socket

```
//UDP
socket(AF_INET, SOCK_DGRAM, 0);

//TCP
socket(PF_INET, SOCK_STREAM, 0);

/*소켓에 주소 할당*/
sockaddr(AF_INET, port, addr);

//포트번호 할당 & 소켓 구조체에 sockaddr 구조체 할당
bind(socket fd, const struct sockaddr *addr, len);
```

엔디안 변경

- 빅 엔디안: 상위 바이트 값이 메모리의 작은 주소 (네트워크)
- 리틀 엔디안: 하위 바이츠 값이 메모리의 작은 주소 (호스트 바이트, 주로 사용하는 것)

IP, Port 주소와 같은 리틀 엔디안을 빅 엔디안으로 변경할 필요 존재.

```
htonl //host to network long
htons //host to network short
ntohs //network to host long
ntohs //network to host short
```

- IP 변환

```
inet_addr(const char* cp); //ip4 to big
inet_network(const char* cp); //ip4 to little
inet_aton(const char *cp, struct in_addr *inp); //ipv4 to big, 주소 구조체에 저장까지 지원
char *inet_ntoa(struct in_addr in); //big to ipv4
```

- IP6 지원

```
//af: IP 버전, src: 바꿀 값, dst: big형태로 변환 후 저장

//ip->network
inet_pton(int af, const char *src, void *dst);

//network->ip
inet_ntop(int af, const char *src, char *dst, socklen_t size);
```

- ip byte 변환

```
//ip to host + host to network == ip to network
inet_network() + htonl() == inetaddr();
```

- socket에 주소 할당

```
char ip = "127.0.0.1";
int port = 9001;

struct sockaddr_in sockaddr;

//소켓 주소 구조체에 IP 버전, 네크워크 바이트 IP와 port 할당
sockaddr.sin_family = AF_INET;
sockaddr.sin_addr.s_addr = inet_addr(ip);
sockaddr.sin_port = htons(port);
```

호스트 이름과 IP 주소

host name -> IP

```
//host name(www.example.com) -> IP
struct hostent *gethostbyname(const char *name);
```

hostent 구조체에는 host name을 IP로 변경하는 여러 정보가 존재한다.

```
struct hostent *ent;
struct in_addr **res;

char *h_name = "www.example.comm"
//host name -> IP
ent = gethostbyname(h_name);
//IP주소 하나식 가져오기 (배열 형태로 저장됨)
res = (struct in_addr **)ent -> h_addr_list;
```

소켓 프로그램

- Server

```
//클라이언트 접속을 기다리는 함수
listen(int sockfd, int backlog);
listen(srvsock, 5) //최대 5명
```

```
//클라이언트의 연결 요청 받기
accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);
accept(srvsock, (struct sockaddr*)&clntaddr, &clntaddrlen);
```

- Client

```
//클라이언트가 sockfd에게 연결을 요청한다
connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
connect(srvsock, (struct sockaddr *)&clntaddr, sizeof(clntaddr));
```

- 입/출력

```
//sockfd에 buf데이터를 len만큼 송신
send(int sockfd, const void *buf, size_t len, int flags);

//sockfd에서 온 데이터를 buf에 len만큼 수신
recv(int sockfd, void *buf, size_t len, int flags);
```

소켓 송/수신 프로그램 흐름

- Server

1. socket() - 소켓 생성
2. sockaddr_in - 소켓 주소 저장//IP버전, IP, PORT
3. bind(srvsock, srvaddr, srvaddrlen) - 소켓에 주소 할당
4. listen(srvsock, 최대 연결가능) - 클라이언트의 연결 기다리기
   클라이언트의 연결 요청(connect)
5. accept(srvsocket, clntaddr, clntaddrlen) - 클라이언트 연결 허용, clntsock에 저장
6. read/write

```
int main(){
    int srvsock, clntsock;

    struct sockaddr_int srvaddr, clntaddr;
    char rbuf[1024];
    char wbuf[] = "hello from server";

    srvsock = socket(AF_INET, SOCK_STREAM, 0);

    memset(&srvaddr, 0, sizeof(srvaddr));

    srvaddr.sin_family = AF_INET;
    //어느 IP에서도 접속 가능(0.0.0.0)
    srvaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
    srvaddr.sin_port = htons(PORT);

    bind(srvsock, (struct sockaddr *)&srvaddr, sizeof(srvaddr));

    listen(srvsock, 5);

    clntaddrlen = sizeof(clntaddr);

    /*client로부터 연결 요청*/
    clntsock = accept(srvsock, (struct sockaddr *)&clntaddr, &clntaddrlen);

    int readlen = read(clntsock, rbuf, sizeof(rbuf)-1);
    rbuff[readlen] = '\0';

    write(clntsock, wbuff, sizeof(wbuff));

    close(clntsock);
    close(srvsock);

    return 0;
}
```

