Xilinx Zynq FPGA,TI DSP, MCU 기반의 프로그래밍 전문가 과정

날 짜 : 2018 . 4. 2

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 정한별
hanbulkr@gmail.com

<gserver.c>

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

#defineBUF_SIZE		128
#define MAX_CLNT	256

typedef struct sockaddr_in	si;
typedef struct sockaddr *	sp;

int clnt_cnt = 0;
int clnt_socks[MAX_CLNT];
int data[MAX_CLNT];
int thread_pid[MAX_CLNT];
int idx;
int cnt[MAX_CLNT];
//
pthread_mutex_t mtx;

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

void sig_handler(int signo)
{
	int i;

	printf("Time Over!\n");

	pthread_mutex_lock(&mtx);

	for(i = 0; i < clnt_cnt; i++)
		if(thread_pid[i] == getpid())
			cnt[i] += 1;

	pthread_mutex_unlock(&mtx);
```

```
	alarm(3);
}

void proc_msg(char *msg, int len, int k)
{
	int i;
	int cmp = atoi(msg);
	char smsg[64] = {0};

	pthread_mutex_lock(&mtx);

	cnt[k] += 1;

	if(data[k] > cmp)
		sprintf(smsg, "greater than %d\n", cmp);
	else if(data[k] < cmp)
		sprintf(smsg, "less than %d\n", cmp);
	else
	{
		strcpy(smsg, "You win!\n");
		printf("cnt = %d\n", cnt[k]);
	}
	// 스트링에 가져다가 붙이는것.
	// 사실 좋은 구조는 클라이언트한테 숫자 쏘고 클라이언트 종료되게 해야함.
	strcat(smsg, "Input Number: \n");
	write(clnt_socks[k], smsg, strlen(smsg));

#if 0
	for(i = 0; i < clnt_cnt; i++)
	{
		if(data[i] > cmp)
			sprintf(smsg, "greater than %d\n", cmp);
		else if(data[i] < cmp)
			sprintf(smsg, "less than %d\n", cmp);
		else
			strcpy(smsg, "You win!\n");

		strcat(smsg, "Input Number: ");
		write(clnt_socks[i], smsg, strlen(smsg));
	}
#endif
	// 이제 언락을 시켜준다.
	pthread_mutex_unlock(&mtx);
}

void *clnt_handler(void *arg)
{
	int clnt_sock = *((int *)arg);
	int str_len = 0, i;
```

```
	char msg[BUF_SIZE] = {0};
	char pattern[BUF_SIZE] = "Input Number: \n";

	//
	signal(SIGALRM, sig_handler);

	// 쓰레드의 pid 값이 나온다 getpid 에서. 쓰레드를 통해 들어온거니까
	pthread_mutex_lock(&mtx);
	thread_pid[idx++] = getpid();
	i = idx - 1;
	printf("i = %d\n", i);
	// 아 쓰레드마다 자기가 관리하는 클라이언트의 썪이고 크리티컬 섹션이 나오지 않게 하기 위해
	// 락을 걸어준 것이다.
	write(clnt_socks[i], pattern, strlen(pattern));
	pthread_mutex_unlock(&mtx);

	alarm(3);

	while((str_len = read(clnt_sock, msg, sizeof(msg))) != 0)
	{
		// 잘 들어왔으니 알람을 초기화 한다.
		alarm(0);
		proc_msg(msg, str_len, i);
		alarm(3);
	}

	pthread_mutex_lock(&mtx);

	for(i = 0; i < clnt_cnt; i++)
	{
		if(clnt_sock == clnt_socks[i])
		{
			while(i++ < clnt_cnt - 1)
				clnt_socks[i] = clnt_socks[i + 1];
			break;
		}
	}

	clnt_cnt--;
	pthread_mutex_unlock(&mtx);
	close(clnt_sock);

	return NULL;
}

int main(int argc, char **argv)
{
	int serv_sock, clnt_sock;
	si serv_addr, clnt_addr;
```

