

# TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-03-28 (26 회차)

강사: Innova Lee(이상훈)

gcccompil3r@gmail.com

학생: 정유경

ucong@naver.com

## [리눅스 시스템 프로그래밍 – 마지막 시간]

### 1. sigaction() System Call - sigaction.c

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

struct sigaction act_new; // 전역변수로 선언
struct sigaction act_old;

void sigint_handler(int signo)
{
    printf("Ctrl+C\n");
    printf("If u push it one more time then EXIT\n");
    sigaction(SIGINT, &act_old, NULL);
// act_old 에 NULL 이 저장되어 있으므로, SIGINT 들어오면 아무것도 안하고 종료하게 된다.
}

int main(void)
{
    act_new.sa_handler = sigint_handler; // act_new 의 핸들러 등록
    sigemptyset(&act_new.sa_mask);
// 특정 시그널을 막는 방법, 여기서는 아무것도 막지 않겠다는 뜻
// 중요한 작업, 우선적으로 처리해야하는 작업을 할 때에는 시그널을 막아놓는다

    sigaction(SIGINT, &act_new, &act_old);
// cf. signal(SIGINT, (void *)sig_handler);
// act_old : 이전에 동작시켰던 시그널 핸들러 저장(없으면 NULL)
    while(1)
    {
        printf("sigaction test\n");
        sleep(1);
    }
    return 0;
}
```

```

yukyong@yukyong-Z20NH-AS51B1U:~/Workspace/0329$ ./a.out
sigaction test
sigaction test
^C
If u push it one more time then EXIT
sigaction test
sigaction test
^C
yukyong@yukyong-Z20NH-AS51B1U:~/Workspace/0329$ vi sigaction.c

```

- \*. 다른사람이 작성한 코드 쉽게 이해하는 방법 : '&'에 주목  
주소 즉, 포인터를 전달한 것들은 함수 안에서 값이 바뀔 것임을 알 수 있다.
- \*. 함수는 반환값 하나, 포인터를 쓰면 여러개를 반환할 수 있다.

\*. **struct sigaction** act\_new;

```

struct sigaction
{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}

```

**sa\_handler**: signal 번호를 가지는 시그널이 발생했을 때 실행된 함수를 설치

**sa\_mask** : sa\_handler 에 등록된 시그널 핸들러 함수가 실행되는 동안 블럭되어야 하는 시그널의 마스크를 제공

\*. **sigemptyset**(&act\_new.sa\_mask);

```

#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

```

sigemptyset 함수는 인자로 주어진 시그널 셋인 set 에 포함되어 있는 모든 시그널을 비운다.

sigfillset 는 set 에 포함된 모든 시그널을 채운다.

\*.&act\_new.**sa\_mask**

sa\_mask 는 sa\_handler 에 등록된 시그널 핸들러 함수가 실행되는 동안 블럭되어야 하는 시그널의 마스크를 제공한다.

## 2. kill() System Call - kill.c 로 test.c 종료시키기

실행방법: gcc -o test test.c → ./test & (&: 백그라운드에서 동작 시킨다)

ps -ef | grep test 로 찾는다.

gcc -o kill kill.c → kill [pid]

### Kill.c

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
int main(int argc, char* argv[]) // 메인에서 인자를 받는다
{
    if(argc < 2) // 인자 2 개보다 적은경우 사용법 출력
        printf("Usage:./exe pid\n");
    else
        kill(atoi(argv[1]), SIGINT); // 입력한 문자열을 숫자로 바꾼다
    return 0;
}
```

### Test.c

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

void gogogo(int voidv)
{
    printf("SIGINT Accur!\n");
    exit(0); // 프로세스 종료
}

int main(void)
{
    signal(SIGINT,gogogo); // SIGINT 들어오면 gogogo 핸들러 실행

    for(;;)
    {
        printf("kill TEST\n");
        sleep(2); // 2 초마다 한번씩 "Kill Test 출력"
    }
    return 0;
}
```

