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

날 짜 : 2018 . 3. 29

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

<sigaction.c>

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

struct sigaction act_new;
struct sigaction act_old;

void sigint_handler(int signo)
{
	printf("Ctrl +C\n");
	printf("If you push it one more time then exit\n");
	sigaction(SIGINT , &act_old, NULL);
}

int main(void)
{
	// 시그널 두번째 인자에 행동을 등록하는것이다.
	act_new.sa_handler = sigint_handler;

	// 시그널을 막으려면 ignore 했는데
	// 이것도 특정 시그널을
	// empty 아무것도 막지 않겠다.?
	// 아주 중요한 것들을 할 때 시그널을 막아야 하는데 그때를 위한.
	sigemptyset(&act_new.sa_mask);

	//시그널은 이전 엑트를 보냈다.
	// '&' 가 들어있고 주소, 포인터를 전달한 것이다.
	// 무언가 주소를 통해서 값이 변경 시킨다는 것을 생각할 수 있다.
	// 포인터를 쓰면 함수는 원래 하나를 처리하는데 여러개를 처리 할 수 있다.
	// act_old 이전에 등록했던 시그널을 받아오는것.
	sigaction(SIGINT, &act_new, &act_old);

	while(1)
	{
		printf("sigaction test\n");
		sleep(1);
	}
	return 0;
}
```

<thread.c>

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

// 리턴도 모든지 하고 인자도 모든지 받을 수 있다.
// void 를 쓰는 이유.
void *task1(void *X)
{
	printf("Thread A Complete\n");
}

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

}

int main(void)
{
	pthread_t ThreadA, ThreadB;

	// a 와 b 에 뭔가를 넣어주려하고 있음. task1,task2 를 구동시키겠다.
	// 쓰레드를 이렇게 동작 시키겠다고 등록 시킨것이다.
	// 앞뒤로 아무것도 안붙였다는 뜻이다. null 은.
	pthread_create(&ThreadA, NULL, task1, NULL);
	pthread_create(&ThreadB, NULL, task2, NULL);

	// 조인을 하는 순간 메모리에 올리는 것이다.
	pthread_join(ThreadA, NULL);
	pthread_join(ThreadB, NULL);

		return 0;
}

// compile 시 -lpthread 를 뒤에 붙여야 컴파일이 작동한다.

// thread 는 종속적이다.
// 문제가 되는 곳은 critical section 이다.
// 그래서 lock 을 걸어준다.

// 그래픽 카드는 속도는 느리다 하지만 밴드위스가 높다.
// 그래픽카드는 cpu 가 1000 개정도 들어있다.
// 그래픽 카드는 밴드위스랑 갯수가 중요함
// 그래픽 카드는 병렬 처리, dsp 는 빠른 처리를 할때 특화되어 있다.
// cpu 는 순차 처리에 특화 되어있다.
```

<kill.c>

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

 // 프로세스가 프로세스를 죽이는것을 보여주기위함 프로그램.
int main(int argc, char *argv[])
{
        if(argc <2)
                printf("Usage : ./exe pid\n");
        else
                kill(atoi(argv[1]),SIGINT);

        return 0;
}
```

<test.c>

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

void gogogo(int voidv)
{
        printf("SIGINT Accur!\n");
        exit(0);
}
int main(void)
{
        // 시그널 종류 , 행동지침
        signal(SIGINT, gogogo);
        for(;;)
        {
                printf("kill Test \n");
                sleep(2);
        }

        return 0;
}

// gcc -o test test.c
// ./test & 백그라운드에서 실행한다는 뜻이다.
// 백그라운드라 터미널에서 계속 살아난다.
// & 로 실행을 하면 pid 값이 나옴.
// ctrl + c 가 안꺼지는 이유는 백그라운드로 들어가면 제어권이 사라져서 그럼.
```