```
socklen_t addr_size;
pthread_t t_id;
int idx = 0;

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

srand(time(NULL));

pthread_mutex_init(&mtx, NULL);

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;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));

if(bind(serv_sock, (sp)&serv_addr, sizeof(serv_addr)) == -1)
	err_handler("bind() error");

if(listen(serv_sock, 2) == -1)
	err_handler("listen() error");

for(;;)
{
	addr_size = sizeof(clnt_addr);
	// 클라이언트의 접근을 승인한다. 다음 클라이언트가 들어오기 전까지 블록한다.
	clnt_sock = accept(serv_sock, (sp)&clnt_addr, &addr_size);

	thread_pid[idx++] = getpid();

	// 여기서 lock 을 건다. data 가 꼬이지 말라고 하는것이다.
	// 소켓파일이 서버랑 클라이언트랑 공유메모리라서.
	// 이 순간부터 clnt_sock 은 건들수 없다.
	pthread_mutex_lock(&mtx);
	data[clnt_cnt] = rand() % 3333 + 1;
	clnt_socks[clnt_cnt++] = clnt_sock;
	// 다 하고 나서  락을 풀어준다.
	pthread_mutex_unlock(&mtx);

	// 쓰레드에  전달되는 인자 (마지막꺼_fd 값)
	pthread_create(&t_id, NULL, clnt_handler, (void *)&clnt_sock);
```

```
                // 떼어내다... 쓰레드 t_id 는 식별자 혹은 id (cpu 에 새로 할당됨)
                pthread_detach(t_id);
                printf("Connected Client IP: %s\n", inet_ntoa(clnt_addr.sin_addr));
        }

        close(serv_sock);

        return 0;
}
```

<gclient.c>

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

#define BUF_SIZE                128

typedef struct sockaddr_in      si;
typedef struct sockaddr *       sp;

char msg[BUF_SIZE];

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

// pthead_creat 의 마지막인자는 여기로 들어온다. 여러개 보낼려면 구조체 형으로 보내야한다.
// 3 번째 인자에는 void * 형으로 들어와야한다... 어떤게 올지 모르니까.
// 검색 속도를 올릴려면 , 예를 들어 이진트리를 병렬 검색을 한다
void *send_msg(void *arg)
{
        int sock = *((int *)arg);
        char msg[BUF_SIZE];

        for(;;)
        {
                // 무한루프를 돌면서 계속 읽고 전송해 준다.
                fgets(msg, BUF_SIZE, stdin);
```

```
                write(sock, msg, strlen(msg));
        }

        return NULL;
}

void *recv_msg(void *arg)
{
        int sock = *((int *)arg);
        char msg[BUF_SIZE];
        int str_len;

        for(;;)
        {
                // 서버로 부터 받아서
                str_len = read(sock, msg, BUF_SIZE - 1);
                // 0 으로 해주는 이유는 먼저 긴게 들어오고 나중에 짧은게 들어올때 덮어씌면서 뒤가 남음.
                // memset 을 해주는게 좋지만 널문자를 끝맺음을 함으로 동작에 이득을 취한다.
                msg[str_len] = 0;
                // 모니터의 출력을 한다.
                fputs(msg, stdout);
        }

        return NULL;
}

int main(int argc, char **argv)
{
        int sock;
        si serv_addr;
        pthread_t snd_thread, rcv_thread;
        void *thread_ret;

        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]));

        // 커넥트 하는 순간 어셉트가 동작한다.
        // 위에서 설정한 (포트와 주소 정보(serv_adr)) 를 가지고 connect 를 시도한다.
        if(connect(sock, (sp)&serv_addr, sizeof(serv_addr)) == -1)
                err_handler("connect() error");

        // 송신과 수신을 분리하기 위해서 한다. fork 랑 비슷하다.
```

```
        // 전송.
        pthread_create(&snd_thread, NULL, send_msg, (void *)&sock);
        // 수신.
        pthread_create(&rcv_thread, NULL, recv_msg, (void *)&sock);
        // 본격적인 구동은 join 을 할때. 더이상 센드, 혹은 보낼게 없거나 하면 멈춤.
        pthread_join(snd_thread, &thread_ret);
        pthread_join(rcv_thread, &thread_ret);

        close(sock);

        return 0;
}
```