```
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb: ~/JamieCh/0329
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ ./test &
[1] 6860
kill TEST
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ kill TEST
kill TEST
kill TEST
kill TEST
kill TEST

[1]+  Terminated                  ./test
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ ./test
kill TEST
kill TEST
kill TEST
kill TEST
kill TEST
kill TEST
Terminated
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ █

yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329
yukyoun+ 6862 6820  0 02:27 pts/18    00:00:00 grep --color=auto test
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ kill 6860
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ ps -ef | grep test
yukyoun+ 6864 6536  0 02:28 pts/2     00:00:00 ./test
yukyoun+ 6866 6820  0 02:28 pts/18    00:00:00 grep --color=auto test
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ kill 6864
yukyoungh@jamiech-lenovo-ideapad-320s-13ikb:~/JamieCh/0329$ █
```

### 3. pthread\_create(), pthread\_join() - thread.c

\*. 컴파일시 옵션 -lpthread 줄 것

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

void *task1(void* X) // void 포인터 : 뭐든지 받고 뭐든지 리턴할 수 있다
{
    printf("Thread A Complete\n");
}

void *task2(void *X)
{
    printf("Thread B complete\n");
}

int main (void)
{
```

```
pthread_t ThreadA, ThreadB; // 스레드 만들 때: 'pthread_t' ( pthread.h 에서 확인가능)
```

```
/*스레드를 메모리에 올린것이 아니라 구동시키겠다고 등록만 해놓음*/
```

```
pthread_create(&ThreadA, NULL, task1, NULL);
```

```
pthread_create(&ThreadB, NULL, task2, NULL);
```

```
/*실제 메모리에 올라가는 시점*/
```

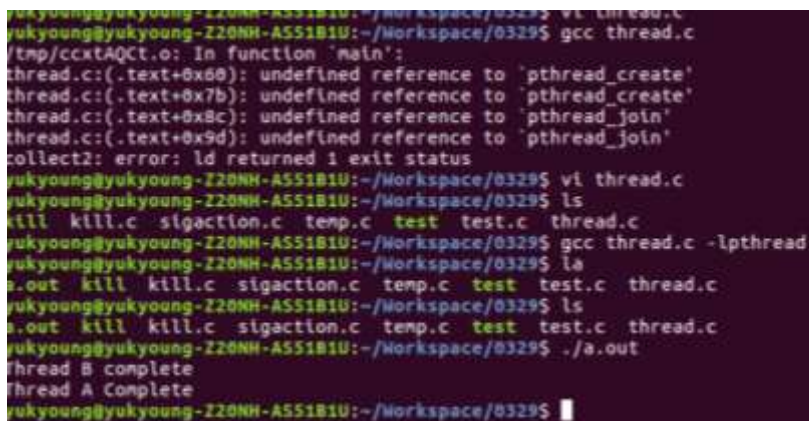
```
pthread_join(ThreadA, NULL);
```

```
pthread_join(ThreadB, NULL);
```

```
/*NULL 은 신경쓰지 말것*/
```

```
return 0;
```

```
}
```



```
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ vi thread.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ gcc thread.c
/tmp/ccxtAQct.o: In function 'main':
thread.c:(.text+0x60): undefined reference to 'pthread_create'
thread.c:(.text+0x7b): undefined reference to 'pthread_create'
thread.c:(.text+0x8c): undefined reference to 'pthread_join'
thread.c:(.text+0x9d): undefined reference to 'pthread_join'
collect2: error: ld returned 1 exit status
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ vi thread.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ls
kill kill.c sigaction.c temp.c test test.c thread.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ gcc thread.c -lpthread
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ls
a.out kill kill.c sigaction.c temp.c test test.c thread.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./a.out
Thread B complete
Thread A Complete
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$
```

\*. void 포인터 : 자료구조에서 재귀호출 해제하면서 push 할때 void 포인터로 받고 pop 할때 강제 캐스팅 했었음

\*. 병렬처리: 스레드를 이용하여 연산량을 극대화시키는 기법

\*. TCP/IP Protocol Stack???

