출력 결과에서 동일한 프로세스의 출력끼리 모아보겠습니다.

```
1) [S] waiting for a client..#00
2) [S] I open the [S-1874484] at #00
3) [S] waiting for a client..#01
4) [S] I open the [S-1874485] at #01
5) [S] waiting for a client..#02
6) [S] I open the [S-1874486] at #02
7) [S] My Child [S-1874484] is finised with 0
8) [S] My Child [S-1874486] is finised with 0
9) [S] My Child [S-1874485] is finised with 256
```

출력의 순서는 바꾸지 않고,
그냥 같은 프로세스 출력끼리 모아놓았습니다.
[S]는 서버의 부모 프로세스를 의미합니다.

1)번 출력에서 서버의 부모 프로세스는 첫 번째 클라이언트를 기달리고,
2)번 출력에서 클라이언트가 accept 되었을 때 pid가 1874484인 자식 프로세스를 만들어서 동작을 맡깁니다.
3)번 출력에서 두 번째 클라이언트를 기달리고, 4)번 출력에서 pid가 1874485인 자식 프로세스를 만들고 동작을 맡깁니다. 이후 5)번 6)번 출력도 같은 동작을 하였습니다.
7)번 출력에서 18774484 자식프로세스가 정상 종료하여 0이 출력이 되었고,
8)번 출력 또한 자식 프로세스가 정상 종료가 되었습니다.
결과 출력이 될 때, 8)번 까지만 출력이 되고 9)번은 출력되기까지 시간이 걸렸는데, 이는 9)번의 자식 프로세스(1874485)가 클라이언트의 n2_s 코드를 맡았기 때문입니다. n2_s코드는 10초간 대기를 하고 메시지를 보내지 않은 채 접속을 종료합니다. n2_s가 대기 하는 동안 자식프로세스는 종료를 하지 못하고 대기 하고 있다가, 대기 이후에도 메시지를 전달받지 못하고 종료되었기에, 비정상 종료인 256의 코드가 9)번에 출력이 된 것입니다.

```
[S-1874484] Connected: client IP addr=127.0.0.1 port=56424
[S-1874484] I said Hello to Client!
[S-1874484] Client says: Hi~ I am Client!!
```

같은 자식 프로세스들 끼리 출력 값을 모아놓은 것입니다.

```
[S-1874485] Connected: client IP addr=127.0.0.1 port=56426
[S-1874485] I said Hello to Client!
[S-1874485] Can't receive message
```

전 페이지에서 [S] 출력에서 보았던 것과 같이 9)번 출력 즉 1874485 자식 프로세스가 클라이언트로부터 메시지를 받지 못하고 종료한 것을 알 수 있습니다.

```
[S-1874486] Connected: client IP addr=127.0.0.1 port=56430
[S-1874486] I said Hello to Client!
[S-1874486] Client says: Hi~ I am Client!!
```

**추가 :**

아래의 왼쪽 사진은 강의 영상 속 교수님의 서버 결과 출력입니다.

```
ubuntu@41983:~/hw12$ ./server
[S] waiting for a client..#00
[S] Connected: client IP addr=10.0.0.249 port=42254
[S] waiting for a client..#01
[SS] Service: I am your child! pid=3052751
[SS] I said Hello to Client!
[S] Connected: client IP addr=10.0.0.249 port=42256
[S] waiting for a client..#02
[SS] Service: I am your child! pid=3052752
[SS] I said Hello to Client!
[SS] Client says: Hi~I am Client!!

[S] Connected: client IP addr=10.0.0.249 port=42258
[S] Child #00 is finished with 0
[SS] Service: I am your child! pid=3052753
[SS] I said Hello to Client!
[SS] Client says: Hi~I am Client!!

[S] Child #01 is finished with 0
[SS] Client says:
[S] Child #02 is finished with 0
ubuntu@41983:~/hw12$
```

왼쪽에 있는 강의의 출력 결과 값과 제가 짠 코드의 결과 값이 조금 다른 걸 알 수 있습니다.

(제 출력 값→)

```
[S] My Child [S-1874486] is finised with 0
[S-1874485] Can't receive message
[S] My Child [S-1874485] is finised with 256
ubuntu@201619460:~/hw12$
```

그런데, 제 서버 코드의 마지막 recv 함수,
즉 자식프로세스가 마지막 메시지를 받는 부분의 예외처리를
LMS를 보고 수정하기 이전으로 바꾸게 된다면
이 둘의 결과가 매우 유사해집니다.

아래의 빨간 박스가 제 코드를 수정 한 후 출력 한 값입니다.

```
72          //2. recv msg from client          ↗ 수정
73          if (recv(c_sock,buf,BUFFSIZE, 0) == -1) {
74              printf("[S-%d] Can't receive message\n",getpid());
75              exit(1);
76          }
77
78          printf("[S-%d] Client says: %s\n",getpid(), buf);
79
80          close(c_sock);
```

코드 수정시 출력 값

```
[S] My Child [S-1934031] is finised with 0
[S-1934029] Client says:
[S] My Child [S-1934029] is finised with 0
ubuntu@201619460:~/hw12$
```

아마 서버가 클라이언트 n2_s에 대해
데이터 크기가 0인 데이터를 수신했다고 판단했는지,
아니면 데이터 수신에 실패하였다고 판단했는지를 코드에 따라 다르게 인식하여 생긴 차이점 같습니다.

**개인과제 12-2)**

**코드:**

최대 10개까지 쓰레드를 사용하여 전송 요청을 동시에 처리하는 서버코드 와 인자를 받아 특정 파일을 요구하는 클라이언트 코드입니다. 서버는 10번 파일 전송요청을 받으며, 클라이언트는 원본파일이름에 C를 앞에 붙여 저장합니다.
일단 상대적으로 간단한 클라이언트 코드부터 보겠습니다.

