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

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
28 일차 (2018. 04. 01)

게임 서버부분

```
#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>

#define BUF_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;            //lock 의 키 값

void err_handler(char *msg)     //에러 메시지를 입력 받아 출력하는 함수
{       fputs(msg, stderr);
        fputc('₩n', stderr);
        exit(1);                }

void sig_handler(int signo)     // 시간 초과를 알리는 함수
{       int i;

        printf("Time Over!₩n");

        pthread_mutex_lock(&mtx);     //임계 영역 잠금. 다른 쓰레드의 접근을 막을 수 있다.
          for(i = 0; i < clnt_cnt; i++)
                if(thread_pid[i] == getpid())     //pid 값이 쓰레드 pid 값과 같을 경우 숫자세기.
                        cnt[i] += 1;
        pthread_mutex_unlock(&mtx);  //임계 영역 잠금 해제.

        alarm(3);       }       //3 초 알람
```

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

	pthread_mutex_lock(&mtx);	//임계 영역 잠금.

	cnt[k] += 1; // 몇 번 입력했는지 세주는 것

	if(data[k] > cmp) //데이터가 큰지 작은지에 대해
		sprintf(smsg, "greater than %d₩n", cmp);
	else if(data[k] < cmp)
		sprintf(smsg, "less than %d₩n", cmp);
	else	// 크거나 작지 않으면 같은 것이므로 숫자를 맞추어서 알려주는 것.
	{	strcpy(smsg, "You win!₩n");
		printf("cnt = %d₩n", cnt[k]);		}

	strcat(smsg, "Input Number: ₩n"); //스트링에 갖다 붙이는 거
	write(clnt_socks[k], smsg, strlen(smsg));//클라이언트 소켓에 정보 전달. 위 문자열 출력
#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
	pthread_mutex_unlock(&mtx);  }		//임계 영역 잠금 해제

void *clnt_handler(void *arg)	// pthread_create 의 3 번째 인자로 함수가 실행되면서 4 번째 인자가 인자
{	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); //3 초 내에 입력하라는 것
```

```
        pthread_mutex_lock(&mtx); //임계 영역 잠금. lock 으로
        thread_pid[idx++] = getpid(); //쓰레드의 pid 값을 저장하는 것
        i = idx - 1; // 현재 인덱스 값을 확인하기 위해서
        printf("i = %d₩n", i);
        write(clnt_socks[i], pattern, strlen(pattern));// 첫번째 클라이언트에 패턴을 쓰겠다는 것. 패턴 써줌
        pthread_mutex_unlock(&mtx);
 //락을 풀어줌. = 임계 영역 잠금 해제. 왜 해줘야 하는가? 클라이언트가 여러명이면 쓰레드도 여러개 생성된다. 소켓도 쓰레드 마다 허용하는 것이 있다. 락을 걸지 않으면 작업하면서 데이터가 서로 꼬일 수 있다.
        alarm(3); //3 초마다 대기

        while((str_len = read(clnt_sock, msg, sizeof(msg))) != 0) //write 로 패턴 보냈으니까 클라이언트가 쓴 것을 수신하는 것. 클라이언트의 숫자  입력을 읽음
        {       alarm(0); //수신이 잘 되었으니 알람을 끈다.
                proc_msg(msg, str_len, i); //얼마만큼 들어왔는지
                alarm(3);                       } // 끝나지 않았을 경우 다시 3 초 맞추어주어 맞을 때까지반복

        pthread_mutex_lock(&mtx);       // 임계 영역 잠금

        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;
                }
        } // 종료하는 부분이 없어서 맞추어도 계속 돌게 된다.

        clnt_cnt--;  //????
        pthread_mutex_unlock(&mtx);  //  잠금 해제
        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)     // 인자가 2 개가 아닌 경우 오류 메시지 출력
        {       printf("Usage: %s <port>₩n", argv[0]);
                exit(1);                        }
```