## 4. CPU vs 그래픽카드

### \*. OpenCL??

\*. CPU 는 여러개, 그래픽카드는 만개

그래픽카드를 CPU 대신 사용하지 않는 이유는 클럭스피드가 떨어지기 때문이다.

하지만 CPU 가 여러개이므로 Bandwith 는 엄청나게 넓다.

(데이터를 한번에 입력 받을 수 있는 폭, 병렬로 처리할 수 있는 능력이 크다)

그래서 데이터를 병렬로 처리할 때는 그래픽카드가 우세하고 고속, 순차처리를 할때는 클럭 스피드가 높은 DSP(CPU)가 우세하다.

\*. 그래픽카드는 클럭스피드가 중요하지 않다. Bandwith 와 개수가 중요하다.

\*. DSP: 디지털 신호 프로세서

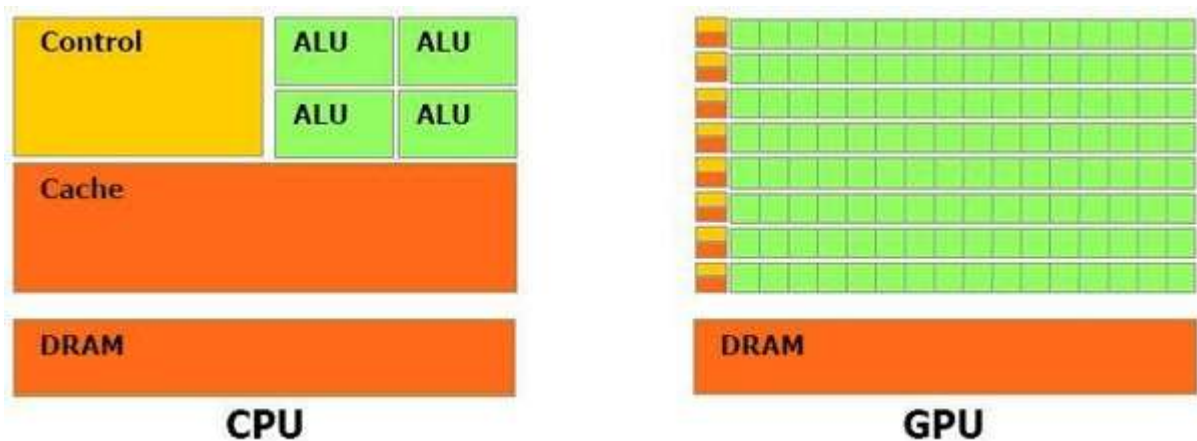
### \*. 그래픽카드 = GPU??

\*. CPU(ARM or x86 CPU)와 GPU

CPU : 적은 연속적인 작업에 유리

GPU : 많은 병렬 작업 처리에 유리

CPU 는 명령어가 입력된 순서대로 데이터를 처리하는 직렬(순차) 처리 방식에 특화된 구조를 가지고 있다. 한 번에 한 가지의 명령어만 처리한다. 따라서 연산을 담당하는 ALU 의 개수가 많을 필요가 없다.



CPU 내부 면적의 절반 이상은 캐시 메모리로 채워져 있다. 캐시 메모리는 CPU 와 램(RAM)과의 속도차이로 발생하는 병목현상을 막기 위한 장치다. CPU 가 처리할 데이터를 미리 RAM 에서 불러와 CPU 내부 캐시 메모리에 임시로 저장해 처리 속도를 높일 수 있다. CPU 가 단일 명령어를 빠르게 처리할 수 있는 비결도 이 때문이다.

반대로 GPU 는 여러 명령어를 동시에 처리하는 병렬 처리 방식을 가지고 있다. 캐시 메모리 비중이 크지 않고 연산을 할 수 있는 ALU 개수가 많다. 1 개의 코어에는 수백, 수천개의 ALU 가 장착돼 있다.