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

#define BUFFSIZE 4096

int main(int argc, char* argv[]) {

    if(argc != 4) {
        printf("< Usage: %s IP_to_access PortNumber file >\n", argv[0]);
        return 1;
    }
```

클라이언트코드는 과제 11과 같게 인자로
IP주소, 포트번호, 파일명 순서대로
받게 했고
이 형식 이외의 형식으로 실행 할 시
실행이 되지 않도록 예외처리를 해 주었습니다
(라인 13 )

그 이후의 소켓을 열고, ( 라인 27 )
컨넥트를 합니다. ( 라인 36 )

과제 11과 같이 서버로부터 받아서 저장할
파일의 이름은 원본 파일 이름 앞에
C를 붙인 이름이며
그 이름을 저장할 배열 saveFname(save file name) 과 send, recv 함수 반환 값 처리를 위한 check 변수가 있습니다.
( 라인 22, 라인 24 )

```c
    int c_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t c_addr_size;
    char buf[BUFFSIZE] = {0};
    char saveFname[10];
    FILE *wfp;
    ssize_t check;
    int Fsize;

    c_sock = socket(AF_INET, SOCK_STREAM, 0); //소켓열기

    bzero(&server_addr, sizeof(server_addr)); // server_addr 초기화

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(atoi(argv[2])); // 입력받은 인자로 포트번호 입력 (check)
    server_addr.sin_addr.s_addr = inet_addr(argv[1]); // 입력받은 인자로 ip 입력 (check)


    if (connect(c_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
        perror("[C] Can't connect to a Server"); // 커넥트 예외처리
        exit(1);
    }
```

```c
    //전송 받을 파일이름 sever에 전달
    check = send(c_sock ,argv[3], sizeof(argv[3]), 0);
    if ((check == -1) || (check == 0)) {
        perror("[C] Can't send file name\n");
        exit(1);
    }

    saveFname[0] = 'C';
    strncpy(saveFname+1,argv[3],9);
    saveFname[9] = '\0';
```

클라이언트에서 컨넥트가 되었다면, 전송 받을 파일 이름을 서버에 전달합니다. ( 라인 43 )
파일 이름은 마지막 명령행 인자를 의미하므로 마지막 인자를 서버에 보냅니다.

그리고 다운 받을 파일 이름을 미리 설정해 둡니다.
( 라인 49 ~ 51 )

saveFname 배열에 첫 문자를 C로 설정하고, ( 라인 49 )
마지막 명령행 인자. 즉 파일 이름을 9개 만큼 saveFname 배열의 2번째부터 채워 넣습니다. ( 라인 50 )
그리고 혹시나 파일 이름이 saveFname 배열보다 더 많은 문자열로 이루어져 있을 때를 대비한 예외처리를 해줍니다.
( 라인 51 : strncpy 함수는 널 문자를 마지막에 넣지 않는 경우가 있으므로 )

```
53      //응답받기
54      check = recv(c_sock,buf,BUFFSIZE, 0);
55      if ((check == -1) || (check == 0)) {
56          perror("[C] Can't send file name\n");
57          exit(1);
58      }
59
60      if(buf[0] == 'Y' && buf[1] == '\0')
61      {
62          wfp = fopen(saveFname,"wb");
63          if (wfp == NULL){
64              perror("[C] Can't File open\n");
65              exit(1);
66          }
67
68          while(1)
69          {
70              memset(buf,0,BUFFSIZE);
71
72              if ((Fsize = recv(c_sock, buf, BUFFSIZE, 0)) == -1) {
73                  perror("[C] Can't receive file data.\n");
74                  exit(1);
75              }
76              if (Fsize == 0)
77              {
78                  break;
79              }
80              fwrite(buf,1,Fsize,wfp);
81
82          }
83
84          printf("[C] 파일 수신을 완료하였습니다.\n");
85          fclose(wfp);
86
87      } else if(buf[0] == 'N' && buf[1] == '\0')
88      {
89          printf("[C] 파일이 없으므로 수신하지 않습니다.\n");
90      }
91
92      close(c_sock);
93
94      return 0;
95  }
```

클라이언트가 서버에 전송 받을 파일명을 보내고 난 뒤,
서버로부터 그 파일이 있는지 응답을 받습니다.
( 라인 54 )

만약 서버에 파일이 있다면 ( 라인 60 )

파일을 저장할 이름(saveFname)으로 열고,
( 라인 62 )
서버로부터 파일을 전송 받습니다.
( 라인 68 ~ 85 )

파일 전송에는 recv함수의 리턴 값은 수신한 데이터 크기임을 이용합니다.

while문을 돌며 서버로부터 버프 사이즈만큼 지속적으로 파일을 받으며, 파일을 버프 사이즈만큼 받았을 때마다 리턴 값을 비교하여 파일 전송이 완료되었는지 파악합니다.
( 라인 68 ~ 82 )

만약 서버에 파일이 없다는 응답을 받았다면
파일을 수신하지 않고 클라이언트를 종료하게 됩니다.
( 라인 87 )

그럼 이제 서버코드를 보겠습니다.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <sys/socket.h>
6   #include <netinet/in.h>
7   #include <arpa/inet.h>
8   #include <pthread.h>
9   #include <time.h>
10
11  #define BUFFSIZE 4096
12  #define SERVERPORT 7799
13  #define MAX_THREADS (10)
14
15  pthread_t tid[MAX_THREADS];
16
17  struct twovalue{
18      int inum;
19      int sock;
20  };
21
```

왼쪽의 코드는 include 항목과 define, 그리고 전역 변수가 나타난 코드입니다.

쓰레드의 구분을 할 수있게 해주는 쓰레드ID에 쓰고자 tid 배열을 선언하고,
( 라인 15 )

구조체 twovalue를 정의하고 있습니다. 이 구조체는 int 2개를 멤버로 갖고 있는데, 이는 쓰레드에 인자로 두 개의 값을 넘겨주고자 하여 정의하였습니다.
( 라인 17 )