```
srand(time(NULL));        //  랜덤 설정

pthread_mutex_init(&mtx, NULL); // 전역변수. 위에 정의된 함수는 없는데 lock 건 것이 없어서 초기화 한 것 pthread_mutex_t mtx;에서 가져왔다.

serv_sock = socket(PF_INET, SOCK_STREAM, 0); // 서버 소켓 파일디스립터 저장

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) //bind 로 서버 소켓에 주소 부여
        err_handler("bind() error");       //반환 값이 -1 이면 에러. 실패했을 시 -1 을 반환함으로

if(listen(serv_sock, 2) == -1) // 두명 받고 있어서 두명 이상이면 안 됨
        err_handler("listen() error");

for(;;) //실제 알고리즘 들어가는 부분
{       addr_size = sizeof(clnt_addr);
        clnt_sock = accept(serv_sock, (sp)&clnt_addr, &addr_size);
```
// 클라이언트의 접근을 승인하는 것. 리턴 결과는 클라이언트의 소켓이다. 서버 프로그램이 클라이언트의 연결 요청으로 서버 소켓과 다른 소켓을 생성하여 파일 디스크립터를 반환 받기 때문이다. 다음 클라이언트가 올 때까지 블로킹.

```
        thread_pid[idx++] = getpid(); //  배열에 프로세스 pid 저장

        pthread_mutex_lock(&mtx);
```
//lock 을 거는 이유는 데이터 꺼지지 말라고 하는 것. lock 을 여기다 거는 이유? 값이 꼬이는 것을 방지하기 위해서이다. 소켓 파일이 서버랑 클라이언트랑 공유되어서.. 중간부터 락이 걸렸으니까 쓰레드가 돌고 있어도 배열에 접근을 할 수가 없기 때문이다.

```
        data[clnt_cnt] = rand() % 3333 + 1;
        clnt_socks[clnt_cnt++] = clnt_sock; //clnt_socks 배열에 clnt_sock 저장하는 것
        pthread_mutex_unlock(&mtx); // 쓰레드가 작업할 수 있게끔 락을 풀어주는 것

        pthread_create(&t_id, NULL, clnt_handler, (void *)&clnt_sock); //t_id 는 쓰레드 아이디 값생성. clnt_handler 가 무엇인가? 쓰레드가 되는 함수 자체이다. 4 번째는 쓰레드에 전달되는 인자이다.
```

pthread_detach(t_id); // man 페이지로 확인 detach a thread detach 는 떼어낸다는 뜻으로 위에서 생성하고 (지역변수로 t_id 가 선언되어 있다. man 페이지로 확인 쓰레드 id 값이다) 프로세스랑 분리 시키겠다는 뜻이다. 엄밀하게 말해서 cpu 에 할당 별도로 동작 시킴

printf("Connected Client IP: %s₩n", inet_ntoa(clnt_addr.sin_addr));

}// 클라이언트 아이피 주소 몇인지 알려준다. 다음 클라이언트가 들어오기 전까지 블로킹하고 있다.

close(serv_sock);

return 0; }

게임 클라이언트 부분

```
#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) // 에러 메시지 출력
{ fputs(msg, stderr);
fputc('₩n', stderr);
exit(1); }
```

void *send_msg(void *arg) //서버가 연결된 소켓이 인자로 넘어온다. pthread_create 의 마지막 인자는? 구동시키려하는 함수의 인자가 된다. 어떤 구조체가 올지 모르니까 void 를 사용. 너무 많아서 구조체로 만들면 좋다.