## [리눅스 네트워크 프로그래밍 – 첫 시간]

### 5. 네트워크 이론

#### 5-1 네트워크 프로그래밍에서 가장 중요한 것

[1] CS(Client, Server)

[2] 토폴로지

- 위상수학은 제외
- 네트워크의 요소들(링크, 노드 등)을 물리적으로 연결해 놓은 것, 또는 그 연결 방식(구성도)을 말한다
- 그래프 알고리즘

[3] TCP/IP 프로토콜 - **OSI 7 Layer 와 TCP/IP 4 계층**

- 이론적으로는 OSI 7layer → 실제로는 4 계층으로 만들어서 구현 및 사용 (TCP/IP 프로토콜)
  - 라우터, 스위치, 각종 OS 에 들어간다
  - 계층이 적어서 가볍다
  - (처음부터 리눅스에 포팅하여 만들어져있기 때문에) 유닉스, 리눅스에 최적화 되어있다.
- 그래서 네트워크 장비들은 리눅스나 유닉스를 사용하게 된다.

\*. 네트워킹 → 인터넷 → ip 가 필요하다

\*. ip (internet protocol) ← **중요하다!!**

- 'ifconfig' 입력시 ip address 볼 수있다.

Inet addr : 192. 168.30.9 (ipv4, inet6 addr 는 ipv6)

- Hwaddr : 1 바이트씩 6 개로 표기 → 00:00:00:00

\*. IPv4, IPv6, NAT

[1] IPv4 → 127.0.0.1

32 비트를 사용 8 비트(0~255)씩 4 자리를 끊어 쓴다.

0 번은 gateway, 255 번은 Broadcast 로 예약되어 있다.

하나의 망 안에 250 여개 정도밖에 못들어간다는 이야기.

그런데도 휴대폰 쓰고 와이파이 쓰고(ip 통신 활성화 된다는 말)

\*. 게이트웨이: 서로 다른 네트워크로 들어가는 네트워크 포인트

\*. 라우터: IP Packet 이 원하는 목적지까지 원활하게 갈 수 있도록 경로를 정해주는 역할을 하는 장비

\*. **서브넷 마스크?**

[2] IPv6 - **IPv4 와 IPv6 의 차이점**

\*. ipv6 가 나온 이유

- IPv4 프로토콜의 주소가 32 비트로 제한(2 의 32 승 개)되어 주소 공간 및 국가별로 할당된 주소가 거의 소진되고 있다는 한계점으로 인한 것

- 휴대폰, 가전제품, 스마트홈, 스마트카 등등 ip 통신하는 장치들이 늘어나면서 모두 공인 IP 를 필요로 하게된다.

자동차에 공유기를 따로 달아줄수는 없다.



즉, **센서네트워크(갑자기 센서?)**를 구축하기 위해 만들어진 것이 IPv6 이다.

### [3] NAT - **NAT 프로토콜(공인 IP, 사설 IP)**

IP 에는 2 가지 종류가 있다.

공인 IP : 있어야 외부와 통신이 가능. WAN 통신에 반드시 필요, 네이버나 다음에 접속할때 필요하다

사설 IP : 공유기

공인 IP 는 하난데 여러명이 인터넷을 사용할 수 있는 이유는 **NAT 프로토콜** 덕분이다.

### **MAC 통신과 IP 통신(스위치 장비와 라우터 장비)**

[출처] 26 회차 교육 로그|작성자 silenc3502

\*. MAC 통신이란

- MAC: HW 장치의 고유번호 → LAN 카드의 고유번호, 식별자

- 스위치에 MAC 주소가 흔적으로 남는다.

(MAC 주소를 보고 어디로 보낼지 결정하는게 스위치이기 때문)

cf. ip 를 보고 경로설정을 하는 것은 라우터

- 스위치에 MAC 을 찾고싶다고 요청하면 Broadcasting 해버린다. (DDOS 공격이 가능한 이유)

해당 MAC 주소를 가진 기기가 반응하면 **세션?!** 이 맺어지면서 통신이 이루어진다.

cf. 스위치가 본인의 MAC Table 을 검색하여 요청받은 MAC 을 찾지 못하면 (스witch는 192.168.0.X 대역을 관리한다) 관리하는 대역이 아니므로 라우터에 ip 를 보내어 경로를 물어본다. 해당 ip 를 가진 장치가 이를 인식하면 통신이 이루어진다.