일단 서버 코드의 메인함수부터 보겠습니다.

```
138  int main(void) {
139      int *status, tcounts = 0; // 스레드 반환값, 스레드 생성갯수
140      struct twovalue args[MAX_THREADS]; // 인자 전달 위해
141      int i, s_sock, c_sock;
142      struct sockaddr_in server_addr, client_addr;
143      socklen_t c_addr_size;
144      char buf[BUFFSIZE] = {0};
145
146      s_sock = socket(AF_INET, SOCK_STREAM, 0); // 소켓 열기
147
148      int option = 1;     //SO_REUSEADDR의 옵션 값을 TRUE로 : 프로그램 종료후 다시실행때 bind()에서 오류 무시하기위해 사용
149      setsockopt(s_sock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));
150
151      bzero(&server_addr,sizeof(server_addr)); //server_addr 초기화
152
153      server_addr.sin_family = AF_INET;
154      server_addr.sin_port = htons(SERVERPORT);
155      server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // 동작하는 컴퓨터 IP주소 자동 할당
156
157      if (bind(s_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
158          perror("[S] Can't bind a socket"); // 바인드 안된 경우 예외처리
159          exit(1);
160      }
```

먼저, 쓰레드에 인자 전달을 하기 위해 쓰레드의 최대 생성 개수만큼 twovalue 구조체를 배열로 선언하였습니다.
( 라인 140 )
그 이후 서버에서 소켓을 열고, ( 라인 146 ) 바인드를 하였습니다.( 라인 157 )
IP주소는 과제 12-1과 동일히 INADDR_ARNY를 사용하여 동작하는 컴퓨터의 IP주소를 자동으로 할당 받습니다.
( 라인 155 )
바인드 까지는 거의 자식 프로세스로 관리하는 서버와 크게 다르지 않았습니다.

바인드 이후 listen을 해주는데, 여태 코드와는 다르게 뒤의 인자를 10으로 설정하여 동시 접속을 허용할 최대 클라이언트 수를 10개로 설정합니다. ( 라인 164 )

그리고 for 문을 돌며 10번 accept를 합니다.
( 라인 168 )

accept가 될 때마다

```
162      printf("#00\t#01\t#02\t#03\t#04\t#05\t#06\t#07\t#08\t#09\n"); // 10개의 쓰레드 명시해주기 위해
163
164      listen(s_sock,10); //동시 접속을 허용할 최대 클라이언트 수 10개
165      c_addr_size = sizeof(struct sockaddr);
166
167      for(i=0; i<10; i++, tcounts++) {
168          c_sock = accept(s_sock, (struct sockaddr *) &client_addr, &c_addr_size);
169          if (c_sock == -1) {
170              perror("[S] Can't accept a connection"); //accept 예외처리
171              exit(1);
172          }
173
174          args[i].inum = i;
175          args[i].sock = c_sock; //연결된 클라이언트 소켓 전달
176          if ( pthread_create(&tid[i], NULL, sending, &args[i]) != 0 ) {
177              perror("Failed to create thread");
178              goto exit;
179          }
180
181      }
```

쓰레드를 만들고, accept된 클라이언트와의 파일 전송을 생성한 쓰레드에게 맡기도록 합니다.
이때 쓰레드에 인자로 i의 값과 연결된 클라이언트 소켓을 args 배열 속 twovalue 구조체를 통해 전달합니다.
( 라인 174, 175 )
쓰레드로 수행할 코드의 함수는 sending으로 main 함수 전에 선언 되어있습니다.

그러면 이제 sending함수를 보겠습니다.

sending 함수입니다.  메인 함수에서 생성된 쓰레드는 이 함수부터 수행을 시작합니다.

```
22  void* sending (void *arg) {
23      int i, *ret;
24      struct twovalue indent = *((struct twovalue *)arg);
25      char buf[BUFFSIZE]; // 클라이언트와 전송할때 쓸 버퍼
26      char division[80];
27      ssize_t check; //QnA 내용처럼 send와 recv 반환값 검사용 변수
28      FILE *rfp;
29
30
31      ret = (int *)malloc(sizeof(int)); //스레드 종료때 반환값위해
32
33      srand((unsigned int)indent.inum);
34      sleep(rand()%5);
35
36      for(i=0; i<indent.inum; i++) division[i]='\t';
37
38      time_t t = time(NULL);
39      struct tm tm = *localtime(&t);
40      printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec); //시작시간 출력
41
42      for(i=0; i<3; i++) {
43          printf("%s%d...\n", division, i);
44          sleep(1);
45      }
```

쓰레드의 상태 종료 값으로 쓰이기 위한 int형 포인터 ret이 선언 되어있고, ( 라인 23 ) 미리 동적 할당을 해두었습니다. ( 라인 31 )

이 sending 함수의 쓰레드 상태 종료 값은
파일 전송 성공 시 1,
파일이 없어서 전송을 안할 시 0,
예외 종료 시 -1을 전달하도록 구현 하였습니다.

그리고 인수로 받은 구조체를 indent에 저장하였습니다. ( 라인 24 )
즉 구조체 indent의 멤버 변수 inum 에는
메인함수에서의 인자 i가 저장되었고,
멤버 변수 sock에는 연결된 클라이언트 소켓이 저장 되었습니다.

파일 전송과는 별개로 여러 전송 요청이 동시에 처리됨을 확인 할 수 있도록 랜덤적으로 최대 4초간 대기시간을 가진 후 함수를 실행하도록 설계하였습니다. ( 라인 33 ~ 34 )

그리고 강의 내용에서와 동일한 방법으로 편하게 동시 처리를 확인할 수 있도록 출력 양식을 바꾸었습니다.
( 라인 36, 그 이후 printf 함수 형식 모두 )

그리고 좀 더 확실하게 쓰레드가 동시에 동작 한다는 걸 결과 값에서 확인하고자 쓰레드가 동작할 때와 끝날 때 그 때의 현재 시각 분:초 를 출력하도록 구현하였습니다. ( 연두색 박스, 라인 38 ~ 40 )
쓰레드의 모든 예외 처리에도 쓰레드를 종료하기 전, 동일하게 분:초 출력을 하도록 구현 되있으므로, 이후 코드에선 가독성을 위해 시간 출력 코드는 모두 연두색 박스로 표시를 하겠습니다.

그리고 마지막으로 쓰레드를 시작한 후 3초간 강의 속 코드와 동일한 기법으로 카운트를 3초간 함 으로써 쓰레드의 시작 시간과 종료 시간에 차이가 나도록 구현하였습니다. 이는 동시 작동을 확인하기 위함입니다.

이후, 이제 출력적 설정은 끝나고 파일 전송 관련 코드가 나옵니다.