```
{ int sock = *((int *)arg); // 만들어진 서버의 소켓이 전달됨
char msg[BUF_SIZE];

for(;;) //계속 사용자의 입력을 받고 서버 소켓에 전송하는 것을 반복한다.
{ fgets(msg, BUF_SIZE, stdin); // 입력을 받겠다는 소리이다. msg 에 저장
write(sock, msg, strlen(msg)); } // 입력한 메세지 값이 서버로 전달.

return NULL; }
```

```
void *recv_msg(void *arg)
{       int sock = *((int *)arg); //int 형 포인터로 형변환 한 arg  포인터를 저장. 위에서 void 로 받았기에
        char msg[BUF_SIZE];
        int str_len;

        for(;;) //실제 알고리즘 부분
        {   str_len = read(sock, msg, BUF_SIZE - 1); //서버에서 들어온 정보를 읽어 msg 에 넣는다.
            msg[str_len] = 0;
            fputs(msg, stdout);             } //수신하는 거. 서버로부터 수신받은 메세지를 msg 에 넣어주고 모
니터에 출력(write 0 과 같다)
        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);       // 소켓 생성

        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, (sp)&serv_addr, sizeof(serv_addr)) == -1) // 이걸 하는 순간 서버에서 소켓을 acce
pt 하는 것이다.
                err_handler("connect() error");

        pthread_create(&snd_thread, NULL, send_msg, (void *)&sock); //송신. 인자는 (void *)&sock 이다.
        pthread_create(&rcv_thread, NULL, recv_msg, (void *)&sock); //수신
// 송신과 수신을 분리하기 위해 쓰레드를 만들었다. Fork 와 같다. 위에서 만들고 join 으로 동작. Join 하는
순간 쓰레드는 동작을 시작한다.
        pthread_join(snd_thread, &thread_ret); // 송, 수신이나 쓰레드가 없으면 끝남.
        pthread_join(rcv_thread, &thread_ret);

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

파일 전송 서버

서버 코드

```
#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)
{   int serv_sock, clnt_sock, 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);                                  }

  fd = open("file_server.c", O_RDONLY); // 무언가 전송하고, 다른 것을 하고 싶으면 여길 변경.
  serv_sock = socket(PF_INET, SOCK_STREAM, 0); // 여기 빼고 기존이랑 비슷하다. 소켓 생성

  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, (sap)&serv_addr, sizeof(serv_addr)) == -1)
    err_handler("bind() error");

if(listen(serv_sock, 5) == -1) //  클라이언트 5 명 받음.
  err_handler("listen() error");

clnt_addr_size = sizeof(clnt_addr);

clnt_sock = accept(serv_sock, (sap)&clnt_addr, &clnt_addr_size);

for(;;) // 실제 알고리즘 부분
{   read_cnt = read(fd, buf, BUF_SIZE); //읽은 바이트 수를 리턴. buf 에 file_server.c 를 읽어서 저장

    if(read_cnt < BUF_SIZE) //read_cnt 는 읽은 바이트 수
    {   write(clnt_sock, buf, read_cnt); //clnt_sock 에 읽은 바이트 수 만큼 buf 를 쓴다.
        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;       }
```

클라이언트 코드

```
#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]));

  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) // 실제 알고리즘 부분은 2 줄
     write(fd, buf, read_cnt);

  puts("Receibed File Data");
  write(sock, "Thank you", 10);
  close(fd);
  close(sock);
  return 0;        }
```

컴파일하고 옆의 사람과 파일을 전송할 수 있다. 근데 한 사람과 파일 전송이 끝나면 실행이 종료된다.

## gethostbyname()

```
#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 <port>₩n", argv[0]);
      exit(1);                                          }

    host = gethostbyname(argv[1]); //hostent 구조체(이름, 별명, 주소타입, ip address 등)를 얻어온다.

    if(!host) // host 가 없으면 에러
      err_handler("gethost.....error!");

    printf("Official Name : %s₩n", host->h_name); // host 이름 출력

    for(i=0; host->h_aliases[i]; i++)//별칭이 있다면 출력한다. 없을 수도 있다.
       printf("Aliases %d : %s₩n", i+1, host->h_aliases[i]);

    printf("Address Type: %s₩n", (host->h_addrtype == AF_INET) ? "AF_INET" : "AF_INET6");
//IP 타입을 알아보는 것. host 주소 타입 출력. IPv4 이면 AF_INET 을 IPv6 이면 AF_INET6 출력

    for(i=0; host->h_addr_list[i]; i++)//IP 주소 포트 번호는 80
      printf("IP Addr %d: %s₩n", i+1, inet_ntoa(*(struct  in_addr *)host->h_addr_list[i]));

 return 0;         }
```