<file_server.c>

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

#define BUF_SIZE                32

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

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

        int serv_sock, clnt_sock, id,fd;
        char buf[BUF_SIZE] = {0};
        int read_cnt ;

        si serv_addr, clnt_addr;
        socklen_t clnt_addr_size;

        if(argc != 2)
        {
```

```
        printf("use:%s <port>\n", argv[0]);
        exit(1);
}

// 파일 서버를 열고 있다. 만약 뭔가를 전송 하고 싶으면 여기를 바꾸면 됩니다.
fd = open("file_server.c" ,O_RDONLY);
serv_sock = socket(PF_INET, SOCK_STREAM , 0);

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

// 네트워크를 저장하는 순서가 다르다 Byte order(little endian , big endian)
// 다른 컴퓨터와 통신을 이루려면 이것을 맞추어야 한다.
// host to network short (htons)
// host to network long (htonl)
// serv_addr.sin_addr.s_addr 은 INADDR_ANY 가 자신의 주소를 컴터가 찾아준다.(127.0.0.1)
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET; // AF_INET (IPv4 로 고정)
serv_addr.sin_port = htons(atoi(argv[1]));
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

// fd(serv_sock), 연결요구 받기전 자신의 주소를 지정해 클라이언트의 연결 요구를 이곳으로 오게한다.
if(bind(serv_sock, (sap)&serv_addr, sizeof(serv_addr)) == -1)
        err_handler("bind() error");

// 보통 연결의 최대 값은 5 이다 backlog 에 accept() 한수 실행시 누적되는 연결 수를 알 수 있다.
// listen 을 통해서 여러 클라이언트를 준비한다.
if(listen(serv_sock, 5) == -1)
        err_handler("listen() error");

clnt_addr_size = sizeof(clnt_addr);

// 연결이 성공된 clnt_addr(주소) 를 받아 저장한다. 블록킹 함수로 연결을 기다린다.
// 클라이언트와 소통을 하기 위해 자동을 소켓을 생성한다. (클라이언트쪽 소켓)
//
clnt_sock = accept(serv_sock , (sap)&clnt_addr, &clnt_addr_size);

for(;;)
{
        // 클라이언트에게 글을 받을 때까지 대기 한다. *블록상태...
        read_cnt = read(fd, buf, BUF_SIZE);

        if(read_cnt < BUF_SIZE)
        {
                write(clnt_sock ,buf, read_cnt);
                break;
        }

        write(clnt_sock, buf , BUF_SIZE);
```

```
	}

	shutdown(clnt_sock, SHUT_WR);
	read(clnt_sock ,buf, BUF_SIZE);
	printf("msg from client: %s\n", buf);

	close(fd);
	close(clnt_sock);
	close(serv_sock);
	return 0;
}
```

<file_client.c>

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

#define BUF_SIZE	32

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

int main(int argc, char **argv)
{
	char buf[BUF_SIZE] ={0};
	int fd, sock, read_cnt;
	si serv_addr;

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

```
        }

        fd = open("receive.txt", O_CREAT | O_WRONLY);
        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]));

        //  connect 신호가 날라오면 서버에서 accept 시 연결이 된다.
        if(connect(sock, (sap)&serv_addr, sizeof(serv_addr)) == -1)
                err_handler("connect() error");
        else
                puts("Connected ......");

        while((read_cnt = read(sock ,buf, BUF_SIZE)) != 0)
                write(fd, buf, read_cnt);

        puts("Received File Data");
        write(sock, "Thack you", 10);
        close(fd);
        close(sock);
        return 0;