- Client

1. socket() - 소켓 생성
2. sockaddr_in - 소켓 주소 저장//IP버전, IP, PORT
3. bind(clntsock, clntaddr, clntaddrlen) - 소켓에 주소 할당
4. connect(clntsock, clntaddr, clntaddrlen) - 서버에 연결 요청 **서버의 accept**
5. read/write

```
int main(){
    int clntsock;

    struct sockaddr_in clntaddr;
    char wbuff[] = "hello from client";
    char rbuff[1024];

    clntsd = socket(AF_INET, SOCK_STREAM, 0);

    memset(&clntaddr, 0, sizeof(clntaddr));

    clntaddr.sin_family = AF_INET;
    //호스트 자신
    clntaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    //접속할 주소
    clntaddr.sin_port = htons(PORT);

    connect(clntsock, (struct sockaddr *)&clntaddr, sizeof(clntaddr));

    write(clntsock, wbuff, sizeof(wbuff));
    int readlen = read(clntsock, rbuff, sizeof(rbuff)-1);
```

```
    rbuff[readlen] = '\0';

    close(clntsock);
    return 0;
}
```

TCP Socket - iterative 모델

다수의 client에게 순차적인 서비스를 제공한다.

- Server

```
while(1){
    accept()
    while(1){
        read() / write()
    }
}
```

- Client

```
connect()
while(1){
    fgets(wbuff, BUFSIZ-1, stdin);
    write() / read()
}
```

while문을 통해서 구현할 수 있지만, client1 연결하는 동안 client2가 연결하면 정상적으로 작동하지 않는다.

UDP Socket

- connect() 불필요, PORT로 통신
- 다자간 통신 쉽게 구현
- 신뢰적이지 않기 때문에 응용 프로그램 수준에서 신뢰성 있게 데이터를 전송해야 한다.

```
int clntsock = socket(AF_INET, SOCK_DGRAM, 0);

sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr
*dest_addr, socklen_t addrlen);

recvfrom(int sockfd, const void *buf, size_t len, int flags, struct sockaddr
*src_addr, socklen_t *addrlen);

//sendto, recvfrom: 오버헤드
//connect()-> read/write: 오버헤드 적음
```

1. 다중 수신 가능
2. recvfrom(서버역할)이 켜져 있지 않아도 데이터 전송 가능
3. 데이터 도달 여부를 확인하지 않는다.
4. listen/connect 없음
5. UDP에서 connect를 통한 read/write 가능하지만 다중 수신 안됨.
6. bind()를 통해 포트 번호 지정 가능하지만, 호출하지 않아도 sendto()알아서 해줌

Advanced Socket

- IPv6 inet_aton() or inet_addr() -> inet_pton()
  inet_ntoa() -> inet_ntop()
  gethostbyname() -> getaddrinfo()

- getaddrinfo()
  host -> ip를 제공하는 gethostbyname()과 달리 hostname or ip 모두 지원한다.

```
getaddrinfo(DNS of IP, http or port, addrinfo *hints, addrinfo **res);
```

```
struct addrinfo hints;
struct addrinfo *res;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UPNSPEC; //미정
hints.ai_socktype = SOCK_STREAM; //TCP
hints.ai_flags = AI_PASSIVE; //IP 자동할당

//NULL: 서버프로그램에서 주소 필요없음, port
getaddrinfo(NULL, "3490", &hints, &res);


/*client에서는*/
hints.ai_family = AF_UPNSPEC; //미정
hints.ai_socktype = SOCK_STREAM; //TCP
//flag 할당 x

//sockaddr_in 과정 간략화
getaddrinfo("www.example.net", "3490", &hints, &res);

int sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

bind(sockfd, res->ai_addr, res->ai_addrlen)

---

/*기존(예전)의 방식*/
int sockfd;
struct sockaddr_in my_addr;
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);

my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = inet_addr(IP);
my_addr.sin_port = htons(MYPORT);

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr));
```