→ 사실상 이 과정은 IPC 이다. 즉 네트워킹은 많이 느린 IPC 이다. (외부의 사람과 통신이 가능하다)

\*. ARP 스푸핑 : ip 충돌(중복으로 인해)나면 스위치는 MAC 을 찾을수가 없다 → 스위치는 계속

Broadcastin 한다 → 전기신호는 ADC 를 통해 디지털 신호로 바뀌어 메모리에 쌓이게 된다 → 계속 쌓이면 서버가 뻥게된다.

(이를 막기 위하여 스위치 대역폭을 넓혀서 공격자를 인식하고 접속하지 못하도록 차단한다)

## **6. socket(), bind(), listen(), accept(), connect() System Call**

- **server.c , basic\_clint.c**

```
/*Basic Server.c*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```

#include <sys/socket.h>

typedef struct sockaddr_in si;
typedef struct sockaddr * sap;

void err_handler(char* msg)
{
    fputs(msg,stderr);
    fputc('\n', stderr);
    exit(1);
}

int main(int argc, char** argv)
{
    int serv_sock;
    int clnt_sock;

    si serv_addr;
    si clnt_addr;
    socklen_t clnt_addr_size; // client 주소의 크기: 32bit(4byte)

    /* 전달하려는 문자열을 서버에 setting*/
    char msg[] = "Hello Network Programming!";

    if(argc !=2)
    {
        printf("use: %s <port>\n", argv[0]);
        exit(1);
    }
    serv_sock = socket(PF_INET, SOCK_STREAM, 0);

    if(serv_sock == -1)
        err_handler("socket() error");
    memset(&serv_addr, 0 ,sizeof(serv_addr));
    serv_addr.sin_family= AF_INET; // TCP 소켓만들때 사용하는 패턴, 알아두자
    serv_addr.sin_addr.s_addr= htonl(INADDR_ANY);
    /*INADDR_ANY : 자기자신을 주소로 받겠다.?? 127.0.0.1(로컬호스트=나)과 같은 의미*/
    serv_addr.sin_port= htons(atoi(argv[1]));

    /*스코프 바인딩: 해당 스코프를 지정하겠다
    네트워크의 바인딩: ip 주소를 셋팅한다. 바인드 하면 서버에 ip 가 셋팅이 된다. 127.0.0.1 이 셋팅*/

```

```

        if(bind(serv_sock, (sap)&serv_addr, sizeof(serv_addr)) == -1)
            err_handler("bind() error");
/*5 명 받겠다. 그 이상 받으면 안된다*/
        if(listen(serv_sock, 5) == -1)
            err_handler("listen() error");
        clnt_addr_size=sizeof(clnt_addr); //32 비트
        clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);
/*클라이언트의 접속을 accept 하려고 기다린다, accept 하면 client 소켓의 파일디스크립터가 반환된다.
실제로는 클라이언트를 기다리는 것은 listen, listen 을 거쳐서 accept 에서 클라이언트의 접속을
허용해준다 */
/*&clnt_addr : 어떤 아이피가 나에게 접속했는지 알수있다*/
        if(clnt_sock == -1)
            err_handler("accept() error");

/*원격에 있는 파일 즉, 소켓을 사용하기 위해서 IPC 통신을 한다.
네트워크는 원격으로 IPC, 동기화, 세마포어를 하는 것!! 파일 디스크립터 받아서 read, write 할 수있다*/
/*write 해서 원격에 있는 클라이언트 소켓에 쓴다 그래서 clnt 에서 메세지가 나온 것! */
// 클라이언트 소켓에 쓴다 → 버퍼 msg → fd 즉, sock 으로 전달되었다!!!
        write(clnt_sock, msg, sizeof(msg));
        close(clnt_sock);
        close(serv_sock);

        return 0;
}