}
```

<gethostbyname.c>

```
#include<stdio.h>
#include<unistd.h>
#include<arpa/inet.h>
#include<netdb.h>
#include<stdlib.h>


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

```
int main(int argc ,char **argv)
{
	int i;
	struct hostent *host;

	if(argc != 2)
	{
		printf("use : %s <poer>\n", argv[0]);
		exit(10);
	}

	host = gethostbyname(argv[1]);

	if(!host)
		err_handler("gethost ... error !");

	printf("Official Name : %s\n", host ->h_name);

	for(i = 0; host->h_aliases[i] ; i++)
		printf("aliases %d : %s\n" , i+1, host -> h_aliases[i]);

	printf("Address Type: %s\n", (host->h_addrtype == AF_INET) ? "AF_INET" :"AF_INETS");

	//ntohs()-"Network to Host Short"
	//ntohl()-"Network to Host Long"
	for(i= 0 ; host->h_addr_list[i]; i++)
		printf("IP Addr %d:%s \n", i+1, inet_ntoa(*(struct in_addr *)host->h_addr_list[i]));

	return 0;
}
```

<pthread 활용법>

서버와 클라이언트 프로그램관 TCP 통신을 한다. 클라이언트는 편의상 fork 를 사용하여 5 개의 프로세스를 생성하고, 각 클라이언트는 랜덤값을 서버로 전송하고, 처리 결과값을 출력한다.
서버는 각 클라이언트마다 thread 를 열고, 클라이언트에서 전송 받은 값을 전역변수인 result 에 더한값을 저장하고, 클라이언트에 처리 결과를 넘겨준다. 저장한 값을 순차적으로 처리하기 위해서 mutex 를 사용하여 처리한다.

Mutex, 뮤텍스)는 동시 프로그래밍에서 공유 불가능한 자원의 동시 사용을 피하기 위해 사용되는 알고리즘으로, 임계 구역(critical section)으로 불리는 코드 영역에 의해 구현된다.

공유 데이터를 접근하는 프로그램 내부의 이른바 임계 구역이라는 부분은 홀로 수행되도록 보호되어야 하며, 다른 스레드가 동일한 부분의 프로그램을 수행해서 동일한 공유 데이터를 접근하는 것을 막아야 한다.

< pthread 함수를 쓰기 위해서 >

1. #include <pthread.h>

2. 컴파일할때는 반드시 **-lpthread** 옵션을 주어야 합니다.

3. pthread_t : pthread 의 자료형을 의미

int pthread_create(pthread_t *th_id, const pthread_attr_t *attr, void* 함수명, void *arg);

- pthread 생성합니다

첫 번째 인자 : pthread 식별자로 thread 가 성공적으로 생성되면 thread 식별값이 주어진다.
두 번째 인자 : pthread 속성(옵션), 기본적인 thread 속성을 사용할 경우 NULL
세 번째 인자 : pthread 로 분기할 함수. 반환값이 void* 타입이고 매개변수도 void* 으로 선언된 함수만 가능하다.
네 번째 인자 : 분기할 함수로 넘겨줄 인자값. 어떤 자료형을 넘겨줄 지 모르기 때문에 void 형으로 넘겨주고 상황에 맞게 분기하는 함수내에서 원래의 자료형으로 캐스팅해서 사용하면 된다.
리턴 값 : 성공적으로 pthread 가 생성될 경우 0 반환

int pthread_join(pthread_t th_id, void thread_return);**

- 특정 pthread 가 종료될 때까지 기다리다가 특정 pthread 가 종료시 자원 해제시켜준다.
첫 번째 인자 : 어떤 pthread 를 기다릴 지 정하는 식별자
두 번째 인자 : pthread 의 return 값, 포인트로 값을 받아오는 점을 주의할 것.

void pthread_exit(void* ret_value);

- 현재 실행중인 thread 를 종료시킬 대 사용한다.
보통 pthread_exit 가 호출되면 cleanup handler 가 호출되며 보통 리소스 해제하는 일을 수행한다.

void pthread_cleanup_push(void* (함수명), void* arg);

- pthread_exit()가 호출될 때 호출된 handler 를 정하는 함수.