클라이언트로부터 파일의 이름을 받습니다.
( 라인 49 )

오른쪽 코드에서, recv 함수의 예외 처리를 하는 부분이 있는데,
예외 시 exit 가아니라 pthread_exit 함수를 호출하고 있습니다.
주황색 선으로 표시된 부분은 소켓을 닫고( 라인 51 ), ret 속 값을 갱신하고,( 라인 52 ) 그 갱신한 ret값으로 쓰레드 상태 종료 값을 주며 쓰레드를 종료하는 코드입니다.( 라인 56 )  이 또한 연두색 박스처럼 반복해서 나오는 코드이므로, 가독성을 위해 이후 코드에서 주황색 선으로 표시하겠습니다.

```
47      // file name 받기
48      check = recv(indent.sock,buf,BUFFSIZE, 0);
49      if ((check == -1) || (check == 0)) {
50          printf("%sCan't receive message\n", division);
51          close(indent.sock);
52          *ret = -1;
53          t = time(NULL);
54          tm = *localtime(&t);
55          printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec);
56          pthread_exit(ret);
57      }
58
```

```
59    rfp = fopen(buf,"rb");
60    if (rfp == NULL) { //파일이 없음
61        printf("%sNo File\n", division);
62        check = send(indent.sock,"N",sizeof("N"), 0); //파일 없다고 메세지보냄
63        if ( (check == -1) || (check == 0) ) {
64            printf("%sCan't send message\n", division);
65        }
66        close(indent.sock);
67        *ret = 0;
68        t = time(NULL);
69        tm = *localtime(&t);
70        printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec);
71        pthread_exit(ret);
72    }
73
74    check = send(indent.sock,"Y",sizeof("Y"), 0); //파일이 있음, 있다고 메세지보냄
75    if ((check == -1) || (check == 0)) {
76        printf("%sCan't send message\n", division);
77        close(indent.sock);
78        *ret = -1;
79        t = time(NULL);
80        tm = *localtime(&t);
81        printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec);
82        pthread_exit(ret);
83    }
85    //파일 전송
86    while(1){
87        memset(buf,0,BUFFSIZE);
88        indent.inum = fread(buf, 1, BUFFSIZE, rfp);
89
90        if(indent.inum < BUFFSIZE){
91
92            if (send(indent.sock, buf, indent.inum, 0) == -1) {
93                printf("%sCan't send\n", division);
94                close(indent.sock);
95                *ret = -1;
96                t = time(NULL);
97                tm = *localtime(&t);
98                printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec);
99                pthread_exit(ret);
100           }
101           break;
102       }
103       if (send(indent.sock, buf, BUFFSIZE, 0) == -1) {
104               printf("%sCan't send\n", division);
105               close(indent.sock);
106               *ret = -1;
107               t = time(NULL);
108               tm = *localtime(&t);
109               printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec);
110               pthread_exit(ret);
111           }
112   }
113
114   fclose(rfp);
115   close(indent.sock);
116
117   printf("%sSent\n", division);
118   *ret = 1;
119   t = time(NULL);
120   tm = *localtime(&t);
121   printf("%s%d:%d\n",division,tm.tm_min, tm.tm_sec);
122   pthread_exit(ret);
123 }
```

클라이언트한테 파일의 이름을 받은 후
그 이름으로 파일을 열어봄으로써 서버가 파일을 갖고 있는지 확인을 합니다. ( 라인 59 )

서버가 파일이 있다면 열기에 성공할 것입니다.

그 후 갖고 있는지 아닌지 정보를 클라이언트에게 보내고, 만약 파일을 갖고 있지 않다면 쓰레드를 종료 합니다. ( 라인 62 라인 74 )

만약, 파일이 있다면 전송을 시작합니다.

열린 rfp 파일에서 fread를 통해 항목을 1바이트씩 Buffsize만큼 읽어 buff에 데이터를 저장하였고, ( 라인 88 )
그 읽은 데이터를 클라이언트에게 전송하는 것을 while문을 통해 반복하였습니다.
( 라인 103 )

이때, fread는 읽어온 항목 수를 리턴 하므로, 리턴 값을 indent.inum에 저장하여 파일로 부터 얼마만큼의 바이트를 읽었는지 파악하였습니다. ( 라인 88 )
여기에서 리턴 값 저장 변수를 size_t 타입으로 하지 않은 이유는 버프사이즈가 4096이기에 굳이 새로운 변수를 선언하지 않고 이미 있는 int 자료형을 사용해도 전혀 문제가 되지 않았기 때문입니다.

파일을 전송하다가,
만약 읽은 바이트가 Buffsize보다 작다면, 이는 파일을 끝까지 다 읽었다는 뜻이므로 읽은 데이터를 마저 전송을 마친 뒤 while문을 빠져 나와 파일 전송을 종료하도록 하였습니다.
( 90~102 라인)

파일 전송이 모두 완료되고 쓰레드를 종료하였습니다. ( 라인 112 이후 )

이제 다시 쓰레드 생성한 메인함수로 돌아와보겠습니다.

```
183     }
184
185 exit:
186     // 반환값 출력
187     for(i=0; i<tcounts; i++) {
188         pthread_join(tid[i], (void **) &status);
189         printf("Thread no.%d ends: %d\n", i, *status);
190     }
191     close(s_sock);
192
193     return 0;
194 }
```

쓰레드를 총 10번 생성 한 후
쓰레드의 상태 종료 값을 출력하였습니다.

이미 언급했지만, 각 쓰레드의 상태 종료 값은
파일 전송 성공 시 1,
파일이 없어서 전송을 안할 시 0,
예외 종료 시 -1입니다.

**실행 결과:**

서버를 열고 10개의 클라이언트를 동시에 실행하여 10개의 파일 전송을 요구했습니다.
파일명은 a~j로 요구 하였고, a, b, c 파일만 서버에 만들어 두었습니다.
서버는 갖고 있지 않은 파일은 전송을 하지 않습니다. 즉 a, b, c 3개의 파일만 전송을 합니다.
그리고 클라이언트를 실행할 때, 10개의 클라이언트 모두에게 인자로 일일이 제 IP를(10.0.0.134)
타이핑하기가 불편하여 INADDR_ANY를 이용하였습니다.
INADDR_ANY는 #define를 통해 0으로 설정되어 있고, 동작하는 컴퓨터의 IP주소를 자동으로 할당받게 해줍니다.
그러므로 명령행 인자의 IP값에 0을 넣으면 동작하는 컴퓨터의 IP주소로 할당될 거라 생각했고, 작동하였습니다.