<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;

	char msg[] = "Hello Network Programming";
	if(argc != 2)
	{
		printf("use: %s <port>\n", argv[0]);
		exit(1);
	}
	// 소켓은 파일이다, 원격에 있는 파일을 말하는 것. 그래서 ipc 를 쓴다.
	// 네트워크는 결국 원격의 ipc , 원격의 세마포어이다.
	// 소켓을 열어라 소크 스트림은  tcp 소켓을 사용한다는 뜻이다.
	// 리턴은 파일 디스크립터가 나온다.
	serv_sock = socket(PF_INET, SOCK_STREAM, 0);

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

	// 서버 어드레스의  메모리를 한번 지워준다.
	memset(&serv_addr, 0, sizeof(serv_addr));
	serv_addr.sin_family =AF_INET; // 여기 패턴을 익혀야 한다.
	// 자신의 주소를 받겠다. 127.0.0.7 = 로컬호스트
```

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

	//스코프 바인딩. 서버의 ip 주소를 세팅한다. 127.0.0.7 이됨
	if(bind(serv_sock, (sap)&serv_addr, sizeof(serv_addr)) == -1)
		err_handler("bind() error");

	// 5 명 까지 받겠다라는 뜻이다. 실제로 클라이언트  기다리는 곳이다.
	if(listen(serv_sock, 5) == -1)
		err_handler("listen() error");

	clnt_addr_size = sizeof(clnt_addr);
	// accept 는 서버 소켓이 클라이언트의 실제 접속 허용을 해주는 곳이다.
	clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr, &clnt_addr_size);

	if(clnt_sock == -1)
		err_handler("accept() error");
	// 원격에 있는 클라이언트에게 롸이트를 한다.
	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)
	{
		printf("use: %s <IP> <port> \n", argv[0]);
		exit(1);
	}

	// 네트워크 상의 파일 디스크립터를 받는다.
	// 파일의 오픈과 같은 역할이라고 보면 된다.
	sock = socket(PF_INET, SOCK_STREAM, 0);

	if(sock == -1)
		err_handler("socker() error");
	// 여기..... 부터 4 줄이 tcp ip 를
	// 서버어드레스를 초기화 하고 있다.
	memset(&serv_addr, 0, sizeof(serv_addr));
	serv_addr.sin_family = AF_INET;
	// 우리가 입력한 ip 주소가 들어감
	serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
	serv_addr.sin_port = htons(atoi(argv[2]));// 포트번호가 들어감.

	// 자신의 파일 디스크립터를 가지고 서버에 연결함.
	if(connect(sock, (sap)&serv_addr, sizeof(serv_addr))==-1)
		err_handler("connect() error");
	// 서버 디스크립터가 받아온 정보에서 msg 에 서버에서 받은것을 보냄.
	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;
}
```

<serverfork.c>

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/socket.h>


int main(void)
{

	// 소켓도 파일이다. ls -l 옵션에 's' 에 속한 배열 0 은 tcp 1 은 udp 이다.
	int fd[3];
	int i;
	fd[0] = socket(PF_INET, SOCK_STREAM, 0); // tcp 이다.
	fd[1] = socket(PF_INET, SOCK_DGRAM, 0);  // udp 이다.
	fd[2] = open("test.txt", O_CREAT | O_WRONLY | O_TRUNC);

	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;
}
```

<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");
	// 데이터가 끊기지 않게 할려고 쓴다.
	// 중간에 데이터가 우회하는것은 그래프 알고리즘이다.
	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 conut : %d\n", str_len);
	return 0;
}
```