보통 자원 해제용이나 mutex lock 해제를 위한 용도로 사용된다.

void pthread_cleanup_pop(int exec);

- 설정된 cleanup handler 를 제거하기 위해서 사용되는 함수.

exec 값을 0 일 경우 바로 cleanup handler 제거하고

그외의 값을 가질 경우 cleanup handler 를 한번 실행한 후 제거한다.

pthread_t pthread_self(void);

- 현재 동작중인 pthread 의 식별자를 리턴한다.

쓰레드 동기화 함수

pthread_mutex_init
pthread_mutex_destroy
pthread_mutex_lock
pthread_mutex_unlock
pthread_cond_init
pthread_cond_signal
pthread_cond_boradcast
pthread_cond_wait
pthread_cond_timewait
pthread_cond_destroy

쓰레드 Attribute 함수

pthread_attr_init
pthread_attr_distroy
pthread_attr_getscope
pthread_attr_setscope
pthread_attr_getdetachstate
pthread_attr_setdetachstate

쓰레드 시그널 관련 함수

pthread_sigmask
pthread_kill
sigwait

쓰레드 취소 함수

pthread_cancel
pthread_setcancelstate
pthread_setcancelstate
pthread_setcanceltype
pthread_testcance

<네트워크 프로그래밍 기본기>

유닉스의 모든 것은 파일이라는 말이 있다. 이 말은 유닉스 프로그래밍을 해 보면 모든 I/O 작업이 파일에 쓰고 읽음으로써 이루어 진다는 말이다. 네트웍의 작동 또한 마찬가지이다. 양쪽의 컴퓨터가 각각 파일, 즉 socket 을 열어서 그 파일을 연결시켜 통신이 이루어지는 것이다.
주로 쓰이는 socket 의 타입은 2 종류이다. 하나는 stream socket 이고 다른 하나는 datagram socket 이다.

Stream socket 은 안정적인 양방향의 연결을 만들 때 필요한 소켓이다. 이 연결은 양방향으로 통신이 가능하며, 데이터들의 순서가 정확하다. 또한 이 연결은 tcp 프로토콜을 사용하므로 에러 체킹이 되어 안정적인 데이터 전송이 가능하다.

반면 datagram socket 은 단방향 일회성이라 할 수 있다. 하나의 packet(데이터 덩어리)에 주소를 집어넣어 보내면 그 패킷이 혼자서 목적지로 찾아가는 방식이다.

소켓 관련 시스템 호출

connection-oriented socket 을 만들어 연결을 하기 위해서는 다음과 같은 과정이 필요하다.

Server: socket() -> bind() ->listen() -> accept()
Client: socket() -> connect()

이 시스템 호출들은 대부분 sockaddr 구조체를 인수로 필요로 한다. sockaddr 구조체는 다음과 같다.
(서버→bind, accept 에 필요　　클라이언트→connect 에 필요)

```
#include <sys/socket.h>
struct sockaddr
{
    unsigned short sa_family;   /* addressmfamily, AF_xxx   */
    char  sa_data[14]; /*14bytes of protocol address */
};
```

Sa_family 는 프로토콜의 종류를 결정하는 것으로, 우리가 사용할 것은 AF_inet(Internetfamily)
AF_INET family 를 위해 다음과 같은 구조체가 sockaddr 구조체를 지정하기 위해 정의되어 있다.

```
#include <netinet/in.h>
struct sockaddr_in
{
     short int  sin_family;    /*Address family */
     unsigned short int sin_port; /* Port number */
     struct in_addr     sin_addr; /*internet address  */
     unsigned char      sin \_zero[8]; /*Same size as struct sockaddr*/
};
struct in_addr
{
    unsigned long s_addr;
}
```

sin_family 는 AF_INET 로 지정될 것이며, sin_port 는 포트번호, sin_addr 은 인터넷 주소를 지정한다. Sin_zero[]는 sockaddr 과 크기를 맞추기 위한 배열이기 때문에 sockaddr_in 을 사용하기 전에 bzero()나 memset()을 이용해서 0 으로 초기화한다.