서버를 실행 시키고,

```
ubuntu@201619460:~/hw12$ ./server
#00     #01     #02     #03     #04     #05     #06     #07     #08     #09
```

a부터 j까지 파일명으로 client를 생성하였습니다.

```
ubuntu@201619460:~/hw12$ ./client 0 7799 a & ./client 0 7799 b & ./cli
ent 0 7799 c & ./client 0 7799 d & ./client 0 7799 e & ./client 0 7799
 f & ./client 0 7799 g & ./client 0 7799 h & ./client 0 7799 i & ./cli
ent 0 7799 j
[1] 2140109
[2] 2140110
[3] 2140111
[4] 2140112
[5] 2140113
[6] 2140114
[7] 2140115
[8] 2140116
[9] 2140118
[C] 파일이 없으므로 수신하지 않습니다.
[C] 파일이 없으므로 수신하지 않습니다.
[C] 파일 수신을 완료하였습니다.
[C] 파일이 없으므로 수신하지 않습니다.
[C] 파일이 없으므로 수신하지 않습니다.
[C] 파일이 없으므로 수신하지 않습니다.
[C] 파일 수신을 완료하였습니다.
[2]   Done                    ./client 0 7799 b
[3]   Done                    ./client 0 7799 c
[4]   Done                    ./client 0 7799 d
[5]   Done                    ./client 0 7799 e
[6]   Done                    ./client 0 7799 f
[9]+  Done                    ./client 0 7799 i
ubuntu@201619460:~/hw12$ [C] 파일이 없으므로 수신하지 않습니다.
[C] 파일 수신을 완료하였습니다.
[C] 파일이 없으므로 수신하지 않습니다.
ubuntu@201619460:~/hw12
ubuntu@201619460:~/hw12$
[1]   Done                    ./client 0 7799 a
[7]-  Done                    ./client 0 7799 g
[8]+  Done                    ./client 0 7799 h
ubuntu@201619460:~/hw12$
ubuntu@201619460:~/hw12$
ubuntu@201619460:~/hw12$
ubuntu@201619460:~/hw12$
ubuntu@201619460:~/hw12$
```

서버 실행 결과입니다.

```
ubuntu@201619460:~/hw12$ ./server
#00     #01     #02     #03     #04     #05     #06     #07     #08     #09
                39:8
                0...
                                        39:8
                                        0...
                                                                        39:8
                                                                        0...
                1...
                        39:9
                        0...
                                                                39:9
                                                                0...
                                                39:9
                                                0...
                                        39:9
                                        0...
                                                1...
                                                                        1...
                2...
                        1...
                                                        39:10
                                                        0...
                                                1...
                                        1...
                                                                1...
                                                2...
                                                                        2...
39:11
0...
        39:11
        0...
                No File
                39:11
                        2...
                                                2...
                                        2...
                                                                2...
                                                        1...
                                        No File
                                        39:11
                                                                        Sent
                                                                        39:11
1...
        1...
                        No File
                        39:12
                                                No File
                                                39:12
                                        No File
                                        39:12
                                                                Sent
                                                                39:12
                                                        2...
2...
        2...
                                                No File
                                                39:13

Sent
39:14
Thread no.0 ends: 1
        No File
        39:14
Thread no.1 ends: 0
Thread no.2 ends: 0
Thread no.3 ends: 0
Thread no.4 ends: 0
Thread no.5 ends: 0
Thread no.6 ends: 0
Thread no.7 ends: 0
Thread no.8 ends: 1
Thread no.9 ends: 1
ubuntu@201619460:~/hw12$
ubuntu@201619460:~/hw12$
ubuntu@201619460:~/hw12$
```

서버는 실행할 때 시간을 출력하고 3초간 카운트를 한 뒤 파일 전송을 하고 종료하기 전 다시 한번 시간을 출력합니다.

이 실행 결과를 각 쓰레드가 실행하고 있던 시간대별로 정리하면 다음과 같습니다.

00 - 39:11 ~ 39:14
01 - 39:11 ~ 39:14
02 - 39:8 ~ 39:11
03 - 39:9 ~ 39:12
04 - 39:9 ~ 39:12
05 - 39:8 ~ 39:11
06 - 39:9 ~ 39:12
07 - 39:10 ~ 39:13
08 - 39:9 ~ 39:12
09 - 39:8 ~ 39:11

실행 내용과 마지막 쓰레드의 상태 종료 값을 보면, 0번 8번 9번 쓰레드가 파일을 전송했음을 알 수 있습니다.

위에서 정리한 시간을 표로 나타내면 다음과 같습니다.

| 쓰레드\시간(39:) | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|
| 00 | | | | | | | |
| 01 | | | | | | | |
| 02 | | | | | | | |
| 03 | | | | | | | |
| 04 | | | | | | | |
| 05 | | | | | | | |
| 06 | | | | | | | |
| 07 | | | | | | | |
| 08 | | | | | | | |
| 09 | | | | | | | |

각 쓰레드가 실행을 시작한 시간과 종료된 시간을 보면,
쓰레드들이 동시에 동작 하고 있음을 알 수 있습니다.

**쓰레드와 프로세스의 성능차이?:**

프로세스와 쓰레드의 성능차이를 증명할 수 있는 방법이 무엇이 있을까 고민을 해보았습니다.
그러자 생각난 것이, 속도였습니다. 동일한 일을 더 빠르게 해낸다면 더 좋은 성능을 가진 것이니까!