```

```

/*Basic Client.c*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

typedef struct sockaddr_in si;
typedef struct sockaddr * sap;

void err_handler(char *msg)
{
    fputs(msg, stderr);
    fputc('\n', stderr);
    exit(1);
}

```

```

int main(int argc, char **argv)
{
    int sock;
    int str_len;
    si serv_addr;
    char msg[32];
    if(argc !=3) // argc 가 3 개 IP 주소를 알아야하므로! (옆사람의 아이피 주소를 넣어보자)
    { // 사실 아이피는 무조건 192.168.0.X
        printf("use: %s <IP> <port>\n", argv[0]); // ./clnt <ip> <port>
        exit(1);
    }
    /*포트번호(서비스번호): 통로로서 각 번호마다 특정 역할(서비스)을 수행한다.
    80: WWW.
    20,21: FTP (file transfer protocol – up/download)
    22: SSH
    cf. 7777 우리가 만든 임의의 포트 */

    /* <ip> <port>를 잘 입력하여 주면 소켓으로 간다.*/
    sock = socket(PF_INET, SOCK_STREAM,0); // socket 은 네트워크의 open!!
    // 유닉스에서는 소켓도 파일이므로 파일 디스크립터를 반환한다. (sock 가 파일디스크립터)
    //IPv4 에 해당하는 소켓스트림(TCP 패킷)을 사용하겠다

    if(sock == -1)
        err_handler("socket() error");

    /*serv_addr 초기화 패턴*/
    /*&serv_addr 는 si 구조체 안에는 다음 3 개의 멤버가 들어있다.*/
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET; // UDP 말고 TCP 를 쓸것이다.
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // 아이피주소
    serv_addr.sin_port = htons(atoi(argv[2])); // 어떤 서비스를 열 것인가 (포트번호 7777)

    /*serv_addr 와 connect 하겠다 → 서버의 listen 에서 받아서 accept 하여 서버와 클라이언트간 통신된다*/
    // sock: 자기 자신에 대한 네트워크 파일 디스크립터
    // (sap)??? &serv_addr 에 연결한다.
    if(connect(sock, (sap)&serv_addr, sizeof(serv_addr))!=-1)
        err_handler("connect() error");

    /*서버에서 write 했으니까 read 한다!!*/
    // read 는 블로킹함수 들어올때까지 움직이지 않음

```

// 파일디스크립터(sock)에서 읽어와서 msg 에 저장

```
str_len = read(sock, msg, sizeof(msg)-1);
```

```
if(str_len == -1)
```