< Byte Ordering Routine >

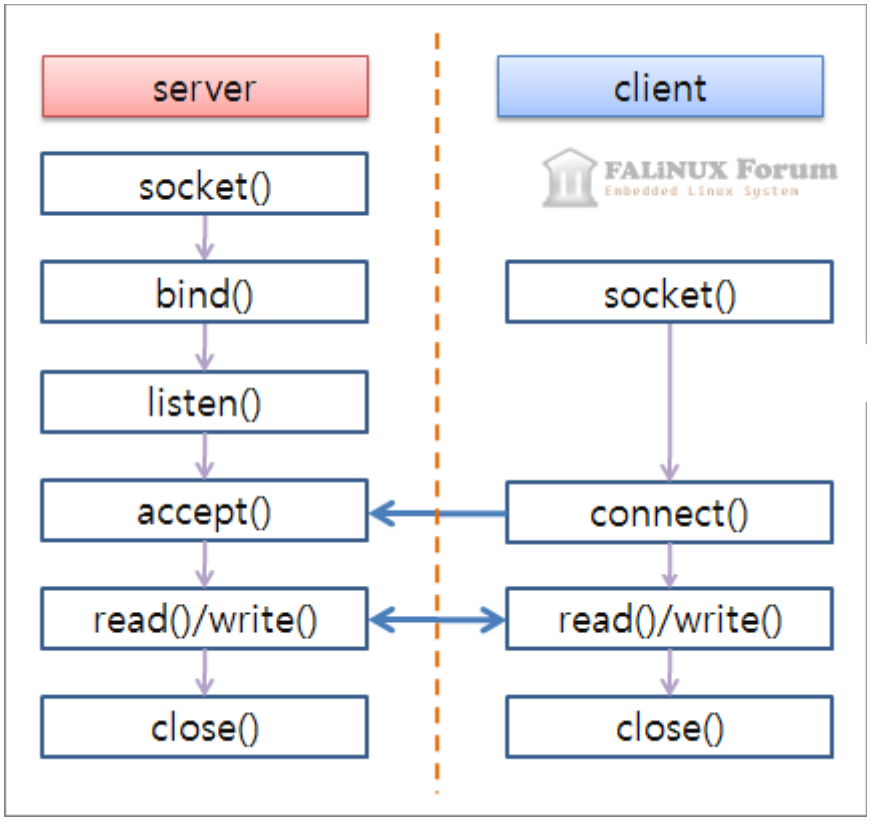
컴퓨터 마다 byte 들을 변수에 저장하는 순서가 다르다.
네트웍 상에서의 데이터의 byte order 를 Network byte order.

우리가 sin_port 와 sin_addr 에 값을 지정하려면 컴퓨터의 host byte order 인 변수 값을 network byte order 로 바꾸어서 저장해야 한다.

htons()-"Host to Network Short"
htonl()-"Host to Network Long"
ntohs()-"Network to Host Short"
ntohl()-"Network to Host Long"

serv_addr.sin_addr.s_addr 은 INADDR_ANY 가 자신의 주소를 컴터가 찾아준다.(127.0.0.1)

tcp/ip 동작 방식



Server

1) 서버 소켓을 만든다. socket()

int socket(int family, int type, int protocol);
fd =socket(PF_INET, SOCK_STREAM, 0);

family 는 소켓 가족을 지정하는 부분이다. 우리는 AF_INET 만을 사용한다.
type 은 stream socket 이라는 것을 말하는 SOCK_STREAM 으로 지정을 한다.
ptocol 은 0 을 지정한다.
이 시스템 호출은 파일 지정번호(file descriptor)를 반환한다.
우리는 이 함수 호출을 다음처럼만 쓰게 될 것이다.