<inet_addr.c>

```
// 주소체계라고 한다. big 엔디안 little 엔디안...
// 서로 같은 규격으로 맞추지 않으면 꼬인다.
// 네트워크 형식에 맞게 변경을 시킨다.

// 처음이 0 이라서 굳이 출력이 안되는 것이다. 그래서 09050703
// 가 아니라

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

int main(int argc, char **argv)
{
        char *addr1 = "3.7.5.9";
        char *addr2 = "1.3.5.7";

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

        conv_addr = inet_addr(addr2);
       if(conv_addr == INADDR_NONE)
                printf("Error!\n");
       else
                printf("Network Ordered Integer Addr:%#lx\n",conv_addr);
       return 0;
}
```

sigaction()

sigaction() 함수는 signal() 보다 향상된 기능을 제공하는 시그널 처리를 결정하는 함수입니다. signal() 에서는 처리할 행동 정보로 시그널이 발생하면 호출이될 함수 포인터를 넘겨 주었습니다. 그러나 sigaction()에서는 struct sigaction 구조체 값을 사용하기 때문에 좀더 다양한 지정이 가능합니다.

```
struct sigaction {
// 시그널을 처리하기 위한 핸들러. SIG_DFL, SIG_IGN 또는 핸들러 함수
    void (*sa_handler)(int);
      // 밑의 sa_flags 가 SA_SIGINFO 일때
    void (*sa_sigaction)(int, siginfo_t *, void *);
      // sa_handler 대신에 동작하는 핸들러

    sigset_t sa_mask;           // 시그널을 처리하는 동안 블록화할 시그널 집합의 마스크
    int sa_flags;               // 아래 설명을 참고하세요.
    void (*sa_restorer)(void);  // 사용해서는 안됩니다.
}
```

옵션	의미
SA_NOCLDSTOP	signum 이 SIGCHILD 일 경우, 자식 프로세스가 멈추었을 때, 부모 프로세스에 SIGCHILD 가 전달되지 않는다.
SA_ONESHOT 또는 SA_RESETHAND	시그널을 받으면 설정된 행도을 취하고 시스템 기본 설정인 SIG_DFL 로 재설정된다.
SA_RESTART	시그널 처리에 의해 방해 받은 시스템 호출은 시그널 처리가 끝나면 재시작한다.
SA_NOMASK 또는 SA_NODEFER	시그널을 처리하는 동안에 전달되는 시그널은 블록되지 않는다.
SA_SIGINFO	이 옵션이 사용되면 sa_handler 대신에 sa_sigaction 이 동작되며, sa_handler 보다 더 다양한 인수를 받을 수 있습니다. sa_sigaction 이 받는 인수에는 시그널 번호, 시그널이 만들어진 이유, 시그널을 받는 프로세스의 정보입니다.

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

인수.

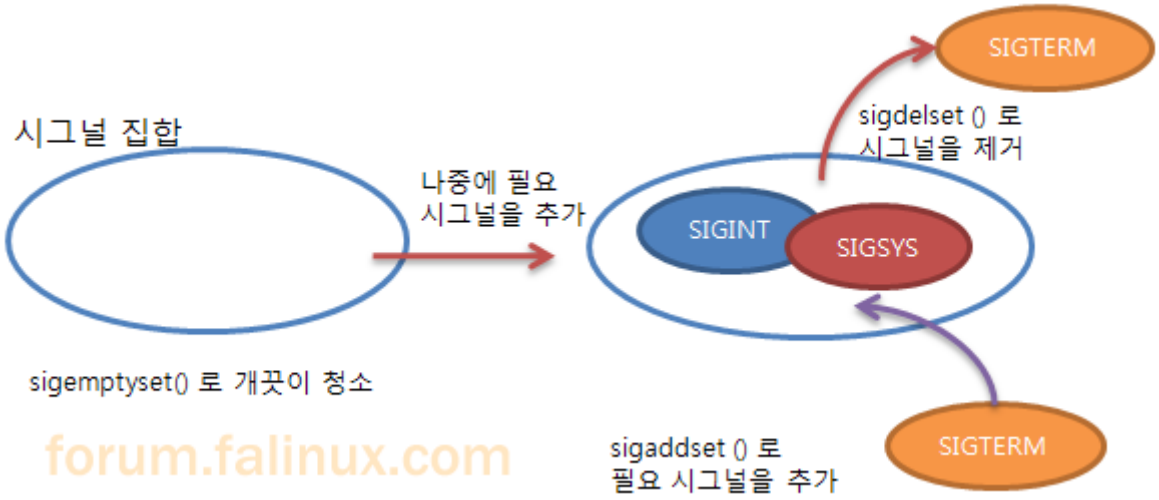
int signum 시그널 번호
struct sigaction *act 설정할 행동. 즉, 새롭게 지정할 처리 행동
struct sigaction *oldact 이전 행동, 이 함수를 호출하기 전에 지정된 행동 정보가 입력됩니다.