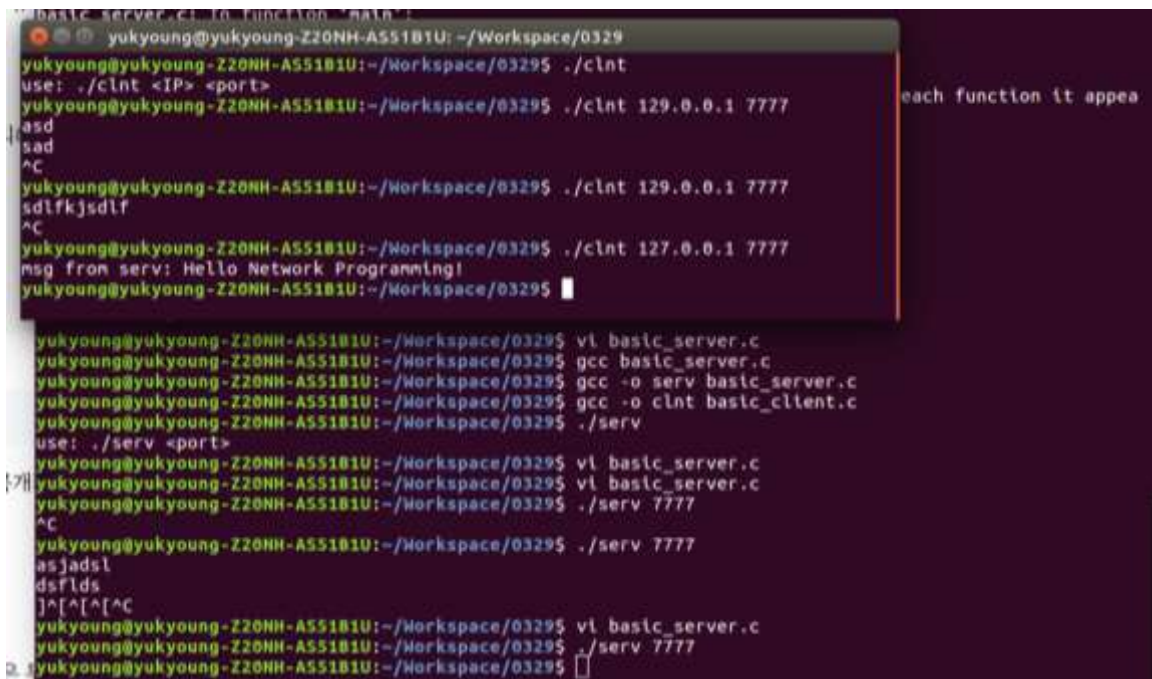
```
    err_handler("read() error!");
```

```
    printf("msg from serv: %s\n", msg);
```

```
    close(sock);
```

```
    return 0;
```

```
}
```



```
yukyoung@yukyoung-Z20NH-A551B1U: ~/Workspace/0329
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./clnt
use: ./clnt <IP> <port>
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./clnt 129.0.0.1 7777
asd
sad
^C
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./clnt 129.0.0.1 7777
sdlfkjsdlf
^C
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./clnt 127.0.0.1 7777
msg from serv: Hello Network Programing!
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$

yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ vi basic_server.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ gcc basic_server.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ gcc -o serv basic_server.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ gcc -o clnt basic_client.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./serv
use: ./serv <port>
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ vi basic_server.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ vi basic_server.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./serv 7777
^C
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./serv 7777
asjads!
dsfids
]^^[^^[^C
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ vi basic_server.c
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$ ./serv 7777
yukyoung@yukyoung-Z20NH-A551B1U:~/Workspace/0329$
```

\*. 127.0.0.1 : 자기 자신

## 7. Socketfd.c - 이 예제를 통해 소켓이 파일임을 알수 있다.

```
/*Socketfd.c*/
```

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

```
#include <fcntl.h>
```

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

```
#include <sys/socket.h>
```

```
int main()
```

```
{
```

```

    int fd[3];
    int i;
    /*IS_SOCK() 옵션에 해당하는 소켓파일 ?! */
    fd[0]= socket(PF_INET, SOCK_STREAM,0);    //Sock_stream: TCP
    fd[1]= socket(PF_INET,SOCK_DGRAM,0);    // Sock_Dgram: UDP
    fd[2]= open("test.txt", O_CREAT|O_WRONLY|O_TRUNC);
    // tast_struct > files_struct > file 의 인덱스 번호??
    // socket() 도 open()도 모두 파일이다! TCP 든 UDP 든 전부 파일로 받아들인다.소켓의 종류는 중요하지
    않다.

    for(i=0;i<3;i++)
        printf("fd[%d]=%d\n",i,fd[i]);

    for(i=0;i<3;i++)
        close(fd[i]);
    return 0;
}

```

\*. TCP 와 UDP 의 차이점: 3way handshaking

## 8. read 중간에 데이터의 손실이 없게 만들기 - read\_client.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

typedef struct sockaddr_in si;
typedef struct sockaddr* sap;

void err_handler(char*msg)
{
    fputs(msg,stderr);
    fputc('\n', stderr);
    exit(1);
}

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

```

```

int sock;
int str_len=0;
si serv_addr;
char msg[32] = {0};
int idx=0, read_len=0;

if(argc != 3)
{
    printf("use: %s <IP> <port>\n", argv[0]);
    exit(1);
}

```

```

sock = socket(PF_INET, SOCK_STREAM,0);

```

```

if(sock == -1)
    err_handler("socket() error");

```

```

memset(&serv_addr,0,sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
serv_addr.sin_port = htons(atoi(argv[2]));

```