2) bind()

// fd(serv_sock), 연결요구 받기전 자신의 주소를 지정해 클라이언트의 연결 요구를 이곳으로 오게한다.
if(bind(serv_sock, (sap)&serv_addr, sizeof(serv_addr)) == -1)
myaddr 은 자기 자신의 주소이다.
Addrlen 은 myaddr 의 크기를 말한다. Sizeof(myaddr)과 같이 쓰면 될 것이다. 이 시스템 호출은 소켓에 자신의 주소를 지정한다. 서버는 클라이언트로부터 연결 요구를 받기 전에 자신의 주소를 지정해 줌으로써 이 주소로 오는 신호는 모두 이 소켓으로 들어오라고 말해 주는 것이다.

3) listen()

int listen(int sockfd, int baklog);
backlog 는 accept() 한수 실행 시에 얼마나 많은 연결이 누적될 수 있는가를 말한다. 동시에 2 개의 연결 요구가 발생할 때 accept()에서 하나의 연결을 받아들이는 도중에는 다른 하나는 누적(pending)될 것이다. 이러한 누적되는 연결의 개수를 의미한다. 보통 최대값이 5 개이다.

4) accept()

int accept(int sockfd, struct sockaddr * peer,int *addrlen);
peer 는 연결이 성립되었을 때 연결된 상대방의 주소를 저장할 공간이다. 이 시스템 호출은 실제적인 연결을 기다리는 함수이다. 연결 요청이 있을 경우 accept()는 새로운 소켓을 생성하고 연결하여, 이 파일 지정번호를 반환한다. 연결 요청이 없으면 non-blocking 소켓이 아닌한 연결을 기다린다. Non-blocking 소켓에 대해서는 다음에 살펴보기로 한다.

Client

1) socket() 을 연다.

2) connect()

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
servaddr 은 연결을 할 상대방의 주소이다.
이 시스템 호출은 자신의 시스템과 서버의 실제적인 연결을 시도한다.
클라이언트 프로그램은 socket() 시스템 호출 후 이 시스템 호출로 서버에 연결을 시도 할 것이다.

Shutdown(clnt_sock , SHUT_WR)

---

how to 상수

how to 상수

SHUT_RD : recv buffer 만 차단 한다. 해당소켓으로 부터 수신 불가.
SHUT_WD : send buffer 만 차단 한다. 해당 소켓으로 송신 불가.
SHUT_RDWR : 두 버퍼 모두 차단 한다. 더이상 해당 소켓은 통신 불가.

성공한면 0, 실패하면 -1 을 반환한다. Close() 햠수와 마찬가리조 소켓을 종료하지만 how to 상수에 어떤 값을 넣느냐에 따라 read 버퍼와 write 버퍼를 차단할지 선택할 수 있다.

struct hostent *gethostbyname(const char *name);

주어진 호스트 name 에 상응하는 hostent 타입의 구조체를 반환한다. hostent 구조는 아래와 같습니다.

struct hostent

```
{
  char *h_name;                 /* Official name of host.  */
  char **h_aliases;             /* Alias list.  */
  int h_addrtype;               /* Host address type.  */
  int h_length;                 /* Length of address.  */
  char **h_addr_list;           /* List of addresses from name server.  */
#define h_addr  h_addr_list[0]  /* Address, for backward compatibility.  */
}
```

인자

const char *name 호스트 이름이거나 표준 점 표기법의 IPv4 주소, 콜론(그리고 점 표기법도 가능)표기법의 IPv6

반환 값

성공 hostent 구조체

NULL 에러, h_errno 변수에 에러 넘버 대입