이미 만들어 놓은 서버에서 클라이언트로 파일을 전송하는 프로그램을 축소시켜서 새로운 프로그램을 만들었습니다.
각각 fork기반, 쓰레드 기반으로 특정 파일을 다운 받는데, 인자 없이 동일한 파일을 총 4번의 파일전송으로 다른 이름으로 전송해 주는 프로그램이였습니다. 파일 전송 알고리즘은 두 코드 과제와 동일하게 사용했습니다.

accept로 클라이언트의 접속을 대기하고 있을 때, 동시에 4개의 클라이언트를 접속시켜서 clock()을 통해 끝날 때 까지 소모되는 프로그램의 시간을 측정하였습니다.
(아래 코드는 쓰레드 기반의 코드 일부 입니다. fork기반의 코드 및 전체 코드는 너무 가독성이 좋지 않은 것 같아 기재하지 않았습니다. 두 코드 모두 clock()을 쓰기위해 for문을 사용하지 않고 코드를 반복해서 작성하였습니다. for 문을 쓰면 accept 할 때마다 시간이 초기화가 되었고, accept말고 다른 타이밍에 공정히 clock함수를 실행할 수 있는 방법이 떠오르지 않았기 때문입니다. )

```
88      listen(s_sock,5); //동시 접속을 허용할 최대 클라이언트 수 3개
89      c_addr_size = sizeof(struct sockaddr);
90
91      printf("[S] waiting for a client..#\n");
92          c_sock = accept(s_sock, (struct sockaddr *) &client_addr, &c_addr_size);
93          if (c_sock == -1) {
94              perror("[S] Can't accept a connection"); //accept 예외처리
95              exit(1);
96          }
97
98          clock_t start = clock();
99
00
01          args[0]= c_sock; //연결된 클라이언트 소켓 전달
02          if ( pthread_create(&tid[0], NULL, sending, &args[0]) != 0 ) {
03              perror("Failed to create thread");
04              goto exit;
05          }
06
07          c_sock = accept(s_sock, (struct sockaddr *) &client_addr, &c_addr_size);
08          if (c_sock == -1) {
09              perror("[S] Can't accept a connection"); //accept 예외처리
10              exit(1);
11          }
```

```
145
146
147     exit:
148         // 반환값 출력
149         for(i=0; i<4; i++) {
150             pthread_join(tid[i], (void **) &status);
151         }
152         close(s_sock);
153
154         clock_t end = clock();
155         printf("Time: %lf\n",(double)(end - start));
156
157         return 0;
158     }
```

사실 엉청 대단한 듯이 전 페이지에서 이야기를 하였지만, 코드의 실행 결과는 제 생각대로 나와 주지 않았습니다...

```
ubuntu@201619460:~/hw12$ ./threadtest
[S] waiting for a client..#
Sent
Sent
Sent
Sent
Time: 981.000000
ubuntu@201619460:~/hw12$ ./forktest
[S] waiting for a client..#
[S] My Child [S-2372327] is finised with 0
[S] My Child [S-2372331] is finised with 0
[S] My Child [S-2372332] is finised with 0
Time: 452.000000
```

오히려 fork보다 쓰레드가 시간이 더 오래 걸리는 결과가 나왔습니다. 왼쪽 이미지가 실행의 결과이고 마지막 Time 이후의 숫자가 clock을 통해 측정한 프로세스의 총 걸린 시간입니다.
파일 이름에서 알 수 있듯이 위 쪽 결과가 쓰레드 기반 코드이고 아래쪽 결과가 fork 기반 코드입니다.

무엇이 문제인지 많이 고민하고 코드도 이것 저것 바꾸어 보고 다시 짜보기도 하였으나, 제 재량이 부족하여 결국 제가 원하는 해답은 얻지 못했습니다.

혹시 완벽히 같은 함수가 아니니까, 뭔가 비교연산이라든지 오차가 있을 수 있지 않을까? 하여 파일 전송이 아닌 매우 기본적인 함수들도 짜보았으나, 결과는 같았습니다.
비슷한 것도 아니고 간단한 함수에서조차 오히려 쓰레드가 더 많이 시간이 걸린다고 결과가 나왔습니다.
아래의 실행 결과가 간단한 파일의 실행결과입니다.

오른쪽 실행 결과 사진에서
fortee는 fork 기반,
thtee는 쓰레드 기반으로 만들어진
코드의 결과출력입니다.

```
ubuntu@201619460:~/hw12$ ./fortee
0123456789Parent: I am waiting my child
Parent: I am waiting my child
012345678901234567890123456789Parent: I am waiting my child
Parent: I am waiting my child
0123456789Parent: I am waiting my child
Time: 909.000000
ubuntu@201619460:~/hw12$ ./thtee
0123456789 I am waiting my thread
0123456789 I am waiting my thread
 I am waiting my thread
01234567890123456789 I am waiting my thread
0123456789 I am waiting my thread
Time: 1751.000000
```

단순히 프로세스나 쓰레드로 분기하여 for문을 돌며 0부터 9까지 출력하는 다중 프로그램으로 실험했었습니다.
for문을 돌며 개행 문자를 넣지 않았고, 프로세스나 쓰레드간 차이를 표시하지 않아, 출력결과가 조금 보기 불편할 수도 있습니다.
clock함수로 계산된 시간은 마찬가지로 “Time : ” 이후 의 숫자입니다.

동작 자체는 거의 동일하다고 생각하는 간단한 다중 프로그램인데, 이상하게도 clock 함수로 측정하였을 시 fork가 아닌 쓰레드가 종료까지 시간이 더 걸린 결과가 나왔습니다. 이해하기 어려운 결과였습니다.

아무래도 이 실험을 하며 저는
조금 다른 방법으로 접근을 해야겠다고 깨달았습니다.

속도가 아니라면 메모리 쪽으로 성능을 확인해야겠다고 생각을 했고,
이번 12-1 번 과제와 동일하게 접속은 하였으나, Sleep 등을 통해 응답을 하지 않는 프로그램을 제작하였습니다.
fork 기반, 쓰레드 기반 둘 다 최대한 동작을 비슷하게 하도록 작성하였습니다.