결과

```
sue100012@sue100012-Z20NH-AS51B5U:~/project/4_01$ ./a.out naver.com
Official Name : naver.com
Address Type: AF_INET
```

IP Addr 1: 125.209.222.142
IP Addr 2: 125.209.222.141
IP Addr 3: 210.89.160.88
IP Addr 4: 210.89.164.90

## 1. pthread 활용법

쓰레드를 구현할 때 가장 많이 사용 되는 것이 pthread(POSIX thread)이다. 그럼 쓰레드(thread)는 무엇이며, 왜 이용하는가?

쓰레드는 여러개의 클라이언트를 처리하는 서버/클라이언트 모델의 서버프로그래밍 작업을 위해서 주로 사용된다. 비슷한 일을 하는 fork 에 비해서 빠른 프로세스 생성 능력과 적은 메모리를 사용하여 Light Weight 프로세스라고 불리기도 한다.

쓰레드는 프로세스와 달리 다른 쓰레드들과 **메모리를 공유**한다. (보통 프로세스는 자기 자신만의 메모리 영역을 가진다). 이렇게 전역 메모리를 공유하게 되므로 fork 방식에 비해서 좀 더 작은 메모리를 소비하게 된다. 이는 fork 에 비해 쓰레드가 빠른 프로세스 생성 능력과 메모리 공유에 의한 적은 메모리 사용과 쓰레드 간의 좀 더 쉬운 정보 공유가 가능하게 해준다. 이러한 더 빠른 수행 능력을 보이는 이유는 fork 가 기본적으로 모든 메모리와 모든 기술자(파일 기술자등)을 C.O.W(Copy on Write) 방식으로 자식 프로세스를 복사하는데, 쓰레드는 많은 부분을 공유하기 때문이다. C.O.W 도 효율적이나 메모리 자원을 공유하는 것보다는 느릴 수 밖에 없다. 또한 fork 는 부모와 자식이 같이 통신을 하기 위해서는 IPC 를 사용해야 하며, 이는 복잡한 작업이 될 수도 있는데, 쓰레드는 메모리를 공유함으로 IPC 의 사용을 줄이면서도 쓰레드간 정보 교환을 쉽게 할 수 있다.

쓰레드간에 서로 공유하는 자원 : 작업디렉토리 , 파일지시자들, 대부분의 전역변수와 데이타들
UID 와 GID, signal
쓰레드가 고유하게 가지는 자원 : 에러 번호, thread 우선순위, 스택, thread ID, 레지스터 및 스택지시자

그러나 메모리를 공유한다는 것은 장점만 있는 것은 아니다. 모든 쓰레드가 같은 메모리 공간을 공유하게 되므로, 하나의 쓰레다그 잘못된 메모리연산을 하게 되면, 모든 프로세스가 그 영향을 받게 된다. fork 등을 통한 프로세스 생성방식은 메모리를 공유하지 않고, OS 가 각각의 프로세스를 보호하는 것도 있어, 한 프로세스의 문제는 해당 프로세스의 문제로 끝나게 된다. 그러나 쓰레드는 이러한 프로세스 보호를 기대할 수 없다. 하나의 쓰레드에 문제가 생기면 전체 쓰레드에 문제가 생길 가능성이 매우 크다. 또한 메모리를 공유하기 때문에 다른 쓰레드가 연산 중일 때, 다른 쓰레드가 접근해서 연산에 영향을 줄 수 있다. 이처럼 여러 task 가 동시에 접근해서 정보가 꼬일 수 있는 공간을 Critical Section(임계 영역)이라고 한다.