반환값

0 성공
-1 실패

sigemptyset() 시그널 집합 내용을 모두 삭제

리눅스에서 매우 다양한 시그널이 있습니다. 이 시그널을 하나씩 처리할 때가 있다면 여러 개를 하나로 묶어서 한꺼번에 처리하는 것이 편할 때가 있습니다. 이렇게 필요한 시그널을 하나의 집합으로 묶어 주는 함수가 준비되어 있는데 그 중에 하나가 빈 주머니를 만들듯이 빈 집합을 만들어 주는 함수가 **sigemptyset()**입니다. 이제 여기 다가 아래의 시그널 중에 필요한 시그널을 하나씩 넣어서 필요한 시그널 집합을 만드시면 되겠습니다.



int sigemptyset(sigset_t *set);

인수

sigset_t *set 시그널 집합 변수

반환.

0 집합 변수를 성공적으로 비웠음
-1 실패했음

fputs(const char *str, FILE *stream)

-stream 에 문자열 (str)을 쓰겠다.

fputc(int charater, FILE *stream)

- charater 를 int 형으로 바꿔서 stream 으로 저장.

sockaddr_in (구조체)

– ip 와 포트번호를 가지고 있는 구조체이다.

sockaddr (구조체)

- 소켓 주소의 와꾸를 잡는 녀석, 소켓의 주소를 담는 기본 구조체 역할

- sockaddr_in 와 달리 ip 와 포트번호가 같이 쓰여서 쓸때 불편하다

서버

```
// 여기 패턴을 익혀야 한다.
// 서버 어드레스의  메모리를 한번 지워준다.
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family =AF_INET;
// 자신의 주소를 받겠다. 127.0.0.7 = 로컬호스트
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(atoi(argv[1]));
```

int bind(**int** sockfd, **struct** sockaddr *myaddr, socklen_t addrlen);

int sockfd: 소켓 디스크립터

struct sockaddr *myaddr :

주소 정보로 인터넷을 이용하는 AF_INET 인지 시스템 내에서 통신하는 AF_UNIX 에 따라서 달라집니다.

인터넷을 통해 통신하는 AF_INET 인 경우에는 struct sockaddr_in 을 사용합니다.

socklen_t addrlen : myadd 구조체의 크기

반환

0 : 성공

-1 : 실패

int socket(**int** domain, **int** type, **int** protocol);

socket() 함수는 소켓을 생성하여 반환합니다. 소켓은 파일이다.

int domain :인터넷을 통해 통신할 지, 같은 시스템 내에서 프로세스 끼리 통신할 지의 여부를 설정합니다.

domain	domain 내용
PF_INET, AF_INET	IPv4 인터넷 프로토콜을 사용합니다.
PF_INET6	IPv6 인터넷 프로토콜을 사용합니다.
PF_LOCAL, AF_UNIX	같은 시스템 내에서 프로세스 끼리 통신합니다.
PF_PACKET	Low level socket 을 인터페이스를 이용합니다.
PF_IPX	IPX 노벨 프로토콜을 사용합니다.

int type : 데이터의 전송 형태를 지정하며 아래와 같은 값을 사용할 수 있습니다.

type	type 내용
SOCK_STREAM	TCP/IP 프로토콜을 이용합니다.
SOCK_DGRAM	UDP/IP 프로토콜을 이용합니다.

int protocol : 통신에 있어 특정 프로토콜을 사용을 지정하기 위한 변수이며, 보통 0 값을 사용합니다.