```

if(connect(sock,(sap)&serv_addr, sizeof(serv_addr))!=-1)
    err_handler("connect() error");

```

/\* 네트워크 상에서 예기치 못한 문제 발생, 데이터가 중간에 끊기게 되면 라우터가 우회하여 전송시킨다.  
이때 그래프 알고리즘이 사용된다\*/

/\*data 크기가 전부 16 일때, 12 바이트만 들어오고 나머지 4 바이트가 남은상황에서  
read\_len 은 12 바이트 str\_len 도 12 바이트 다시 read 올라가서 남은 4 바이트를 받아온다.

이를 str\_len 에 더해준다. **중간에 끊기더라도 데이터의 손실이 없게된다!!\*/**

// 코드를 통해서 어떻게 끝까지 read 가 이루어진다는 건지 잘 모르겠다...메커니즘은 이해가 가는데

```

while(read_len = read(sock,&msg[idx++],1))
{
    if(read_len==-1)
        err_handler("read() error!");
    str_len += read_len;
}
printf("msg from serv: %s\n", msg);
printf("read count: %d\n", str_len);
close(sock);

```

```

return 0;

```

```
}
```

## 9. htons(), htonl() System Call - convert\_endian.c

변수에 저장되는 타입에 따라 리틀엔디안과 빅 엔디안으로 나뉜다 바로 그걸 보는 예제

```
#include <stdio.h>
#include <arpa/inet.h>
int main()
{
    unsigned short host_port = 0x5678; // 16 진수 (2 바이트)
    unsigned short net_port; // 56 78 을 1 바이트씩 크로스 매칭한 결과 → 78 56
    unsigned long host_addr = 0x87654321; (4 바이트)
    unsigned long net_addr; // → 21 43 65 87

    net_port = htons(host_port); // host to network // short(2 바이트) 타입으로 처리해라
    net_addr = htonl(host_addr); // long(4 바이트)타입으로 처리해아

    printf("Host Ordered Port: %x\n",host_port);
    printf("Network Ordered Port: %x\n",net_port);
    printf("Host Ordered Address: %lx\n",host_addr);
    printf("Network Ordered Address: %lx\n",net_addr);

    return 0;
}
```

\*. 네트워크에서는 리틀엔디안, 빅엔디안을 전부 하나로 통일시키기 위해 풀어놓는다.  
CPU 마다 엔디안이 다르기 때문  
(실제 메모리에 배치되는 형태로 크로스 매칭해서)

## 10. inet\_addr() System Call - inet\_addr.c

```
#include <stdio.h>
#include <arpa/inet.h>

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

    char *addr1="3,7,5,9"; // 03 07 05 09 → 09 05 07 03 (맨앞 0 은 출력 x)
```



```

char *addr2="1,3,5,7"; // 01 03 05 07 → 7 05 03 01

unsigned long conv_addr = inet_addr(addr1);
if(conv_addr == INADDR_NONE)
    printf("ERROR!\n");
else
    printf("Network Ordered integer Addr: %lx\n", conv_addr);

conv_addr = inet_addr(addr2);
if(conv_addr == INADDR_NONE)
    printf("error!\n");
else
    printf("Nework Orderd Integer Addr: %lx\n", conv_addr);

return 0;
}

```

\*,

AF\_INET 주소체계도 리틀/ 빅엔디안으로 매핑되어 있을텐데  
PF\_INET

\*. inet\_addr() 함수

-함수 파라미터 값에 IP 주소 문자열의 시작주소를 넣어주면 이 함수가 알아서 빅엔디안 32 비트 unsigned long 형의 값으로 만들어줍니다.

-성공하면 빅엔디안 형식의 32 비트 값을, 실패하면 INADDR\_NONE 을 리턴

```

yukyoun@yukyoun-Z20NH-AS51B1U:~/Workspace/0329$ ./a.out
Network Ordered integer Addr: 9050703
Nework Orderd Integer Addr: 7050301
yukyoun@yukyoun-Z20NH-AS51B1U:~/Workspace/0329$

```