이 임계 영역 문제를 해결해주는 것 중 하나가 **pthread_mutex_init()**이다. mutex 는 여러 개의 쓰레드가 공유하는 데이터를 보호하기 위해서 사용되는 도구이다. 보호하고자 하는 데이터를 다르는 코드영역을 '**한번에 하나의 쓰레드만**' 실행 가능 하도록 하는 방법으로 공유되는 데이터를 보호한다.

int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutex_attr *attr);

pthread_mutex_init 는 mutex 객체를 초기화 시키기 위해서 사용한다. 첫번째 인자로 주어지는 mutex 객체 mutex 를 초기화시키며, 두번째 인자인 attr 를 이용해서 mutex 특성을 변경할수 있다. 기본 mutex 특성을 이용하기 원한다면 NULL 을 사용하면 된다. pthread_mutex 가 코드에서 사용 될 때, pthread_mutex_t 는 구조체이고 pthread_mutex_lock() 는 critical section 시작하는 구간으로 해당 쓰레드 외의 다른 쓰레드의

접근은 막아 준다. pthread_mutex_unlock() 는 critical section 종료되는 구간으로 이제 다른 쓰레드 등의 접근을 허용해서 필요한 연산을 하도록 해준다. 즉, pthread_mutex_lock() 과 pthread_mutex_unlock() 사이의 critical section 은 한번에 하나의 쓰레드만 수행할 수 있고, 먼저 이 critical section 에 진입한 쓰레드가 종료할때까지 다른 쓰레드는 대기상태에 있다가, 앞선 쓰레드가 critical section 을 끝내고 빠져나오면 진입하게 된다.

pthread_create()

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

쓰레드 생성을 위해서 사용한다. 즉, **새로운 쓰레드를 생성**한다. 첫번째 인자인 thread 는 쓰레드가 성공적으로 생성되었을때 생성된 쓰레드를 식별하기 위해서 사용되는 쓰레드 식별자이다. 생성된 쓰레드의 ID 를 저장할 변수 포인터가 오는 것이다. 두번째 인자인 attr 은 쓰레드 특성을 지정하기 위해서 사용하며, 기본 쓰레드 특성을 이용하고자 할경우에 NULL 을 사용한다. 3 번째 인자인 start_routine 는 분기시켜서 실행할 쓰레드 함수이며, 4 번째 arg 는 쓰레드 함수의 인자이다. 세번째 인자는 쓰레드가 생성되고 나서 실행될 함수가 온다는 것이다. (함수명도 주소값을 가짐으로 포인터로 볼 수 있다). 즉, 새로운 쓰레드는 start_rountine 함수를 arg 인자로 실행시키면서 생성된다. 성공적으로 생성될경우, 식별자인 (pthread_t *) thread 에 쓰레드 식별번호를 저장하고, 0 을 리턴한다. 실패했을 경우는 0 이 아닌 에러코드 값을 리턴한다. 생성된 쓰레드는 pthread_exit 를 호출하거나 또는 start_routine 에서 return 할 경우 제거된다.

pthread_join

int pthread_join(pthread_t thread, void **retval);

특정 pthread 가 종료될 때까지 기다리다가 특정 pthread 가 종료 시 자원 해제시켜준다. join 된 쓰레드 (종료된 쓰레드)는 모든 자원을 반납하게 된다. 성공하면 0, 실패하면 에러 코드를 반환한다. thread 는 어떤 pthread 를 기다리는지 정하는 식별자이고 **retval 은 pthread 의 반환 값, 포인터로 값을 받아온다. NULL 이 아닌 경우 해당 포인터로 쓰레드의 리턴 값을 받아올 수 있다.

ptrhead_detach

int pthread_detach(pthread_t thread);

**쓰레드를 분리**시킨다. 일반적으로 pthread_create 를 사용하면 쓰레드가 종료되더라도 사용한 자원이 모두 해제 되지 않는다. 반면에 pthread_join 를 사용하면 종료될 때, 자원이 반납된다. ptread_detach 는 pthread_create 를 종료될 때 모든 자원을 해제하게 해준다. 성공하면 0, 실패하면 에러코드 반환한다. thread 는 분리시킬 쓰레드 식별자이다.