반환

-1 이외 : 소켓 식별자

-1 : 실패

socket(PF_INET, SOCK_STREAM, 0);

- IPv4 인터넷 프로토콜을 사용한다.

- 소켓을 열어라 소크 스트림은 tcp/ip 소켓을 사용한다는 뜻이다.

- 리턴은 파일 디스크립터가 나온다.

int connect(**int** sockfd, **const struct** sockaddr *serv_addr, socklen_t addrlen);

connect() 함수는 생성한 소켓을 통해 서버로 접속을 요청합니다.

int sockfd : 소켓 디스크립터

struct sockaddr *serv_addr : 서버 주소 정보에 대한 포인터

socklen_t addrlen : struct sockaddr *serv_addr 포인터가 가르키는 구조체의 크기

반환

0 : 성공

-1 : 실패

http://forum.falinux.com/zbxe/index.php?document_srl=438301&mid=C_LIB

이곳에 밑에 서버에 관한 설명이 잘 나와있다. 시간날때마다 읽어보기!!

<리틀 엔디안 , 빅엔디안 >

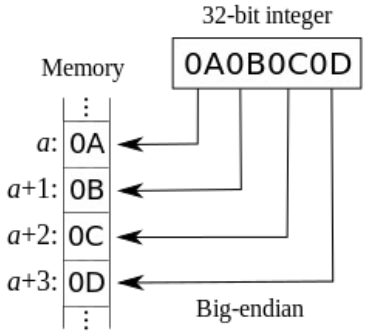
엔디언은 컴퓨터의 메모리와 같은 1 차원의 공간에 여러 개의 연속된 대상을 배열하는 방법을 뜻하며, 바이트를 배열하는 방법을 특히 바이트 순서(Byte order) 라 한다.

엔디언은 보통 큰 단위가 앞에 나오는 빅엔디언(Big-endian) 과 작은 단위가 앞에 나오는 리틀 엔디언(Little-endian)으로 나눌 수 있으며, 두경우에 속하지 않거나 둘을 모두 지원하는 것을 미들 엔디언(Middle-endian) 이라 부르기도 한다.

빅엔디안은 최상위 바이트(MSB) 부터 차례로 저장하는 방식이며,
리틀 엔디안은 최 하위 바이트(LSB) 부터 차례로 저장하는 방식이다.

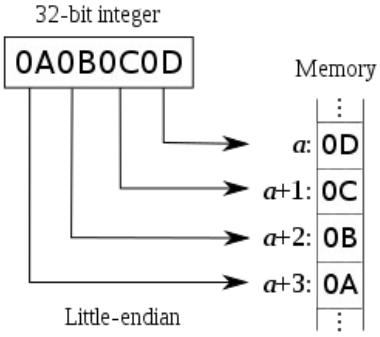
우선 리틀 엔디안은 주로 인텔(Intel)프로세스 계열에서 사용하는 바이트 오더 이다.
메모리 시작 주소가 하위 바이트부터 기록된다는 것이고
반대로 빅 엔디안은 메모리 시작 주소에 상위 바이트부터 기록된다.
주로 UNIX 시스템인 RISC 프로세서 계열에서 사용하는 바이트 오더이다.

빅 엔디안 읽는 순서는 왼쪽 → 오른쪽 순으로 읽으며 , 사람이 보기에는 가장 편한 방식이다.



0A0B0C0D 순으로 나옴.

리틀 엔디안 읽는 순서는 오른쪽→ 왼쪽 순으로 읽으며, 산술연산유닛(ALU)에서 메모리를 주소가 낮은 쪽부터 읽기 때문에 속도가 더 빠르다.



0D0C0B0A 순으로 나옴.

빅엔디안을 안쓰는 이유는 데이터를 다른 시스템으로 전송할 때 서로 다른 데이터 저장 방식의 시스템끼리 통신하게 되면 전혀 엉뚱한 값을 주고 받을 수 있다.