코드의 파일 전송 알고리즘은 과제와 동일하고
클라이언트와 서버가 Sleep을 통해 파일 전송이 늦게 동작하도록 만들었습니다.

```c
void* sending (void *arg) {
    int Fsize;
    int indent = *((int *)arg);
    char buf[BUFFSIZE]; // 클라이언트와 전송할때 쓸 버퍼
    FILE *rfp;

    printf("I am thread of #%02d : pid(%d)\n",indent,getpid());

    sleep(100);

    rfp = fopen("TEST","rb");

    while(1){ //파일 전송
        memset(buf,0,BUFFSIZE);
        Fsize = fread(buf, 1, BUFFSIZE, rfp);
```

```c
printf("[C] Connected!\n");

sleep(100);

// say file name to server
check = send(c_sock, argv[1], sizeof(argv[1]), 0);
if ( (check == -1) || (check == 0)) {
    perror("[C] Can't send message");
    exit(1);
}
```

( 위 코드에서 파일이름 전송 또는 파일 전송 전에 Sleep(100)을 통해 실행이 지속되게 하였습니다. )

저는 프로세스 기반 서버와 쓰레드 기반 서버가 각자 파일을 전송하는 프로그램을 실행하는 동안 사용하는 메모리양을 측정하고자 하였습니다.

처음으로 생각한 방법은 명령어 top을 사용하는 것이였습니다.

```
top - 16:34:44 up 80 days,  1:08,  1 user,  load average: 0.07, 0.04, 0.00
Tasks: 104 total,   1 running, 103 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us,  1.5 sy,  0.0 ni, 97.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   1987.5 total,    762.6 free,    297.4 used,    927.4 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   1497.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    606 root      20   0    8696   2968   2672 S   0.3   0.1 474:08.21 couch.sh
    973 ubuntu    20   0  990784  79288  17084 S   0.3   3.9  18:44.84 code-server
 182963 ubuntu    20   0 1065048  62328  27732 S   0.3   3.1   0:03.78 code-server
 599068 root      20   0 1006408  23244   8892 S   0.3   1.1   2:01.24 snapd
```

실시간으로 top의 메모리 사용 영역이 쓰레드 및 프로세스 기반 프로그램 실행 전과 후로 어떻게 변화하는지
동영상으로 비교 할 생각이였습니다.

그러나 막상 실험을 해보니 생각보다 눈에 띄게 메모리 사용영역이 증가하지 않았습니다. 아마 전체 영역에 비해 각 프로그램이 차지하는 양이 너무 적어서 그런 것 같았습니다.

그래서 저는 두 번째 방법으로 free를 생각했습니다.

free 명령어를 각 프로그램 실행 전, 진행 중, 종료 후에 호출하여
메모리의 사용량을 비교해 보았습니다.

저는 실행 전후의 메모리 사용량 차이를 통해 각 프로그램이 사용하는 메모리양을 비교할 수 있다 생각했습니다.

먼저 프로세스 기반 fork 프로그램입니다.

프로그램 호출 전 free입니다.

```
ubuntu@201619460:~/hw12$ free
               total        used        free      shared  buff/cache   available
Mem:         2035208      305416      781144         972      948648     1532692
Swap:              0           0           0
```

프로그램을 실행하고,

```
ubuntu@201619460:~/hw12$ ./testser
[S] waiting for a client..#00
[S] I open the [S-222626] at #00
[S] waiting for a client..#01
[S-222626] Connected: client IP addr=127.0.0.1 port=59304
[S] I open the [S-222627] at #01
[S] waiting for a client..#02
[S-222627] Connected: client IP addr=127.0.0.1 port=59306
[S] I open the [S-222628] at #02
[S] waiting for a client..#03
[S-222628] Connected: client IP addr=127.0.0.1 port=59308
[S] I open the [S-222629] at #03
[S] waiting for a client..#04
[S-222629] Connected: client IP addr=127.0.0.1 port=59310
[S] I open the [S-222630] at #04
[S-222630] Connected: client IP addr=127.0.0.1 port=59312
```

```
ubuntu@201619460:~/hw12$ ./testcli a & ./testcli b & ./testcli c & ./testcli d & ./testcli e
[1] 222621
[2] 222622
[3] 222623
[4] 222624
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
```

free를 다시 실행하였습니다.

```
ubuntu@201619460:~/hw12$ free
               total        used        free      shared  buff/cache   available
Mem:         2035208      324324      762188         972      948696     1513784
Swap:              0           0           0
```

그리고 마지막으로 프로그램이 종료 후 free입니다.

```
ubuntu@201619460:~/hw12$ free
               total        used        free      shared  buff/cache   available
Mem:         2035208      304240      782192         972      948776     1533868
Swap:              0           0           0
```

동일하게 쓰레드 기반 프로그램도 실험하였습니다.

실행 전 :

```
ubuntu@201619460:~$ free
               total        used        free      shared  buff/cache   available
Mem:         2035208      305708      779892         972      949608     1532400
Swap:              0           0           0
```

실행 중 :

```
ubuntu@201619460:~/hw12$ ./threadtest
[S] waiting for a client..#
I am thread of #04
I am thread of #05
I am thread of #06
I am thread of #07
```

```
ubuntu@201619460:~/hw12$ ./testcli a & ./testcli b & ./
testcli c & ./testcli d & ./testcli e
[13] 2528399
[14] 2528400
[15] 2528401
[16] 2528402
[C] Connecting...
[C] Connecting...
[C] Connected!
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
```

```
ubuntu@201619460:~$ free
               total        used        free      shared  buff/cache   available
Mem:         2035208      307008      778592         972      949608     1531100
Swap:              0           0           0
```

실행 후 :

```
ubuntu@201619460:~$ free
               total        used        free      shared  buff/cache   available
Mem:         2035208      305936      779656         972      949616     1532172
Swap:              0           0           0
```

위 의 free의 used 영역을 정리하면 다음과 같습니다.

| | 실행 전 | 실행 중 | 실행 후 | 실행 때 변화한 메모리 평균 |
|---|---|---|---|---|
| 프로세스 기반 | 305416 | 324324 | 304240 | (18908+20084)/2 = 19496 |
| 쓰레드 기반 | 305708 | 307008 | 305936 | (1300+1072)/2 = 1186 |

일부로 명령어 top에서의 경우를 생각하여 최대한 값을 비교하기 쉽게 하고자
free명령어에 -h같은 단위 옵션을 주지 않았습니다.
평균값은 “((실행 중-실행 전)+(실행 중-실행 후))/2” 로 계산하였습니다.