sockaddr_in -> addrinfo 사용으로 더욱 간단하고 자동화되어있는 소켓 생성 가능

Simple Time Console + Networking

```
const char *response =
    "HTTP/1.1 200 OK\r\n"
    "ConnectionL close\r\n"
    "Content-Type: text/plain\r\n\r\n"
    "Local time is: ";

int bytes_sent = send(socket_client, response, strlen(response), 0);

time_t timer;
time(&timer);
char *time_msg = ctime(&timer);

bytes_sent = send(socket_client, time_msg, strlen(time_msg), 0);
```

Multi Process

- listen(srvsd, 5)
  iterative한 성질을 갖고있어 5개까지 접속 가능하더라도, 처음 연결한 client가 끝날 때까지 기다려야한다.

tcpfirstclnt -> tcptestclnt 변경 후 다중 접속시 단일 프로세스 서버는 multiple client를 처리하지 못한다는 것을 알 수 있다.

- fork()
  호출시점에 존재하는 부모의 모든 정보를 복사한다. 프로세스 나뉜 후 바뀐 자원은 공유되지 않는다.

```
int pid;
if(pid=fork()==0){
    //child
}
else if(pid==-1){
    //error
}
else{
    //parent
}
```

- wait(), waitpid() child process가 종료할 때까지 기다리는 함수

Multi process based server

1. parent: client accept()발생시 fork()
2. parent에서 client socket close
3. child에서 server socket(listen) close

```
clntsd = accept(srvsd, (struct sockaddr *)&clntaddr, sizeof(clntaddr));

if(pid=fork()==0){
    close(srvsd);

    while(1){
        read(clntsd, rbuff, sizeof(rbuff)-1);
        write(clntst, rbuff, sizeof(rbuff));
    }
}
else{
    close(clntsd);
}
```

-> 문제점
client의 연결 요청마다 fork하기 때문에 매우 비효율적

Multi process based client

client에서 multi process 프로그래밍 -> UDP 기반 채팅 구현 가능
client에서 부모에서는 sendto처리, 자식에서는 recvfrom하는 형태로 진행됨.. 사용자간 메세지를 주고받을 수 있다.
서버에서는 연결 client를 관리해서 메세지가 들어오면 해당 송신자 빼고 전체에게 수신.

IPC

- signal
  시그널 핸들러로 발생 시그널 처리 및 좀비프로세스 처리

```
void sigchld_handler(int sig){
    if(sig==SIGCHLD){
        wait(&status);
        close(clntsd);
    }
}

int main(){
    ...
    while(1){
        ...
```

```
            if(pid=fork()==0){
                ...
                END
                kill(getppid(), SIGCHLD);
                close(clntsd);
            }
            else{
                signal(SIGCHLD, sigchld_handler);
            }
        }
        ...
    }
```

시그널 발생시 wait()함수를 호출함으로써 닫힌 자원 바로 반납 가능하다.
그냥 wait() 하면 CHild가 죽을 때까지 Parent는 아무것도 하지 않는다.

Multi Thread & Multiplexing

Process와 다르게 자원을 공유하는 형태이다.

pthread

- pthread_create()를 통해 스레드를 생성할 수 있다.
- pthread_join() 스레드 수행 종료까지 기다리는 함수이다. (= 스레드 끝날 때까지 main 종료하지 않음)
- pthread_detach(): 스레드 종료시 자동 자원

semaphore

- sem_open(): 세마포어 생성
- sem_wait(): 세마포어가 0이면 양수가 될때까지 critical section을 대기상태로 전환
- sem_post(): 작업 끝내고 다른사람도 작업할 수 있도록 알린다.
- sem_unlink(): 세마포어 제거

mutex

- pthread_mutex_init()
- pthread_mutex_lock / unlock
- pthread_mutex_destroy()
- socket programming with thread
  client accept 할 때마다 스레드 생성하여 처리하는 형식. (pthread_join이 아닌 detatch)

Multiplexing

- select()