- 쓰레드 종료시 자원 반납을 하여 메모리 누수 (memory leak) 를 방지할때, pthread_join(), pthread_detach() 외에도 pthread_create() 이 attribute 값을 주어 애초부터 종료시 자원 반납을 하는 쓰레드로 생성할수도 있다.

pthread_exit

void pthread_exit(void *retval);

*retval 은 현재 실행중인 thread 를 종료시킬 때 사용한다. 보통 pthread_exit 가 호출되면 cleanup handler 가 호출되며 보통 리소스 해제하는 일을 수행한다. 쓰레드를 종료시킬때 자원등을 반납해야 되는 등의 clean 하는 프로세스를 해야 할 필요가 있다.

pthread_self
pthread_t pthread_self(void);
현재 실행중인 pthread 의 식별자를 반환한다.

pthread_cleanup_push
void pthread_cleanup_push(void (*routine)(void *), void *arg);
pthread_exit 가 호출될 때 호출된 handler 를 정하는 함수이다. 보통 자원 해제용이나 mutex lock 를 해제할 때 사용한다.

pthread_cleanup_pop
void pthread_cleanup_pop(int execute);
cleanup handler 를 제거하기 위해 사용되는 함수이다. execute 가 0 일 경우 바로 cleanup handler 를 제거하고 그 외의 값을 가질 때는 한번 실행한 후 제거한다.

※ 쓰레드(tread) 사용시 컴파일 할 때, 반드시 -lpthred 옵션을 주어야 컴파일 된다.
※ 헤더 파일은 #include <pthread.h> 이고 pthread_t 는 pthread 의 자료형을 의미한다.

## 2. 네트워크 프로그래밍 기본기

## 3. 기타 정리

bind error
bind 함수는 소켓에 IP 주소와 포트 번호를 지정한다. 이로서 소켓을 통신에 사용할 수 있도록 준비가 된다. 성공할 경우 0 을 실패했을 경우에는 -1 을 반환한다.
close 함수로 소켓을 소멸시켜도 커널은 바로 소멸시키지 않고 몇 초 정도 유지시킨다. 이는 클라이언트와 처리되지 않은 전문을 마저 처리하기 위해서이다. 이 때, bind 된 소켓이 아직 소멸되지 않았는데 같은 주소, 같은 포트로 또 다른 소캣이 bind 를 요청해서 발생하게 된다. 이때, 다시 bind 요청을 하게 되면, 이전 소켓의 생명 시간이 다시 초기화된다. 계속 bind 함수를 호출하면 이전 소켓은 더 오래 살아남게 된다. 어쩔 수 없이 이전 소켓의 Time-WAIT 상태가 끝날 때까지 기다려야 한다.

atoi 함수
C 형식 문자열을 정수로 변환하여 변환된 값을 리턴한다. 10 진법으로 표기된 문자열을 정수로 바꿀 수 있다 성공하면 변환된 정수를 반환하고 실패하면 0 을 반환한다. 단. 문자열은 정수로 되어 있어야 하며 알파벳 영문자, 특수 문자가 포함되면 해당 문자부터는 반환 하지 않는다. 또한, 처음부터 숫자가 아니면 0 으로 변환된다. atof 는 문자열을 배정도형(double)으로 바꿔주는 함수이고, atol 은 문자열을 long 형으로 바꿔주며, atoll 은 문자열을 longlong 령으로 바꾸어진다.