실험 결과는 프로세스 기반 프로그램의 경우가 쓰레드 기반 프로그램의 경우에 비해 메모리 사용량이 대략 18310정도 더 많았습니다.

저는 프로그램을 실행할 때와 안할 때의 메모리 사용량 차이는 그 프로그램이 실행하며 사용하는 메모리의 양이라고도 할 수 있다고 생각하여
이 실험을 통해 프로세스 기반 프로그램이 쓰레드 기반 프로그램에 비해 더 많은 메모리를 사용한다. 라는 결론을 냈습니다.

그러나 생각을 해보면 이 실험은 너무나 변수가 많았습니다.
실험을 하며 free를 계속 할 때 마다 약간씩 값의 변동이 있었고,
위 표의 메모리 변화가 온전히 실험 프로그램 때문에만 변화한 것이라고 단정 지을 수 없기  때문에
이 결과가 완전히 정확하다고는 할 수 없었습니다.

물론 그런 요소를 감안 하더라도 확실히 눈에 띄게 차이가 컸지만, 단정 지을 수는 없었습니다.

뭔가 두 번째 방법은 납득은 갈 수 있으나, 확실하게 증명이 되지 않았습니다.
그래서 조금이라도 더 명확히 해보고자 한 가지 실험을 더 하였습니다.

위에서 쓰인 각각의 프로그램을 파일 전송에 사용 될 때 호출되는 프로세스 및 쓰레드도 자신의 pid를 호출하고,
서버 자기 자신도 자기의 pid를 출력하도록 코드를 바꾸었습니다.

다음 실행 사진은 프로세스 기반 fork를 사용하는 서버와 클라이언트의 출력결과입니다.

```
ubuntu@201619460:~/hw12$ ./testser
[S] Hi! Im Server My pid is 323581
[S] waiting for a client..#00
[S] I open the [S-323707] at #00
[S] waiting for a client..#01
[S-323707] Connected: client IP addr=127.0.0.1 port=59484
[S] I open the [S-323708] at #01
[S] waiting for a client..#02
[S-323708] Connected: client IP addr=127.0.0.1 port=59486
[S] I open the [S-323709] at #02
[S] waiting for a client..#03
[S-323709] Connected: client IP addr=127.0.0.1 port=59488
[S] I open the [S-323710] at #03
[S] waiting for a client..#04
[S-323710] Connected: client IP addr=127.0.0.1 port=59490
[S] I open the [S-323711] at #04
[S-323711] Connected: client IP addr=127.0.0.1 port=59492
```

```
ubuntu@201619460:~/hw12$ ./testcl
i a & ./testcli b & ./testcli c &
 ./testcli d & ./testcli e
[1] 323702
[2] 323703
[3] 323704
[4] 323705
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connecting...
[C] Connecting...
[C] Connected!
[C] Connected!
[C] Connected!
[C] Connecting...
[C] Connected!
```

결과를 보면 서버와 클라이언트 둘 다 접속이 완료되었다는 표시가 뜨고
각 서버와 프로세스의 pid가 출력이 되었습니다.

이때, ps -aux를 호출하여 출력된 pid의 프로세스들을 찾아보았습니다.

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ubuntu    323581  0.0  0.0   2488   520 pts/0    S+   18:24   0:00 ./testser
ubuntu    323707  0.0  0.0   2488    92 pts/0    S+   18:24   0:00 ./testser
ubuntu    323708  0.0  0.0   2488    92 pts/0    S+   18:24   0:00 ./testser
ubuntu    323709  0.0  0.0   2488    92 pts/0    S+   18:24   0:00 ./testser
ubuntu    323710  0.0  0.0   2488    92 pts/0    S+   18:24   0:00 ./testser
ubuntu    323711  0.0  0.0   2488    92 pts/0    S+   18:24   0:00 ./testser
```

ps -aux에서 RSS 영역은 실제 메모리 사용량 (kb단위) 이므로
프로세스 기반 파일 전송 프로그램 서버는 총 980kb의 메모리를 차지하고 있다는 걸 알 수 있습니다.

그럼 이제 쓰레드를 이용해 구현한 파일 전송프로그램을 실행해 보겠습니다.
쓰레드 프로그램 또한 모든 서버 및 쓰레드가 자기 pid를 출력합니다.

```
ubuntu@201619460:~/hw12$ ./threadtest
[S] waiting for a client..#
[S]] I'm Server and My pid is 316942
I am thread of #04 : pid(316942)
I am thread of #05 : pid(316942)
I am thread of #06 : pid(316942)
I am thread of #07 : pid(316942)
```

```
ubuntu@201619460:~/hw12$ ./testcli a & ./testcl
i b & ./testcli c & ./testcli d & ./testcli e
[1] 317275
[2] 317276
[3] 317277
[4] 317278
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connecting...
[C] Connected!
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
```

왼쪽이 서버, 오른쪽이 클라이언트의 출력결과입니다.

클라이언트와 서버가 접속된 것이 출력이 되었고, 각 서버와 쓰레드는 자신의 pid를 출력하였습니다.
그러나 호출된 pid 모두가 동일하고, ps에서도 딱히 같은 이름의 다른 프로세스가 보이지 않습니다.

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      302740  0.0  0.0      0     0 ?        I    18:06   0:00 [kworker/u4:0-ev
ubuntu    316942  0.0  0.0  35424   568 pts/0    Sl+  18:18   0:00 ./threadtest
ubuntu    317275  0.0  0.0   2488   588 pts/1    S    18:18   0:00 ./testcli a
```

쓰레드 기반 파일 전송 프로그램 서버는 총 568kb의 메모리를 차지하고 있다는 걸 알 수 있습니다.

즉, 이번엔 명확하게 “프로세스 기반 프로그램이 쓰레드 기반 프로그램에 비해 더 많은 메모리를 사용한다.”
라고 결론을 내릴 수 있겠습니다.

위 과정들을 통해,
우리는 프로그램 실행 시간 동안 쓰레드 호출 방식이 프로세스 호출 방식에 비해 메모리 효율이 훨씬 효율적임을 알 수 있습니다.