1. 수신한 데이터를 지니고 있는 소켓이 존재하는가?
2. 데이터의 전송이 가능한 socket은 무엇인가?
3. 예외상황이 발생한 소켓은 무엇인가?

```
select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

연결된 소켓에 읽을 데이터가 있는지 없는지 확인
nfds: 연결된 fd 중 가장 높은 값

for문장을 통해 연결된 소켓 (=관심 있는 소켓)을 전부 검사

- listen 소켓에서 이슈 발생시 new client 연결 처리 & 관찰대상 등록
- 그외에는 메세지 전송 이슈

```
// 기본세팅
FD_ZERO(&defaultFds);
//listen소켓 포함
FD_SET(listenSd, &defaultFds);
maxFd = listenSd;

while(1){
    //변경할 수도 있기 때문에 원본 미리 복사해두기
    rFds = defaultFds;

    //이슈 발생 fd존재한다면..
    if((res=select(maxFd+1, &rFds, 0, 0, NULL))==-1) break;

    for(int i=0;i<maxFd+1;i++){
        //이슈 발생 fd만 확인
        if(FD_ISSET(i, &rFds)){
            //listen소켓 -> accept new client
            if(i==listenSd){
                //accept new client
                FD_SET(clntSd, &defaultFds);
                //max 넘으면 변경
            }
            //else -> 기존 client message
            else{
                read()
                for(int j=0;j<maxFd;j++){
                    //송신자 빼고 모두 수신
                    if(i==j) continue;
                    write()
                }
            }
        }
    }
}
```

select를 통해 multi process/thread 없이도 다중통신을 구현할 수 있다.
select의 관심있는 소켓 관리.. 라는 개념을 사용하는 것이다.

SHA 256호출

```
int main(){
    char input[] = "hello world";

    size_t input_len = sizeof(input)-1;

    char hash[SHA256_DIGEST_LENGTH];

    SHA256(input, input_len, hash);

    for(int i=0;i<SHA256_DIGEST_LENGTH;i++)
        printf("%02x", hash[i]);
}
```

TCP 기반 2인 채팅

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


#define PORT 9000

int main() {
    int sockfd, newsockfd1, newsockfd2;

    socklen_t clilen;

    struct sockaddr_in serv_addr, cli_addr1, cli_addr2;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    memset((char *) &serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = (PORT);

    bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
    listen(sockfd, 3);

    clilen = sizeof(cli_addr1);
    newsockfd1 = accept(sockfd, (struct sockaddr *) &cli_addr1, &clilen);

    clilen = sizeof(cli_addr2);
    newsockfd2 = accept(sockfd, (struct sockaddr *) &cli_addr2, &clilen);
```

```
    int pid = fork();

    if (pid == 0) {
        char buff[1024];

        while(1){
            read(cli_addr1, buff, sizeof(buff)-1);
            printf("client 1: %s", buff);
            write(cli_addr2, buff, sizeof(buff));

        }
    } else {
        char buff[1024];

        while(1){
            write(cli_addr1, buff, sizeof(buff));
            printf("client 2: %s", buff);
            read(cli_addr2, buff, sizeof(buff)-1);

        }
    }
    return 0;
}
```

채팅이 되긴 하나 한 번 전송해야 다른 사용자의 답변을 받을 수 있다는 단점 존재

Basic Chatting App

코드...

epoll

select()와 마찬가지호 다수의 fd를 관찰하여 요청이 온 fd를 발견하면 작업 수행

1. eopll 객체 생성
2. epoll 객체 제어
3. 모니터링 시작
4. 조건에맞는 fd의 I/O 수행

```
//epoll fd, 함수를 통해 하려는 작업, 모니터링할 fd, epoll_event 종류와 정보 전달
epoll_ctl(int eopllfd, int op, int fd, struct eopll_event *event);

//epoll 모니터링
epoll_wait()
```

selet와는 다르게 모든 fd를 확인해서 FD_ISSET 돌리는 게 아니라 이벤트 일어난 fd만 따로 저장해 ready로 관리

- epoll mode

1. edge-trigger, 이벤트 일어나면 알림
   버퍼에 데이터 남아있어도 다시 dat ㅁ 채울 때까지 read하지 못한다.
2. level-trigger, 이용 가능하면 알리기
   버퍼에 데이터 남아있으면 계속 접근 가능

- non-blocking socket
  fcntl()

```
makeNBSocket(listenSd);

int makeNBSocket(int socket){
    int res;
    res = fcntl(socket, F_GETFL, 0);
    res!=O_NONBLOCK;
    res = fcntl(socket, F_SETFL, res);
    return  0;
}
```